Contents

List of Definitions

List of Propositions, Lemmas and Theorems

Theorems:

Sequential Omicron:

CHAPTER 1

Object Component Substitutability

An Introduction

This chapter presents the theme of the thesis: object component systems and substitutability of object components.

Section 1.1 presents the background of the work: the need for a formalisation of object-oriented components. Furthermore it presents those characteristics of the models of object component systems which distinguish them from other models, such as communicating processes and functional models.

Section 1.2 introduces refinement relations between components. These are used to express substitutability properties of components. The central concept in relation to substitutability is that of reliable refinements. Simply and informally expressed, the important substitutability property of reliable refinements is:

> The context of some component will not note any difference if it collaborates with the component or if the component is replaced by a component which is a reliable refinement of itself.

Properties required by refinement relations in order to ensure reliability of refinements are expressed through the central proposition of the thesis: the substitution proposition.

Section 1.3 discusses applicability of the reliability property. It presents some examples of situations where it is useful to have components with substitutability properties.

Section 1.4 presents the goals of the presented work: to give a formal framework to reason about object component substitutability. This section also gives an introduction to the remaining chapters.

Section 1.5 summarises the main contributions of the thesis.

1.1 Object Component Substitution

This section will give an introduction to object component substitution and related concepts. First, the history of object-oriented systems is briefly described and the terms object component system (OCS) and component developer are introduced. Next, the motivation for the present work is given. In short this is to create a formal framework for reasoning about substitutability of OCS components. After the subsection on motivation, characteristics of object component systems are presented. Here concepts such as system, object, component and refinement are defined in accordance with the OCS tradition. Section 1.2 introduces the main theme of the thesis: reliable substitution of components.

Object-oriented Systems

Object-oriented technology stems from the programming language Simula (Dahl et al. 1968). However the term "object-oriented" was coined by Alan Key's Smalltalk group at Xerox PARC in Palo Alto, California in the period from 1970 to 1980. A paper referring to "Object-oriented systems" was first published in Byte August 1981 by the Xerox PARC group.

Smalltalk was the first object-oriented system, meaning that Smalltalk is not just a language, but a standardised set of object-oriented libraries covering such things as file system, windowing system, text editing and process switching. Well structured libraries created with subclassing and component substitution in mind, are called Frameworks, the first being the Smalltalk-80 Model-View-Controller[5] user interface Framework, documented in (Krasner and Pope 1988), (Deutsch 1989), (Goldberg 1990).

Apart from a very rudimentary version of Smalltalk, called Methods, the first commercial object-oriented Frameworks were found in the Lisa toolkit written in Classcal (an object oriented version of Pascal). These Frameworks were delivered with the Lisa Computer from Apple Computer in 1982/84. Lisa was discontinued when the MacIntosh was introduced. The work on the Lisa Toolkit and Classcal was continued in the MacIntosh project and was reworked and reappeared as the MacIntosh Toolkit and Object Pascal.

In 1984 Tektronix made Smalltalk-80 (Goldberg and Robson 1983) commercially available, but this implementation did not become widely spread. The language C++ (Stroustrup 1986) also appeared in the middle 80s, something which contributed greatly to the spreading of object-oriented technology. Another contribution was Tektronix's decision to discontinue their Smalltalk development. This created a number of laid off Smalltalkers which started to work outside Tektronix and thereby spreading object-oriented ideas across the USA.

The number of object-oriented programming languages today is large and new languages and object-oriented extensions to existing languages are steadily being published. The book (Blair et al. 1991) gives a general introduction and discussion of object-oriented concepts.

The latest development in commercial object-oriented technology is the explosion of Java (see, eg, http://www.javasoft.com). The Java language has combined many of the good features from all previous commercial object-oriented languages. Gosling, the "father" of Java, said in his talk at OOPSLA'96 that Java owes much to Simula and that "Java is a stealth attack on C++ from Smalltalk programmers".

While C++ was just a programming language, Java, with its large standard libraries, is a complete object-oriented system, just as Smalltalk is. However, compared to Smalltalk, Java programming is less interactive and more rigid, but Java libraries give better support for the creation of distributed and multi process systems. It also has a much larger and greatly increasing user community and considerable commercial push.

Object component systems - OCS

Dividing a system into components is an idea advocated by most grand old men of computer science, eg, Dahl, Dijkstra and Hoare in (Dahl et al. 1972). The advantages of software composition have in the middle 90s again come into focus and are underlined by a number of papers and books, for instance (Udell 1994), (Hölzle 1993) and (Gamma et al. 1994). As (Henderson-Sellers 1993) points out, component composition is more efficient than traditional programming when the software components already exist and therefore do not have to be designed, programmed and tested for each application and system. Component composition is also safer when the components have been used in other applications and systems where they have been tested and errors removed.

[5] Designing a system using the Model-View-Controller ideas were first done by Trygve Reenskaug and others when developing a CAD/CAM ship-modelling system in the 1970'ies. They did not publish their invention, but Model-View-Controller was introduced to the Smalltalk group at Xerox PARC in 1978/79 by Reenskaug on a one year visit to the Smalltalk group.

Applications and distributed systems composed from software components can also be extended by substituting existing components with new ones. The term *extensible systems* is used for such systems. According to Pountain in (Pountain and Szyperski 1994) it is "the software engineers nirvana" when functionality can be changed or added to an extensible systems at runtime by substituting components. Smalltalk is an extensible system, and new additions to Java make it possible to create extensible systems in Java. Extensible systems can not be created in more traditional languages such as Simula and C++. However, Active X from Microsoft, allows programming of extensible systems in C++, Visual Basic and other languages.

Extensible component based object-oriented systems will in the following be referred to as object component systems, or OCS. The term *component developer* is used to refer to people designing and implementing object component systems.

Object-oriented technology has been widely applied to the design and implementation of components and extensible systems, typically advanced network applications with reactive user interfaces. General design strategies which exploit software components have been published, eg, (Johnson 1992), (Gamma et al. 1994), (Nordhagen 1989). Rules for creating good component designs, including how to create Frameworks, have also been published, see for instance (Johnson and Foot 1988). There are also a number of object-oriented system development methods which support the creation of object-oriented systems. Some examples are RDD (Wirfs-Brock et al. 1990), Objectory (Jacobson et al. 1992), BON (Waldén and Nerson 1995) and OOram (Reenskaug et al. 1995) (first report on OOram (Reenskaug and Nordhagen 1989), article (Reenskaug et al. 1992)). Lately, UML (the Unified Modelling Language see http://www.rational.com) has appeared and created a lot of activity and attracted much interest. Several of the central people which created their own development methods in the late 80ies and early 90ies have joined forces and merged their ideas into UML.

Various standards which support the creation and extension of OCS has been developed. Examples of such standards are COM, SOM, CORBA, OpenDoc and OLE, see Dr.Dobb's Journal of January 1995 for an overview. Lately the focus has been set on two competing solutions, namely ActiveX from Microsoft and "Java beans" from the group of companies supporting Java (see http://www.javasoft.com).

A huge number of formal models have been created based on object-oriented ideas. Some extend algebraic models with ideas taken from object-oriented languages, while other base their work on process models. The chapter on related work, chapter 9 refers to many such works. Lately there has been efforts comparing the various approaches such as (Bruce et al. 1997).

During the last two decades a large number of different object-oriented languages, systems, and model have emerged. As is shown above, lately there has been a clear trend towards consolidation and merging of ideas related to object component systems. The OCS view of systems, components and objects is presented further below.

Motivation

As an increasing number of object component systems are created and maintained, there is an increasing awareness that formal component descriptions are needed. The formal descriptions are needed to ensure that no errors are introduced when component systems are manipulated. System manipulation includes substituting an existing component with a new component and replacing a general design of a component with a more detailed design. If such manipulations can be done and it is ensured that no errors will be introduced, we say that we have *reliable substitution* of components. When we have reliable substitution, the old and new components/designs are similar in some sense. Formally defined relations which define similarity in ways which give properties like reliable substitution, are called *monotonic relations*. Many monotonic relations are defined for various formal models such as functional and process models. The relationship between reliable substitution and monotonicity is discussed further below while chapter 9 presents various models of systems and components.

Along with the growing awareness of the need for a formal definition of similarity based on the object-oriented system view, the component developers get frustrated with existing formalisms. The frustration is caused by the formal models and monotonic relation definitions not conforming with the developers intuitive understanding of components and similarity between components. Discrepancies between component developers' and formalists' concepts are presented in chapter 9. There may be different reasons for this discrepancy. One reason may be that some formalists use many of the same concepts, but do *not* focus on substitutability of OCS components, while this is the main concern of the component developers. Other formalisms are based on concepts which are very different from the concepts used in component system designs. It is therefore difficult to see if the monotonic relations are related to the component developers' concept of similar components and reliable substitution.

It is recognised by many researchers and expressed in, eg, (Wegner 1994) and (Pountain and Szyperski 1994) that at present there is weak support for formal reasoning about components in object component systems while at the same time there is a need for such support.

The main motivation for the present work is the usefulness of OCS components and the lack of and need for formal support for reasoning about such components.

Object Component System Design Characteristics

In order to make a formal framework for reasoning about object component systems, it is necessary to capture the characteristics of such systems. These characteristics are found by looking at object-oriented programming and design languages and by studying, using or developing the designs of preferably large and successful object component systems.

Many of the characteristics found in object-oriented languages have been adopted into formal models to varying degrees. These characteristics include the idea of an object as an entity which combines state and functionality and where an object has an identity independent of its state. Other ideas taken from object-oriented languages are encapsulation, classes and inheritance, virtual procedures and dynamic binding.

The characteristics which are only found by studying, using or developing the designs have been added to formal models to a much lesser extent than those explicitly found in object-oriented languages. The following list summarises the characteristics of object component system designs. The list has been created based on experience with different programming libraries and systems, mainly applications in different versions of Smalltalk-80 from 1984 to 1994, but also libraries and systems in Simula, Smalltalk-78 (substantially different from Smalltalk-80), Smalltalk-V, Object Pascal, C++ and Java. The characteristics are also found in examples in books and papers on object-oriented design such as (Cook and Daniels 1994), (Cox and Novobilski 1991), (Deutsch 1989), (Gamma et al. 1994), (Jacobson et al. 1992), (Reenskaug et al. 1995), (Waldén and Nerson 1995), (Wirfs-Brock et al. 1990). Chapter 2 illustrates in detail the below characteristics by presenting an example design. For a full introduction to OCS design, the reader should consult one or more of the above mentioned books. In the description of the characteristics below, it is assumed that the reader is familiar with concepts from object-oriented programming languages such as variable, message and class. Formal definitions of the concepts used to describe characteristics of OCS designs are given in later chapters of this thesis.

Objects, Components and Systems:

A system: A system is a part of the real world which we *choose to regard as a whole*, separate from the rest of the world during some period of consideration. This definition is from (Holbækk-Hansen et al. 1975), a report on the first object-oriented system analysis method named Delta.

A system consists of objects: The system is viewed as consisting of objects, where each object is associated with a name distinguishing it from all other objects in the system. An object has variables and methods.

Object behaviour: When a system executes, the result is a sequence of actions (trace). We call the actions which stem from execution of an object for the object's behaviour. The actions are of three kinds: sending messages to objects, creating new objects and updating variables.

An object may be a component, but a component may consist of several objects: To make a system of objects more manageable, the system is partitioned into a number of collaborating parts. The partitioning is done by grouping objects into components. In some cases a single object will be a component, but in many cases a component will consist of several objects. Each object is found in one and only one component and will *never* move from one component to another.

Observable behaviour and component collaboration:

Observable behaviour: Components perform actions which involve themselves or other components in the system. We call the sequences of actions involving other components the component's observable behaviour. Components are characterised by their observable behaviour which typically include actions which send messages to objects in other components, actions which create new objects from templates in other components and actions which update shared variables.

Components form dynamic graph-like collaboration structures: If there are actions in one component which involve another component, we say that the two components collaborate. The components in a system will in this way form a collaboration structure. The collaboration structure is not just callers and callees forming tree-like structures, but a general graph of collaborating components. Since one component may get to know of new components through collaborations with other components, the collaboration structure can change during the components' existence. A component may be a client of the other components, the server for other components or both a client and a server.

Refinements:

Similar observable behaviour: A component has similar observable behaviour to another component if the other objects in the system observe similar behaviours from both components. The behaviour is similar if the same objects are involved in sequences of similar actions. For a full definition of similar actions refer to chapter 2.4. When two actions are similar they are typically two actions of the same kind. Two actions are similar if they send the same message to the same object, create objects from the same template or update the same variable.

Refinements of specifications: We call a component whose observable behaviour is similar to the observable behaviour described in a specification a *refinement* of the specification. In presence of non-deterministic behaviour, a refinement may have a more deterministic behaviour than the specification and still be seen as having a similar[6] observable behaviour.

Closed systems with boundary objects

The boundary between the inside and the outside of a system is found in objects with non-deterministic observable behaviour: A computer system is rarely closed since most systems communicates with users and external devices. The system-relevant behaviour of users and external devices are modelled as objects. Often these objects have a non-deterministic behaviour. Thus a deterministic description of the object's behaviour would be outside the scope of the modelling effort.

In object-oriented system development methods there are special names for such objects which model behaviour of users and external devices. In Objectory (Jacobson et al. 1992) these objects are called *actors* and in OOram (Reenskaug et al. 1995) they are called *environment roles*. In the following, such objects are called boundary objects.

An object system is a closed system: A system specification form a closed system when using boundary objects to model users and external devices. The system is closed in the sense that the objects in the system specification only collaborate with each other.

Operational specifications:

Operational specifications: In general, a component can be specified by stating the characteristics of the component explicitly, or by making an implementation (or implementation-like model) of a component where the implementation will displays the desired characteristics when executed. When specification is done by implementing a component with the desired characteristics, the specification is operational rather than declarative. The operational specification approach agrees with object-oriented design traditions such as Objectory, OOram and UML. An example of operational specifications in the form of a contract (Helm et al. 1990) is given in chapter 2.

There is no clear distinction between an operational specification and an implementation of a component since both are executable models. This agrees with object-oriented reuse-traditions where an implementation is seen as a specification. A system developer is allowed to replace the current component implementation with an alternative implementation if the two implementations have similar observable behaviour. In such cases the current component implementation functions as an operational specification, ie, a model which displays the desired properties. When implementations are seen as specifications, there may be refinements of implementations.

Design of components and contexts:

Single components are seldom designed in total isolation: Usually all parts in a collaboration are designed simultaneously. This is done since the quality of a component's design is judged by the flexibility and simplicity of the total collaboration pattern, not just the design of an individual component.

Context dependent specifications: A component's observable behaviour is specified for a given context. The context consists of other components. (Wegner 1995) calls such specifications partial specifications. A partial specification does not specify a component's observable behaviour for all possible contexts. Instead it specifies the component's behaviour in contexts which have similar observable behaviours to a given context. Therefore, a component's specification will also include the specification of the observable behaviour of the component's context. The observable behaviour of the context is also a partial specification in that it will only specify the observable behaviour of

---

[6] "similar" is here used to denote a non-symmetric relation. This is also done in (Milner et al. 1989a) and others. "bisimilar" is in this tradition used to denote a symmetric relation.

the context relative to the component, ie, the behaviour of the context as observed by the component.

Symmetry of component and context: When a design specifies the observable behaviour of both the component, and of the context of the component, then the design contains the specification of all parties in a collaboration. A component specification may then be viewed as both a specification of the component and as the specification of the context. In the latter case the objects which were originally found in the context are viewed as one or more components and the objects in the original component becomes a part of these components' context. There is therefore a symmetry of component and context.

There is component and context symmetry also in that both the component and the context may be substituted with components / contexts which have similar observable behaviour.

Separate development of new versions of component and context: When creating a design it is presupposed that new versions of the components will in general be created separately from each other in space and/or time. The underlying idea is that it should be possible to define standards and have a market for components. This idea is advocated by (Cox and Novobilski 1991), by (Meyer 1988) and (Meyer 1989) and others. Component standards are defined by specifying the components' observable behaviours.

1.2 Reliable Substitution

Specification of components relative to a context

Object specifications are usually done by defining object types. There are many substantially different approaches to defining types for objects (see chapter 9 on related work). In OCS design, object types are created in order to define categories of objects which may replace each other without introducing what is perceived as errors into a software system. In type-terminology, a category is formed by a type and its subtypes. In (Wegner and Zdonick 1988) the principle behind categorisation and typing of objects is formulated as the principle of substitutability:

An instance of a subtype can always be used in any context in which an instance of a supertype was expected.

Subtype relationships are usually written:

$C \leq D$ meaning that C is a subtype of D

The principle of substitutability is formulated for designs and specifications which describe a component by its behaviour in *any context*. OCS components are specified by their behaviour for *a given context*. This has to be taken into account when defining categories of substitutable components. Therefore, a refinement relation between OCS components must be expressed so that the context is included in the relation. We let

$C \leq_B D$ denote that C is a refinement of D in the context B.

A note on type specifications and components:
A type is defined by a specification expressing common properties of all elements of the type. The elements of a type form a set.

Often a relation is defined in order to identify all elements of a type. Such a relation is typically defined between a component and a type specification so that if a component stands in the specific relation to a given type specification, then the component will be included in the set of components of the type.

An OCS component can be viewed as a type specification. Then, all components which are refinements of the type specification can form the set of components of the type.

When OCS component are used as type specifications, there is no clear distinction between a component and a type specification. This is the view taken in this thesis. This is different from many traditional type systems, where a type specification and instance of a type are seen as two fundamentally different things.

Refinements have similar observable behaviour

When $C \leq_B D$ then B should be involved in similar sequences of actions when collaborating with C as when collaborating with D. This means that the sequences of actions involving B in B||C should be similar to the action sequences involving B in B||D. The set of all sequences of actions involving B which come from execution of B||C is called the observable traces of the system B||C relative to B. The objects in B are called *observers*. We can informally define the observable traces of a system relative to an observing component as follows:

> Let B||C denote the system consisting of the components B and C. The observable traces of this system relative to B will be the set of all possible sequences of actions which involve B objects and which occur when the system B||C executes. The observable traces of B||C relative to B is denoted Traces(B||C)/Obs(B).
>
> If both components have deterministic behaviour, the observable traces can be viewed as a single sequence of actions (one trace). If the components have non-deterministic behaviour, the observable traces can be viewed as a set of sequences of actions.

A full formal definition of observable behaviour is found in chapter 4. We say that Traces(B||C)/Obs(B) is *the observable behaviour of C relative to B*, thus understanding it as the observable behaviour we get from combining B and C, ie, B||C, not just the observable behaviour from only executing C.

When C is a refinement of D relative to B then C||B has similar observable behaviour to B||D relative to B, ie,

$C \leq_B D == Traces(B||C)/Obs(B) \approx Traces(B||D)/Obs(B)$

where ≈ denotes similarity of traces. Similarity of observable traces is defined in chapter 4.

Monotonicity and reliable substitution of components

For the subtype relation, the requirement is that the subtype relation is a monotonic. Monotonicity can be formally stated, where $\forall$ B : Γ means "for all types B":

$$\forall\ B, C, D : \Gamma \bullet C \leq D \ \Rightarrow\ B||C \leq B||D$$

This means that when C is combined with a context B, then the combination of B and C, denoted B||C, will be a subtype of the combination B||D. To be true to the type idea where types and elements of the type are seen as two fundamentally different things, this property is more correctly formulated as follows:

$$\forall\ B, C, D, E, F : \Gamma, c : C, d : D, b : B \ \bullet$$

$$C \leq D \ \Rightarrow\ ((b||d \in E \wedge b||c \in F) \Rightarrow F \leq E)$$

where $c \in C$ means that c is an instance of type C
b is some context for c and d, and
$(b||d \in E \wedge b||c \in F) \Rightarrow F \leq E$ means that if b||d is of type E and b||c is of type F then F is a subtype of E.

Intuitively, this means that any element of a subtype of D can reliably substitute any element of type D.

When $(b||d \in E \wedge b||c \in F) \Rightarrow F \leq E$ for any b, then we also have $\forall\ f : (f||b||d \in G \wedge f||b||c \in H) \Rightarrow H \leq G$. This can again be repeated adding more and more components. This property reflects the idea that subtype relations and the principle of substitutability is formulated with an open system in mind; a system which can expand by adding new observing and observable components to the existing components. A component must then be designed with all possible contexts in mind. This is as opposed to OCS design, where components are designed with a context with a certain observable behaviour in mind.

To get reliable substitution of similar OCS components, a property corresponding to monotonicity must be defined for the OCS refinement relation. When defining a monotonic-like property for refinement relations between OCS components it must take into account that the refinement relation is defined relative to a context with a specified observable behaviour. It must also be taken into account that each OCS system is closed. Then all components are defined in a specification and therefore no new components will be added. Instead, any component in the specification may be substituted with a refinement. Therefore, not only a single component may be substituted with a refinement, but components in the component's context may also be substituted with refinements.

Because the refinement relation is defined relative to a context and because an OCS system is closed and all components may be substituted, the OCS property corresponding to monotonicity will be quite different from traditional monotonicity. The OCS monotonicity-like property can be illustrated by considering a system with two components. If there is a system specification B||D and the following hold:

A is a refinement of B relative to D, ie, $A \leq_D B$ and
C is a refinement of D relative to B, ie, $C \leq_B D$

then we want to have a system A||C without any unanticipated effects or erroneous functionality. To avoid such effects and errors, the systems A||C and A||D must have the same A-observable behaviour, and likewise, the systems A||C and B||C must have the same C-observable behaviour. This can be stated:

A should have similar observable behaviour to B relative to C, ie, $A \leq_C B$ and
C should have similar observable behaviour to D relative to A, ie, $C \leq_A D$

The monotonicity-like property can be expressed as follows for a system which is divided into only two components (examples are given in chapter 2):

(*) $\quad A \leq_D B \wedge C \leq_B D \Rightarrow A \leq_C B \wedge C \leq_A D$

The components and component combinations can be illustrated as shown in figure F.1.1 below.

The conclusion $A \leq_C B \wedge C \leq_A D$ ensures that A in A||C will only observe actions which are similar to actions found in A||D and C in A||C will only observe actions which are similar to actions found in B||C. Therefore it can be assumed that nothing new and unexpected will happen in A||C.

The refinement relation can be compared to an assumption/guarantee specification, introduced in (Jones 1983). Such a specification asserts that a system Π provides a guarantee M if its environment F satisfies an assumption E. This corresponds to the following expression using the refinement relation:

$$\Pi \leq_E M \wedge F \leq_M E \Rightarrow \Pi \leq_F M$$

More on the relationship between the refinement relation and assumption/guarantee specification in chapter 9 on related work.

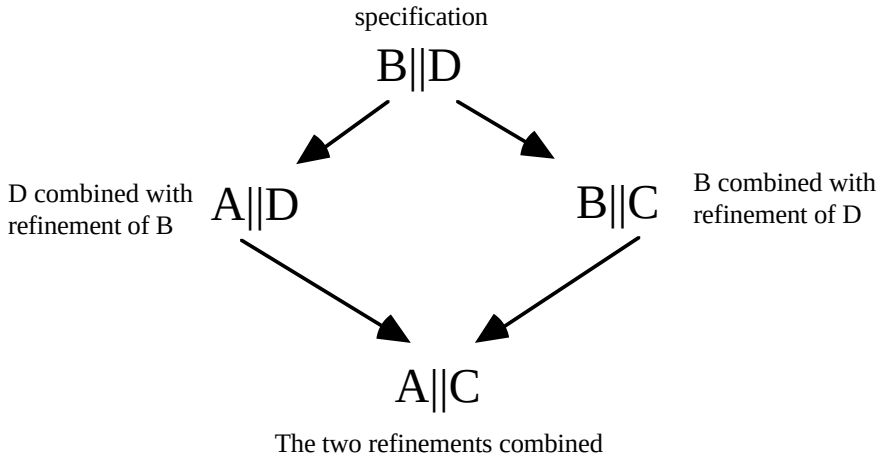


Figure F.1.1 Illustration of how specifications and refinements of components and contexts are combined. B and D are specifications, while A and C are refinements of B and D respectively. B may be viewed as a specification of a component and D as a specification of a context, or vice versa.

The OCS refinement relation has relationships with the refinement relation of CSP (Hoare 1985). The main difference between CSP and the present work, is that CSP describes processes communicating names over channels while the present work describe objects sending messages to other objects. The difference between these two models is discussed in chapter 9 on related work.

Both CSP and the present work define refinement relations based on observable actions. The behaviour of the context is also taken into account in the CSP refinement relation. However, instead of explicitly describing the expected context behaviour, Hoare uses a set of *refusals* to describe what a context should *not* do. The set of refusals is part of a component description.

The two refinement relations differ in that the CSP refinement relation is a binary relation between a component specification and a possible refinement while the present refinement relation is a trinary relation between a component specification, a possible refinement of this specification and a specification of the context's behaviour.

The substitution proposition

OCS designs create systems consisting of several components which all may be substituted. To ensure that similar observable behaviour is maintained when components are substituted and combined the proposition defined below must hold. This proposition is a generalised version of (*) and is called *the substitution proposition*. This proposition defines a monotonicity-like property of a refinement relation which is defined relative to a context:

Definition: The substitution proposition

$$(\forall i \in \{1..n\} : C_i \leq_{D\text{-}i} D_i) \Rightarrow (\forall i \in \{1..n\} : C_i \leq_{C\text{-}i} D_i)$$

where D is the system consisting of the n components $D_1,..,D_n$
C_i is the refinement of D_i and C denotes $C_1||\ldots||C_n$
D-i denotes all D's components except D_i, ie, D-i = $D_1||..||D_{i-1}||D_{i+1}||..||D_n$ and
C-i denotes all C's components except C_i.

The substitution proposition says that

when each components C_i is a refinements of D_i relative to the other components in D,
then C_i is also a refinement of D_i relative to the other components in C.

Such a property is typically interesting when creating a new system where each component is to be a refinement of some component in the specification. When combined, it is then necessary that each of the new components both are refinements of their specifications and that the new component observes that the other new components have similar behaviour to the specification components they are refinements of.

We use the word "reliable" in connection with this property. Reliability is used since it can be associated with the property that a refinement can be expected to have similar observable behaviour in each context with a certain observable behaviour, ie, the refinement has *reliable behaviour*. We define a reliable refinement relation as follows:

Definition: Reliable refinement relations

A *reliable refinement relation* is a refinement relation for which the substitution proposition can be proven.

We also say that when $A \leq_D B$ and the refinement relation is reliable, then A is a *reliable refinement* of B. We define the term *reliable specification* to cover the class of component descriptions for which it is possible to make reliable refinements. If there exists some B and D so that $A \leq_D B$, we say that A is *a reliable component*.

Reliable substitution

When we have reliable substitution, then any number of components can be substituted with their reliable refinements while all the components in the system will observe similar behaviour of their contexts. If we divide D into three sets of components, denoted Dx, Dy and Dz, and similarly we have Cx, Cy and Cz where the indexes of, eg, Dx and Cx are equal so that $C_i \in Cx \Leftrightarrow D_i \in Dx$, then we can express reliable substitution in an even more general form then the general substitution proposition as follows:

Definition: The reliable substitution proposition

$$(\forall\, i \in \{1..n\} : C_i \leq_{D\text{-}i} D_i) \Rightarrow Cx \leq_{CyDz} Dx$$

The proposition says that if Dx, ie, some arbitrary number of components, are substituted with their reliable refinements Cx, then the other components, CyDz, will not observe any difference. The other components may be both "old" components, here denoted Dz, or reliable refinements of old components, here denoted Cy. This is typically the case when maintaining systems, since in such cases just some of the components are replaced with new versions.

The reliable substitution proposition gives a formal definition of *reliable substitution*. In chapter 9 on related work it is shown how the reliable substitution property of refinement relations is linked to composition and decomposition properties of assumption/guarantee specifications.

Strong focus on the objects in the observing context

A central idea in the formulation of the substitution proposition is that similarity of components is defined relative to a context of observers with a given behaviour. The idea is that when two components have observably similar behaviour, then the observing context will not note any difference in the actions they observe. Therefore it is not only the component which is in focus when defining observably similar behaviour. There is an equally strong focus on the observing context and the actions which the objects in the context observe.

In many of the definitions in this thesis the context objects are actually more in focus than the component objects. This is because it is the similarity of the actions the context objects observe which determine if two components have similar observable behaviour. Definition of observability and observable similarity are therefore done relative to a set of observing objects. Therefore the objects in the components play a minor role in these definitions.

Such a strong focus on the observing context objects is not common in most definitions of similarity or monotonic relations between components. In most cases, all objects referred to in these definitions are component objects. This change of focus from component objects to context objects is important to note when reading the rest of this thesis since it might be counter to the readers' expectations.

1.3 Applicability of Reliability Properties

System evolution and change

A system is changed by changing the objects in the system. Depending on the underlying runtime system, changes may in some cases be done directly on the running objects. In other cases, all or part of the system has to be recompiled from new versions of the system code.

One important property of a good system design is that it will simplify the realisation of often occurring kinds of system changes. Examples of often occurring kinds are changes in interest rates and new extensions (PluggIns) to internet WorldWideWeb browsers. Simplifying realisation of changes means minimising the number of objects which have to be changed. A good software designer is therefore a designer who is able to make designs which require few objects to change for common kinds of changes in the requirements and/or the environment of the application.

To make it as simple as possible to manage a system, objects which tend to change at the same time should be grouped into components. The objects should be grouped into components so as to keep the observable behaviour of the components stable, while the objects may change. This is a challenge to software designers.

System changes are often initiated by changes in user needs and in external devices. Such changes require changing the boundary objects which model the users and devices. A goal for a designer is then to create objects which collaborate with the boundary objects in ways which limit the propagation of changes. The ideal is to have a design where a change in a boundary object only leads to changes in the objects within the same component, thus keeping the observable behaviour of the component stable.

One reason why Frameworks and Patterns (Gamma et al. 1994) have growing popularity is because they help find designs which give components with a more stable observable behaviour. This gives more reusable code and programs that are easier to maintain than when the observable behaviour changes.

However, there is a need for methods and tools which can help a developer to verify that one version of a component is a reliable refinement of another version relative to the other components in the collaboration. In practice, there is no verification of reliable refinements. Behaviour properties of OCS components are only checked informally by reading designs and code and by testing a component when it is inserted into a system. To be able to verify that a component is a reliable refinement, it is necessary to have a formal definition of what it means to be a reliable refinement. In the previous section it was defined what it means to be a reliable refinement in rather general terms. Later chapters will give detailed definitions and define the exact properties a component must have in order to be a reliable refinement of a specification. We denote theses properties for *reliability properties*.

When a definition of what it means to be a reliable refinement exists, it can be applied to many aspects of system development and maintenance. A short general discussion of how the reliability properties can be applied is found below. Some examples of how the properties can be applied to reuse and system maintenance are found in chapter 2. Chapter 8 discusses some practical consequences of the reliability properties.

A market for reusable components

Reliability properties can be important in specification and implementation of components in relation to selling and buying software components. When a customer buys a component, it is important to the customer that there are reliable specifications of both the component and the component's context. It is also important to the customer that the implementation of the component is a reliable refinement of the component's specification. If we let ComS and CtxS be the specifications of the component and its context respectively, and let ComI be the component implementation, then it is important to the customer that:

> $ComI \leq_{CtxS} ComS$, ie,
> the implementation is a reliable refinement of the specification relative to the context specification

It is also important that customer knows how to make a reliable refinement of the component's context. This makes it possible to implement a context, CtxI, which is a reliable refinement of the context specification CtxS relative to the component specification ComS, ie, create CtxI such that $CtxI \leq_{ComS} CtxS$.

Reliability will then ensure that when the component and context implementations are combined, they will have observable behaviour as specified and thus function as planned. If reliability is not present in such a situation, the customer might experience unanticipated errors or unanticipated system behaviour.

In practise there is a lack of knowledge on how to make reliable specifications of a component and its context, and how to make a reliable refinement of a specification. Customers of components therefore tend to experience unanticipated behaviour from systems with reused components. One solution which customers feel necessary when errors appear, is to get or buy the source code of the components they use. Then the customer can read the code and find sources of unanticipated behaviour. Some companies have experience so much trouble with bought components that they require source code to be delivered.

New versions of components

When a customer gets a new version of some component, it is important for the customer that the new version is a reliable refinement of the older version. If not, the customer may need to do substantial testing and rewriting to port their code to the new versions. In this case let ComN be the new version of the old component ComI. CtxS is a specification of the context of the components and CtxI the old components context. When CtxI is a reliable refinement of CtxS relative to ComI, ie, CtxI $\leq_{ComI}$ CtxS, and the new component is a reliable refinement of the old component relative to the context specification, ie, ComN $\leq_{CtxS}$ ComI, reliability properties ensure that CtxI $\leq_{ComN}$ CtxS and ComN $\leq_{CtxI}$ ComI, ie, when the new component replaces the old component the system will continue to function as planned.

As in the reuse example, it is important that a component is delivered together with a reliable specification of its collaborators and that the customer knows how to make reliable refinements of the specification. This enables the customer to implement the collaborators of the old version so that they will not change behaviour in unanticipated ways when the old component is replaced by a new version.

It is a problem with many component libraries today that they do not give reliable specifications of components' collaborators. Also, components are changed by library vendors but the new versions are not reliable refinements of older versions of the library. When new versions are not reliable refinements of the older versions, the customer often experience unanticipated behaviour when their existing self made components get new collaborators from the new versions of the library.

Maintenance of extensible systems

Reliability properties are also applicable to maintenance of extensible system. Reliable substitution allows any component in the system to be substituted with a reliable refinement while it is guaranteed that the system will continue to function without creating unanticipated behaviour of existing components. In this case, the components denoted D_i in the general substitution proposition are the existing components, while C_i denote a component which is to replace D_i. We then want the following property:

$$(\forall\, i \in \{1..n\} : C_i \leq_{D\text{-}i} D_i) \Rightarrow C\text{-}x \leq_{Dx} D\text{-}x$$

where D is the system consisting of the n components $D_1,..,D_n$
C_i is the refinement of D_i and C denotes $C_1||\ldots||C_n$
D-i denotes all D's components except D_i, ie, D-i = $D_1||..||D_{i-1}||D_{i+1}||..||D_n$ and
C-i denotes all C's components except C_i.
D_x is some combination $D_i||D_j||..||D_k||D_l$ for $i \neq j \neq k \neq l$ etc. and
D-x is all D except i,j,..,k,l and similar for C-x.

From this we can conclude that when we replace any number of components with their reliable refinements, the objects in the other configurations will not observe any difference in behaviours. The component combination theorem T.6.2 shows properties which ensure that the above property hold for reliable refinements.

Development of large systems

Reliability properties are also applicable when large systems are designed by splitting a design into subdesigns and letting different teams detail each subdesign. If it can be verified that each team has created a reliable refinement of the initial component they were assigned, then reliability ensures that the combined refinements behave as planned in the overall design.

Refer to figure F.1.2 below. The overall design is denoted S. There are three components in S which are named B, D and G. There are three subdesigns named A, C and F where each subdesign is created by a different team. The final step of, eg, the A-team is to verify that A is a refinement of B: A $\leq_{D||G}$ B. Similarly the other teams verify their designs. This gives C $\leq_{B||G}$ D $\wedge$ F $\leq_{D||B}$ G.

The substitution proposition for a system consisting of three components can be formulated as follows:

$$A \leq_{D||G} B \wedge C \leq_{B||G} D \wedge F \leq_{D||B} G \Rightarrow A \leq_{C||F} B \wedge C \leq_{A||F} D \wedge F \leq_{C||A} G$$

If this premiss holds ($A \leq_{D||G} B \wedge C \leq_{B||G} D \wedge F \leq_{D||B} G$), then the collaboration between A, C and F will be similar to the collaboration between B, D and G. This allows separate refinement tests to be performed. When A, C and F are reliable refinements it is ensured that the refinement components collaborate as planned. When separate refinement tests ensure collaboration as planned, there is no need for an extra verification of the behaviour of the complete system composed of A, C and F.

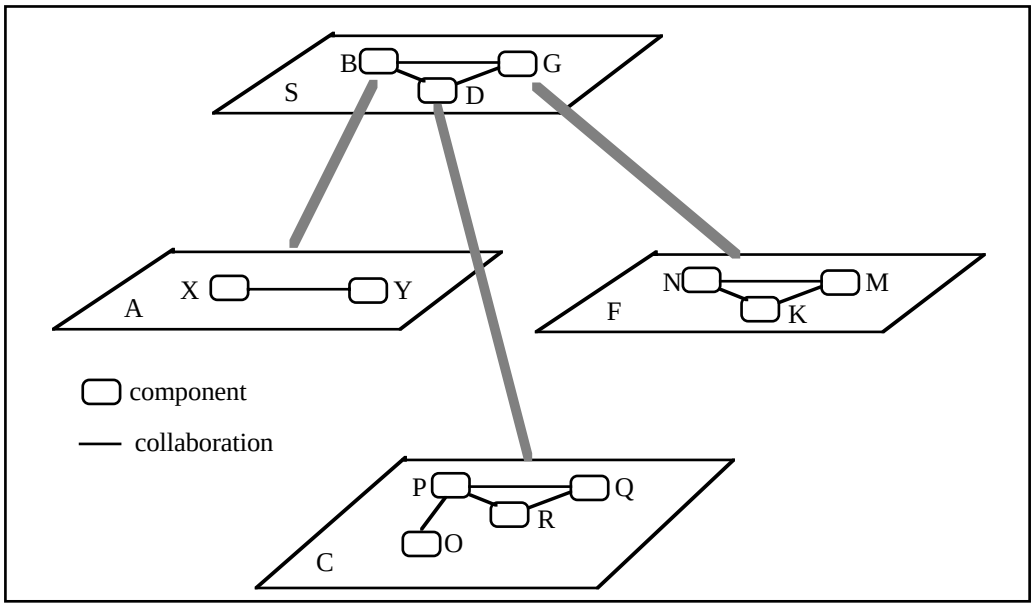


Figure F.1.2 : A hierarchy of designs

Design methods and support tools can be devised to ensure that a subdesign is a reliable refinement of a component in a superdesign. Method and tool development is left for further work.

A design may either be created top down (S first and then A, C and F) or bottom up (A, C and F first and then S). Bottom up designs are often found when reusing libraries and frameworks, while top down is more common when making totally new systems.

A result of a design process may also form a directed graph with multiple roots, since one subdesign may be refinements of components in more than one superdesign. A simplified example of this is illustrated in figure F.1.3 where we have a situation where there are two super designs, with components B1||D1 and B2||D2, and one subdesign, below denoted A. The superdesign B1||D1 can for instance describe the components of a database system where B1 can be the component holding the data and D1 the component doing the storage of this data. The other superdesign B2||D2 can describe the components of an editor where B2 is the component holding the data and D2 the component interacting with the user. A subdesign may be a refinement of both B1 and B2 so that the data it contains can both be stored in the database and edited by the user.

In OOram the designs, above denoted A, Database and User Interface, are described as role models. The round rects in the figure are roles representing objects. The OOram view of this situation is that objects in A have one set of roles in the collaboration with D1 and another set of roles in the collaboration with D2. These roles are described by B1 and B2 respectively. The role models can be combined to describe how the objects in A play different roles in the two superdesigns. Combining role models in this way is called synthesis. This is a very practical way to stitch together systems when more than one Framework is used. In OOram terms, $A \leq_{D1} B1$ and $A \leq_{D2} B2$ would ensure *safe synthesis* of the role models. Safe synthesis is introduced in the book (Reenskaug et al. 1995) where it says:

> *"Synthesis is called safe when the static and dynamic correctness of the base models is retained in the derived model, and unsafe if we retain only the static correctness and have to study the derived model to determine its dynamic correctness."*

The authors of the OOram book emphasises the importance of safe synthesis and give some guidelines on how to ensure it. However, a complete formal proof of safe synthesis do not exist. Applying the reliability

requirements to OOram could be one way of formally proving safe synthesis. A formal definition of safe synthesis is left for further work.

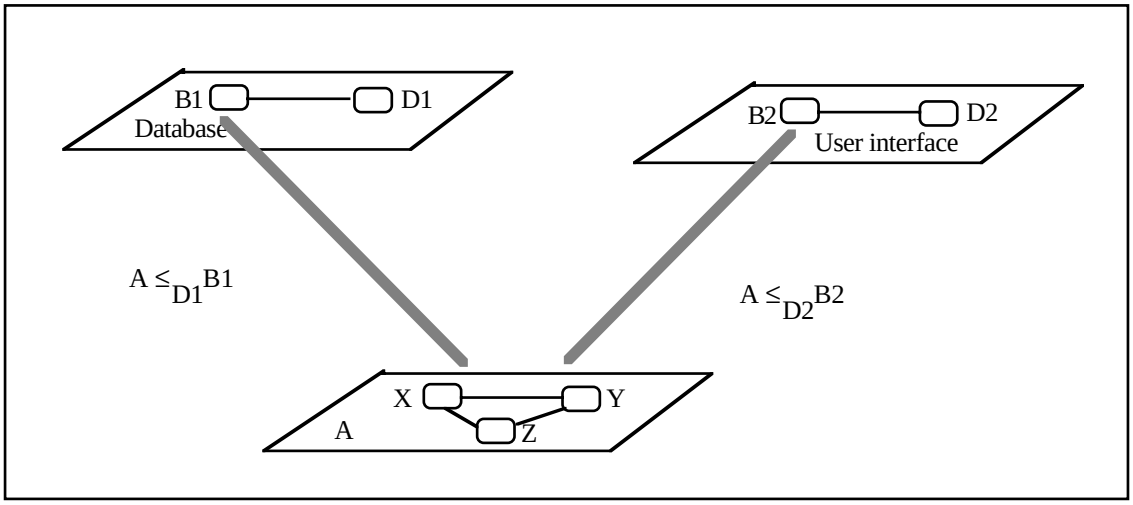


Figure F.1.3 : One component being the refinement of two components

As the above examples show, there are many application areas for reliability properties. It would be very useful to have methods and tools which give system developers maximum benefits from the reliability properties presented in this thesis. However, a significant amount of work must be done before such methods and tools exist. The hope is that the presented formalism and results can form a basis for creating such methods and tools through the deeper understanding it gives of the pit falls and possibilities related to the design and creation of object component systems.

1.4 Goal and Document Structure

The goal of the thesis

The main goal of this thesis is to give a definition of "similar components" which is in accordance with the OCS design tradition. Also, similar components should be defined in a way which ensures that a component can be substituted with similar components without the component's context notifying any difference. This property was expressed in the substitutability proposition presented in section 1.2.

In order to reach this goal, several steps have to be taken to formally define, express and show various properties of object component systems. The different steps are described below with reference to the chapters which present the results.

Defining key concepts in object component design

There are many definitions of object-oriented programming and modelling, see eg (Blair et al. 1991) for an overview. Some have common concepts with different names, while others use the same name for different concepts. The first step in reaching the main goal was therefore to find a consistent set of concepts which is used by most communities successful in designing reusable object-oriented components, and to eliminate concepts which are not used by these developers.

It was of particular interest to formalise the component developers' definition of similarity between components, ie, what properties are considered in order to establish that a component may substitute some other component. This is a necessary basis for defining a refinement relation in accordance with the component developers' tradition.

Many existing OCS designs were studied in order to find the concepts and definitions which are in accordance with OCS design tradition. Chapter 2 presents a version of a widely used and simple example of component specifications, namely a version of the Model-View-Controller design.

A search for applicable formalisms

There are no good reasons to create new formalisms if existing formalisms have sufficient reasoning power, and when results of such reasoning can intuitively be transferred to practice. Therefore, most existing formalisms were studied in search of an appropriate formalisation of OCS concepts.

The study of existing formalisms included models of distributed systems, functional models, trace based models, temporal logic based formalisations and various other approaches to describe objects. In order to use existing formalisms to reason about components, a translation from OCS concepts to the concepts of the relevant formalisms is necessary. Some attempts were made at translating from OCS concepts to the most similar formalisms, see, eg, (Nordhagen 1992). In the translation process concepts were changed to the unrecognisable. It was therefore hard to translate the results of the reasoning back to the realm of object components, similarity definitions and reliable substitution. The conclusion was that none of the existing formalisms modelled OCS concepts directly and/or did not have sufficient reasoning powers to show properties as expressed in the substitution proposition.

The results of this search for existing formalisms is summed up in chapter 9 on related work. The chapter is found after the chapters describing the concepts used in the thesis. This is done in order to be able to compare the concepts defined in this thesis with the concepts found in other related works.

A computational framework for OCS components

OCS developers are used to think, model and express their designs using object-oriented programming languages or some graphical notation based on such languages. Specifications of the designed OCS components and the reasoning about them are done informally since object-oriented design method notations and object-oriented programming languages are not supported by formal reasoning tools.

A language used for reasoning about OCS components must be able to express the designs so that component developers will trust that the object descriptions and proofs actually correspond to his or her intentions. A specification should therefore be intuitive to the developer. The formalism must therefore be based on concepts familiar to component developers.

Component developers tend to make operational specifications of components using design notations and object-oriented programming languages. Therefore the specification language should resemble an object-oriented programming language, but allowing reasoning.

Chapter 3 introduces a new language created for reasoning about OCS components. The language is called Omicron after the Greek letter "o". Omicron is a minimal language with properties and semantics both corresponding to the OCS design concepts and suitable for formal reasoning . The Omicron language allows components to be specified operationally and therefore corresponds with the operational specification tradition of component designers. When the specifications are expressed operationally, there is no clear distinction between specification and implementation of components. Both are configurations of objects.

The Omicron language is not particularly user friendly since the programmer has to describe, and manually control, details which are usually handled by an underlying runtime system. This has been done since the Omicron language is defined primarily for being a formal basis for reasoning about similarity of components. Priority has therefore been given to making a language with simple formal semantics. However, the language makes it possible to describe objects in a manner which is familiar to component developers.

The Omicron calculus

The first sections of chapter 3 leave out details in the formal definition of Omicron, since these parts of the chapter focus on explaining and motivating definitions and should be readable for people without too much experience in theoretic computer science. However, to create a calculus for reasoning about object components, the language must be formally defined. This is done in section 3.4.

Section 3.4 first gives the syntax of Omicron through the use of extended BNF. Next, the Omicron language's semantics is defined through the Omicron calculus. The calculus gives an operational definition of the Omicron language's semantics along the lines of Plotkin's work (Plotkin 1981).

Definition of similar components

Chapter 4 gives a formal definition of "similar components" as perceived by component developers. The definition of similarity is explained and motivated by referring to the examples in chapter 2. Component similarity is defined by a refinement relation which holds when a component is "similar" to another component.

Finding reliability requirements

As mentioned, the definition of similar components should have the properties expressed in the substitutability proposition presented in section 1.2. Chapter 5 shows that it is not possible to show the substitution proposition for the refinement relation of chapter 4, a relation defined based on component developers view of similar components. Because the proposition can not be shown for this relation, there was obviously a lack in component designers' understanding of what must be specified to get reliable substitution when combining "similar" components. This created an additional goal:

> To define a refinement relation which makes it possible to prove the substitution proposition and thereby ensure reliable substitution in component systems.

To get reliability it is necessary to introduce requirements on expressions defining object component and implementations and to strengthen the definition of refinement. The requirements on object component expressions are denoted *reliability requirements*.

Chapter 5 introduces a set of reliability requirements and gives the definition of a reliable refinement relation.

The necessity of the reliability requirements

There are different alternative reliability requirements. The alternative requirements can set requirements either on the Omicron configuration expressions or on relations involving actions or on both. As concluded in chapter 2, the preferred alternatives are those which avoid setting requirements on configuration expressions since this will restrict how implementations and specifications of behaviours are done. This means that it is preferred to set requirements on relations involving actions. However, it is shown that to get reliability, there is no way to avoid some requirements to be set on configuration expressions. The necessity of such requirements is shown by examples in chapter 5. This chapter should be readable without too much training in formal reasoning. It is more important to have a good intuition of object-oriented concepts.

The reliability requirements are sufficient

The sufficiency of the reliability requirements is shown by theorem 6.3 in chapter 6. This theorem shows the substitution proposition presented in chapter 1 for the reliable refinement relation defined in chapter 5. The proof of the theorem builds on a number of propositions, lemmas and theorems and is not easy reading. In the hope of helping the reader, each proposition, lemma and theorem is accompanied by an intuitive explanation.

Not taking reliability requirements for granted

It is interesting to find *only* the necessary reliability requirements, and not take traditional ideas related to reliability requirements for granted. As noted above, the traditional belief is that a reliable refinement of a specification must be of the same type or a subtype of the specification, while this is actually not true as shown in chapter 9.

In order to find only necessary reliability requirements it is important that no reliability requirements are taken for granted. It is therefore important to avoid traditional object-oriented typing and other such ideas which are introduced to help developers write correct code since such ideas have strong relations to reliability requirement assumptions.

One reason Omicron is not very user friendly is because of this lack of language support for creating reliable components. Using the reliability requirements as a basis for defining a language can give a much more user friendly language. However, creating a language which supports the creation of reliable components is a huge task and is therefore left for further study.

Parallel and sequential object component systems

When formalising object component systems, the model will have different properties depending on weather the system allows object to execute in parallel or only allows sequential execution of objects.

In a sequential object component system, objects are passive, in that they are only activated when they receive a message. When the receiver of a message is active, the sender waits for a reply and is passive. Passive objects are common in systems created from, eg, C++ and Smalltalk programs. However, a certain degree of parallelism has been present in most object systems. Simula has co-routines allowing objects to executed in quasi parallel. Smalltalk also have quasi parallel execution of objects. Java includes *treads* which makes it possible to create objects which execute in parallel. Distributed systems are also implemented using OCS design, and such objects are truly executing in parallel.

When developing the formal model of object component systems, it became evident that a language for describing objects executing in parallel was simpler than a language for describing a sequential system. The parallel language is therefore the main language of this thesis. However, a sequential version of the language has also been made. This version is presented in chapter 7.

When making formal frameworks for reasoning about object component systems, one run into many of the same problems encountered when making frameworks for parallel systems. This might be caused by object systems having graph-like collaboration structures. This causes passive objects to get messages while waiting for a reply. This creates similar complexities and instability problems as experienced when objects execute in parallel. It might therefore be expected that the reliability requirements for both parallel and sequential systems will be quite similar. The similarity is shown in chapter 7. The chapter ends with the conclusion that the reliability requirements are, for all practical uses, equal for parallel and sequential object component systems.

The reliability requirements correspond to component development practice

When the reliability requirements are defined, it is possible to see if the formalisation of the component developers concepts has been correct in the following sense:

> If the reliability requirements correspond to what is considered *good practice* among experienced object component system designers then the formalisation of object-oriented concepts has been done in accordance to the tradition of these designers.

Examples of "good practice" can be found in existing libraries of reusable components and to some extent in the design of object-oriented programming languages. "Good practice" have also been published as rules of thumb (Johnson and Foot 1988), or in other informal forms, for instance Patterns (Gamma et al. 1994).

Chapter 8 presents such correspondence between practical consequences of the theoretically founded reliability requirements and the practitioners' view of good design practice. The correspondence is shown by translating the theoretically expressed reliability requirements of chapter 5 to practical advice for making reliable refinements and reliable specifications. These advice are then compared to the "good practises". The conclusion is that there are many similarities, while there are also some reliability requirements which are not covered by "good practises". This is in line with the findings that the component designers definition of "similar components" do not have the properties expressed by the substitution proposition.

Object-oriented programming languages and design notations include some features which are supposed to give reliability properties. A typical such feature is compile time typing. Chapter 8 shows that these features are far

from sufficient to get reliability. For example it is shown that a reliable refinement often is of a supertype of the specification. This is counterintuitive to the type and subtype idea.

New advice for component designers

As mentioned, there are some reliability requirements which are not covered by "good practises". The last section of chapter 8 gives a summary of the lessons learned from the reliability requirements. It sums up how to make reliable refinements and reliable specifications. The most important new lesson is that a reliable specification must include descriptions of objects, not just types of objects. A reliable specification must define the maximum number of objects one component sees from its context. The reliable specification must also specify which messages are used to send references to each of the identified objects. A reliable refinement of the context can have no more objects seen by the component, but may have fewer. Details of this is given in section 8.

Conclusions

Chapter 10 presents the conclusions drawn from the formalisation of object component system designs and the results of finding a definition of a reliable refinement relation. The work on finding reliability requirements has also given some ideas on how to develop the Omicron framework to cover a wider range of concepts and models of object components. These ideas are also given in chapter 10.

A long term goal: Better support for component development and exploitation

A long term goal is to design development tools and processes which better support the design of substitutable components and will help in utilising such components in system development and maintenance. The definition of a reliable refinement relation is a step on the road to meeting this long term goal. The reliability requirements apply to a whole range of topics related to software development, eg, component and system design, programming language design, system development methods, component specification and testing and software development processes and organisations. Such ideas are elaborated on in chapters 8 and 10.

The long term goal is ambitious and there is major and/or mature previous work in the gap between practical OCS design and formal theories. The presented work is therefore a small contribution in bridging this gap and reaching the long term goal.

Document style

In the following chapters there are many references to object-oriented programming languages and analysis and design methods. These references are made by using the name of the languages and methods, while references to literature about the languages and methods are given in appendix D. The languages are Beta, C++, CLOS, Dylan, Eiffel, Java, SELF, Simula and Smalltalk. The methods are BON, Foundation, Fusion, Objectory, OMT, OOram, RDD, Catalysis, Syntropy and UML. Some Design Patterns are also referenced by name and literature references to these patterns are found in appendix D.

Chapters 3 to 6 show some properties of object component systems. The most important properties are given through *theorems* while less important properties are given as *propositions*. Interesting properties which do not need to be shown through formal proofs are stated as *observations*.

1.4 The Contributions of this Thesis

Reliable substitution

The main contribution of the presented work is the idea expressed in the substitution proposition for a dynamic language. This idea is related to the characteristics of object component system designs where components are specified by their observable behaviours relative to a context and where all components may be substituted with components with similar observable behaviour. No formal framework has previously taken these characteristics as a starting point, and therefore no formal definitions of these ideas have previously existed.

The reliability requirements

Formalists often focus on typing and functional aspects of objects. As is shown in the chapter on related work (chapter 9), this is very different from the kind of properties focused on in the definition of "similar components" presented in chapter 4 and the reliability requirements presented in chapter 5. As shown in chapter 8, just one of many reliability requirements is slightly related to traditional typing of objects. However, the traditions focusing on giving informal advice on how to make reusable components have many ideas which are similar to the reliability requirements. The correspondence between the reliability requirements and the informal rules for making reusable components, shown in chapter 8, is an indication that the Omicron framework formalises some important aspects of object component systems.

Previous formal works have not presented results which may correspond to reliability requirements in line with state of the art design practice. This indicates that the presented work is more in line with OCS design concepts than previous formalisations.

Identification of component developers' key concepts

In theoretical computer science there are a few fundamental models of computation. The three most widely studied are:

- the logic model based on predicate calculus
- the functional model based on the lambda calculus and
- the process model formalised through for example the π-calculus

Identification and formalisation of the key concepts used by OCS component developers and comparing them to concepts in other models of computation, contribute to a better understanding of the differences, weaknesses and strengths of the various concepts. This in turn may lead to more fruitful discussions and integration of the various approaches, without missing out important concepts from the component developers as has been the case for several previous attempts. The OCS concepts are presented in chapter 2 and their relation to other models of computation is discussed in chapter 9.

The Omicron computational framework

The most generally applicable result of the presented work is the Omicron computational framework itself. The definition of a reliable substitution is an important example of results which may be derived with calculations with the Omicron framework. This may even be applied to many other aspects and problem areas related to object-oriented programming, systems, languages and applications. Some examples are given in the last chapter where there is a section on further work.

Abstract specifications vs. irrelevant details

Hoare in (Dahl et al. 1972) describes abstraction as follows:

> *In the development of our understanding of complex phenomena, the most powerful tool available to the human intellect is abstraction. Abstraction arises from a recognition of similarities between certain objects, situations, or processes in the real world, and the decision to concentrate on these similarities, and to ignore for the time being the differences. As soon as we have discovered which similarities are relevant to the prediction and control of future events, we will tend to regard the similarities as fundamental and the differences as trivial.*

In these terms we can say that abstract specifications leave out all irrelevant details. It is not obvious, however, which details in a component's behaviour are relevant and would be included in the specification of an OCS design and which are not and should be left out. This is one of the topics which may be studied further once a computational framework like Omicron exists.

It can be argued that the relevant behaviour consists of the properties which have to be specified to ensure reliable substitution. The reliable refinement relation presented in chapter 5 gives in this view a definition of

relevant behaviour. The definition of relevant behaviour can lead to a better understanding of what abstract and reliable specifications are. This means that the reliable refinement relation and the reliability requirements can define limits on abstraction, ie, what must be included and can not be left out in order to ensure reliable substitution. Such limits have not been formally studied before in relation to OCS components.

Differences in the typing of components and of the elements within a component

The presented work shows that there are lessons to be learned about how to type substitutable components as opposed to how to type elements used within a component. More complex relations may be acceptable within a component than between generally substitutable components. Previous work on typing and categorising objects have not made such distinction between typing modelling elements within a component and typing substitutable components.

It has been known for a long time that using encapsulation gives more maintainable code, but at the same time encapsulation can creates problems with fragmentation of an algorithm. Fragmentation makes more complex code and creates the feeling that "everything happens somewhere else" when reading such code. These problems with encapsulation exemplifies the trade-off between simple/complex specification, reusability and the distribution of code and data in the objects. The presented work gives a basis for a more well-founded understanding of such relationships and trade-offs.

Possible results from applying the reliability requirements

The explicit and formal definitions of the reliability requirements can be used to make better system development methods and more extensible systems. The reliability requirements apply to component and system design, programming language design and system development methods. The requirements may be used to create development tools and processes which better support the design of substitutable components and help in utilising such components in system development and maintenance. This in turn can hopefully lead to more efficient development processes, easier maintainable systems and less errors in users' systems.

CHAPTER 2

Model-View-Controller

-

The Classical Example

Section 1.1 briefly listed the characteristics of OCS models which distinguish them from other models, such as communicating processes and functional models. This chapter will exemplify and present these characteristics in more detail.

The chapter starts by presenting an existing OCS design. This is done in order to illustrate the OCS characteristics. The example is the classical example of object-oriented design, the Model-View-Controller Framework (MVC).

The presentation of the example is done in two steps. First the general design and its implementations are discussed in section 2.1. In section 2.2 an example of a specific version of the MVC design is presented. This is the model-view contract which was first published in (Helm et al. 1990).

Observable behaviour was loosely described in section 1.1 under characteristics of object component systems. Section 2.3 describes observable behaviour in detail and uses the MVC Framework to exemplify the descriptions. Similarly, section 1.1 mentioned observably similar behaviour and section 2.4 gives a description of it. Sections 2.3 and 2.4 give informal descriptions while chapter 4 gives the formal definitions of observable behaviour and observably similar behaviour.

Section 2.5 gives two simple and informal examples of reliability requirements and motivates them by giving examples of problems related to reliability in MVC designs and implementations.

After having read this section, the reader will hopefully have an intuitive understanding of object component systems, including those aspects of these systems which are central when defining similarity of components and reliable substitution. The reader should also have understood what it means for one component to have observably behaviour similar to some other component. Furthermore, the reader should have understood that in order to ensure reliable substitution of OCS components it is necessary to do a rigorous study of the similarity of components' observable behaviour relative to a set of observing objects.

2.1 The Model-View-Controller Framework

This section presents a version of the classical example of object-oriented design, the Model-View-Controller Framework (MVC). Various versions of MVC is documented in (Krasner and Pope 1988), (Deutsch 1989) and (Goldberg 1990). MVC is also implemented in different programming languages and is used in the literature to exemplify aspects of object behaviour which is not easily specifyable by existing formal specification methods. This is pointed out in, eg, (Helm et al. 1990), (Holland 1992) and (Eliëns 1994).

After the presentation of the MVC example, the characteristics of object component systems are illustrated by referring to the example. It is expected that the reader is familiar with concepts such as object, instance variable, method and message as defined in (Blair et al. 1991) and used in relation to most object-oriented programming languages.

Overview of the components in MVC

The main idea of MVC is to divide an application's tasks into three parts or components. Each component is assigned some of the functionality the application is to have. Also, each of the application's functions is assigned to one of the three components. The three components and their delegated task are:

The model component is assigned the task of holding the data.
The view component is assigned the task of displaying at all times a correct presentation of the data
The controller component handles user input and initiates actions which will change the held data and update the displayed presentation.

The MVC interface Framework describe how to structure and define the collaboration between the components in order to get a well functioning system. The collaboration is concerned with updating data and notifying the view component about the changes so that the displayed presentation gets updated along with the data.

Typically there will be a set of data which is held in a model and where the data is presented to the user in several ways simultaneously. In such a case there will be several views and controllers which interact with the same model component. Figure F.1 illustrates such a situation with one model and three view/controller pairs. The lines indicate that the connected components collaborate.

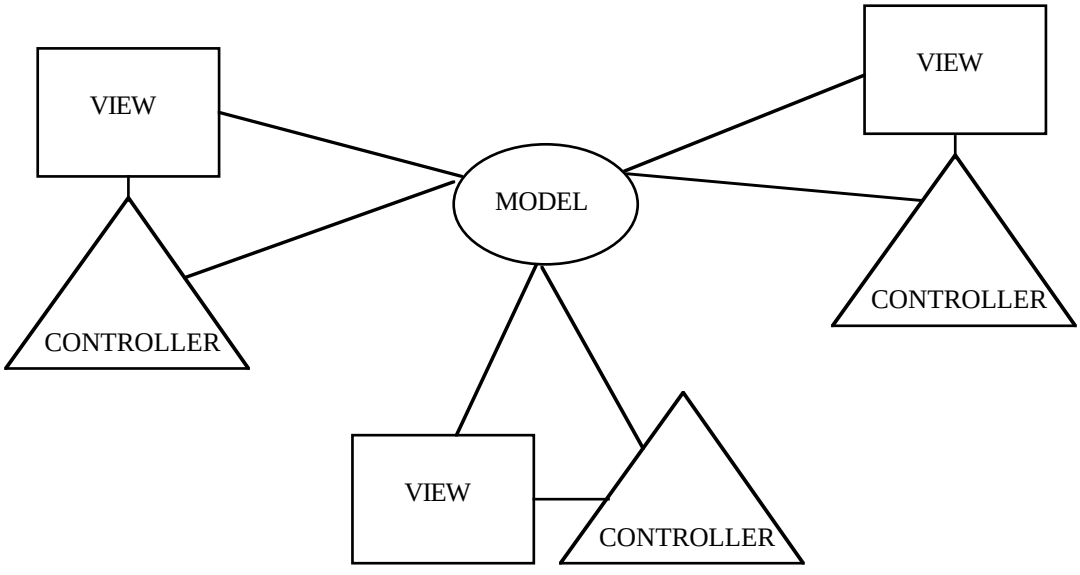


Figure F.1
One model component is collaborating with three view components and three controller components. The rectangle, triangle and oval represent the components and the lines show how the components collaborate.

The design is done so that the view/controller pairs do not know of each other. Instead, the presentations in the views are co-ordinated through functionality in the model. The focus of the MVC design is on how this co-ordination is done. The co-ordination functionality is intended to ensure that all views are updated when the

model is changed by actions originating in one of the controllers. Keeping the view/controllers ignorant of each others existence makes a system more flexible in that it simplifies the task of adding and removing view/controllers. Section 2.2 below presents how the views are co-ordinated by the model in the classical way; the way this has been implemented in versions of Smalltalk since 1980 and in other programming languages in later years.

Design for combining and substituting components

The functionality of the system is divided into three components since this has simplified the maintenance and reuse of the programs making up the system. Each component is implemented by a set of objects and each component may be substituted with a component which collaborates with the other components in a similar way. Similar collaboration is described and exemplified in a section further below.

To change applications and/or reuse components, new components may be added and/or components may be substituted or changed. For instance, a view/controller pair may be added if the model is to be viewed in a new way. If a user of an application wants the presentation of some data to change, a good design would allow such a change to only affect the view component. Also the controller may be substituted or changed. This happens when the user is to get new commands, or the number of commands should be restricted. Typically, there is a controller for each view which handles the full set of user commands and then there is a common default controller which does not allow the user to give commands. This last controller (in some versions of Smalltalk called NoController) is used when the user is only allowed to view a presentation of the model. New systems can be created by combining the different versions of controllers with the same model and combining the model with different view/controller pairs. It is therefore possible to create quite different systems just by combining or replacing components.

Replacing a component can be done when components have similar observable behaviour. Combining components can be done when each component has *the observable behaviour expected by the implementors of the other components*.

Components form dynamic graph-like collaboration structures:

The MVC design gives an example of objects which form a simple graph-like collaboration structure. The graph is formed by the collaboration structure shown by the lines in figure 1. The collaboration structure may also change during the components existence as, eg, new view/controller pairs are added or one of the controllers is replaced with a new controller.

The model component is both a client and a server of the other objects in the system. It is a server since the other objects sends it messages. Messages sent to the model component are for example messages to tell the model about new views. It is a client since it sends messages to objects in its context. Such messages are typically messages to the views notifying them about a change in the data stored in the model. Components with such a double role as both a client and a server is often found in object component designs.

Single components are seldom designed in total isolation:

As previously mentioned, one important property of a good system design is that it will simplify the realisation of often occurring kinds of system changes. A good advice which will simplify system changes is to group objects which tend to change at the same time into components. One aim is then to keep the observable behaviour of the components stable, while the objects may change. One reason why MVC is popular is because this is a design which leads to components with a more stable observable behaviour.

To see and judge how design decisions for the different components will influence this important property of the design, it is necessary to see how the components function together. For instance it must be possible to see if a probable change in a view will affect the observable behaviour of the view relative to the other two components. Because of such considerations when designing components, single components are seldom designed in isolation.

Context dependent specifications:

The MVC design describe how the three components collaborate with each other. It does not specify how each component will collaborate with components with observable behaviour different from the two other components.

The MVC components are active and take initiative and send messages to the other components. Components collaborating with active components might not function properly if their partners do not take the initiative they are expected to take. For instance, a view may not present a correct picture of the model to the user unless the model component takes the initiative and notifies the view when the data in the model changes. The views functionality is therefore dependent on the observable behaviour of components in its context, in this example the model component.

Symmetry of component and context

The MVC design is an example of a system design where all parts of the system may be viewed as components. Also, each component forms the context of the other components. Depending on point of view, a part of the system may either be viewed as a component or as a context. The view point is determined by what part of the system is being manipulated. If, for instance, a controller is to be replaced, the model and view components are viewed as the context. On the other hand, if a new view/controller pair is added to a system, the new view/controller pair is the component, and the model and other existing views and controllers make up the context. Through its existence, a controller may then be viewed both as a component and as part of a context.

Separate development of new versions of component and context

Components can be developed separately in space or time. Separation in space is typically when different parts of a system is developed in parallel. Separation in time is typically when a new component is developed and shall replace some existing component which has been used in different systems.

We first illustrate how the refinement relation applies to separate development of components in time. To illustrate this we use the Smalltalk text editor as a reference. The text editor (the classes TextView and TextController) is a much used component is the Smalltalk system. A huge number of models have been created which interacts with the text editor. All models have similar observable behaviour relative to the text editor. The Smalltalk vendors' implementations of the text editor have differed substantially in the different versions of the Smalltalk-80 (later ParcPlace Smalltak) from 1980 until present. Even with these variations, the observable behaviour of the (most reused version of the) TextView/Controller as observed by the model has remained the same since 1980. Similarly, the observable behaviour of the model relative to the TextView/Controller has remained the same for the same number of years. It is very common to make new versions of the model while it is not very common to replace the Smalltalk text editor by non-vendors. However, it this has been done, see, eg, (Nordhagen 1987) and (Hohan 1988).

The creation of the new version of the Smalltalk text editor documented in (Nordhagen 1987) is one of the practical experiences on which the theoretical work in this thesis builds. The new version of the editor totally replaced the existing version which was delivered with and heavily used in the standard Smalltalk 2.5 system.

Below, reliable substitution is illustrated by giving an example of how various components in a Smalltak system can be replaced.

In all Smalltalk systems there are different versions of model components collaborating with the text editor. Assume that ParcPlace delivers, or you create, a new version of the Smalltalk text editor called NewTextEditor. It must then be ensured that all existing systems will continue to function when the NewTextEditor replaces the old text editor. There are two alternative ways of Ensuring this. One alternative is to ensure this, is to let the developers check that each model in every Smalltalk system collaborates with NewTextEditor as planned. This is rather impractical since it would require a substantial effort. The other alternative is to have a reliable specification of the model and text editor collaboration and then make reliable refinements of the components. This would ensure reliable substitution of components, meaning that the new text editor can replace the old version without creating unanticipated side effects.

The model-editor collaboration is informally described in the Smalltalk documentation. There have been attempts at formally defining the model component's behaviour, see, eg, the example in the next section. However, there have been little or no work in defining appropriate refinement relations and monotonicity properties of such relations so that reliable substitution is ensured.

To ensure reliable substitution it is necessary to show that NewTextEditor is a *reliable refinement* of the text editor. It must also be shown that existing text models are *reliable refinements* of some ideal text model. This means that every text model must have similar observable behaviour to the specified ideal text model. The observers are the objects in the text editor component. If we call the specified ideal text model for TextModel, the existing text editor for TextEditor and an arbitrary text model for MyModel, we must then have:

MyModel has similar observable behaviour to TextModel as observed by the TextEditor

The new text editor NewTextEditor, must have similar observable behaviour to the old text editor as observed by TextModel. This means that we have the following:

NewTextEditor has similar observable behaviour to TextEditor as observed by the TextModel

To get a correct functioning system when installing the NewTextEditor into an existing Smalltalk system and combining it with MyModel, the following must hold:

NewTextEditor has similar observable behaviour to TextEditor as observed by MyModel and
MyModel has similar observable behaviour to TextModel as observed by the NewTextEditor

The different components can be pictured as in figure F.2 below.

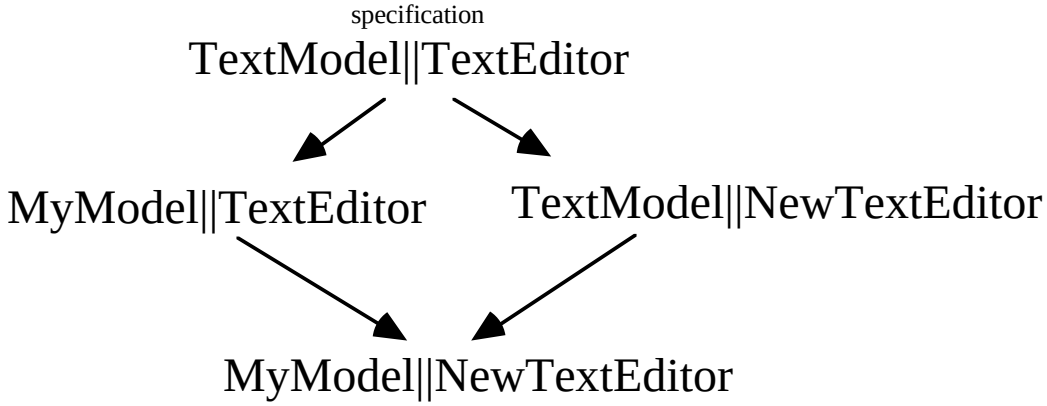


Figure F.2
Illustration of how the different text models and text editors are combined.
|| is used to indicate that components are combined into a system.

The above statements about similar observable behaviour can be restated using the refinement relation notation as follows:

$$\text{NewTextEditor} \leq_{\text{TextModel}} \text{TextEditor} \wedge \text{MyModel} \leq_{\text{TextEditor}} \text{TextModel}$$
$$\Rightarrow$$
$$\text{NewTextEditor} \leq_{\text{MyModel}} \text{TextEditor} \wedge \text{MyModel} \leq_{\text{NewTextEditor}} \text{TextModel}$$

This is the substitution proposition expressed for the TextModel and TextEditor components. Since both the TextModel and the TextEditor can be substituted there is no distinctions between components - which may be substituted, and context - which stay unchanged.

In what follows, the term *configuration* is sometimes used when referring to a set of objects in a system. This is to avoid referring to the set of objects as either component or context. Configuration is therefore used when the set of objects may be viewed both as component and as context. For instance in relation to the above substitution proposition, the TextModel is viewed as both component and context. A TextModel may therefore also be referred to as a configuration, eg, the TextModel configuration.

Reuse, maintenance and reliability properties

The substitution proposition can be applied to practical problems such as reuse of components and maintenance of systems. Some examples are given below.

Reuse
In relation to reuse, the substitution proposition says that a component which is to be reused must have a reliable specification, and the implementation must be a reliable refinement of the specification. If not, the implementation might not function as planned when the component is collaborating with some new context.

To illustrate this we let the TESpec be a reliable specification of the TextEditor and TextModel be a reliable specification of the models behaviour. Then TextEditor must be a reliable specification of TESpec, ie,

$$\text{TextEditor} \leq_{\text{TextModel}} \text{TESpec}.$$

The context with which the reused component is to collaborate, need only be a refinement of the context specification since the reused component is not substituted by another component. In this case the context is implementations of TextModel. Therefore, for the collaboration between the reused TextEditor component and new model components to function as planned, it is necessary that all new models are refinements of TextModel. They need not be reliable refinements, since TextEditor is not replaced by another component. However, if it is a possibility that TextEditor will be replaced, then new models must be reliable refinements. This is discussed below under maintenance.

Maintenance by replacing components with new versions
In relation to maintenance, the substitution proposition says that a component which is to replace some old component must be a reliable refinement of the old component relative to the rest of the system under maintenance. To be able to create a reliable refinement, there must be a reliable specification, and the rest of the system under maintenance must consist of components which are reliable refinements of their respective specifications.

The reason for making a new version of the Smalltalk text editor was that the implementation had some internal errors and the current implementation was difficult to understand. The errors were therefore difficult to correct and also, the implementation was difficult to expand with new functionality. It was therefore decided to replace the old implementation with a new one.

In this case, there were simple and informal specifications of the collaboration between a text editor and a text model. The observable behaviour of the components were quite simple, and thus the implementations of all the text models in the system could be expected to be reliable refinements of the specification. Therefore, installing a new version of text editor into the Smalltalk system created few or no problems in getting the models to collaborate correctly with the new text editor.

2.2 A Simple Model-View Contract

Many notations have been developed for describing object behaviour in a formal way. Most of these notations are developed from notations used for describing functional aspects of systems. They therefore focus on how values are handled when a single object receives messages. Many such approaches are referred to in chapter 9 on related work.

The observable behaviour of the model component in the MVC Framework can not be directly specified using most of these approaches. This is because most of the other notations do not cover object behaviour which include sending messages as a result of receiving messages while such message sending is the key feature of the model's observable behaviour. This key feature is sending a special message to a dynamically changeable set of objects. Chapter 9 gives more detail on why this kind of behaviour can not be directly specified using algebraic specifications and process models.

One approach which takes formalisation of message sending more seriously is that of the group of people working on what they call *contracts*, see, eg, (Helm et al. 1990). The work on contracts is often referred to in the literature, but since the publishing of (Holland 1992), no significant developments have been reported. The group has changed its focus. However, the paper (Helm et al. 1990) presents a version of MVC which specify the key feature of the behaviour of the components in the MVC Framework. In their example the controller is not explicitly included. This is done to simplify the example. The contract in the paper (Helm et al. 1990) is called the model-view contract and is presented below.

Operational specifications

The contract notation makes it possible to specify object collaboration by using a semiformal notation. A contract lists the objects that participate in the task and characterises dependencies and constraints imposed on their collaboration. The contract is *an operational specification* of a model's observable behaviour relative to a context consisting of zero or more view objects and maybe also some other objects (the model's context is discussed below). A slightly modified version of model-view contract using a simplified notation used in (Eliëns 1994) pages 348 - 149 is shown below. First the contract is given and then the notation is explained.

contract model-view (V) {
subject : model *supports* [
state : V;
setvalue(val : V) → [state = val]; notify();
getvalue() → *return* state;
notify() → ∀ v ∈ views • v. update ();
attach(v : view) → v ∈ views;
detach(v : view) → v ∉ views;
]
views : set<view> *where* view *supports* [
update() → draw();
draw() → subject.getvalue; [view *reflects* subject.state];
subject(m : model) → subject = m;
]
invariant:
∀ v ∈ views • [v *reflects* subject.state]
Instantiation:
∀ v ∈ views • subject.attach(v) & v.subject(subject);
}

To indicate the type of variables, the notation *v : type* is used expressing that variable *v* is of type *type*. The *subject* of type *model* has an instance variable *state* of type *V* that represents (in an abstract fashion) the value of the *model* object. Methods are defined using the notation

method(…) → action

Actions may consist either of other method calls or conditions that are considered to be satisfied after calling the method. Quantifications as for example in

∀ v ∈ views • v. update ();

is used to express that the method *update()* is to be called for all elements in *views*.

The actual protocol of interaction between a *model* and its *view* objects is quite straightforward. Each *view* object may be considered as a handler that must minimally have a method to install a model and a method *update* which is invoked, as the result of the *model* object calling *notify()*, whenever the information contained in the model changes. The effect of calling *notify()* is abstractly characterized as a universal quantification over the collection of *view* objects. Calling *notify()* of *subject* results in calling *update()* for each *view*. The meaning of *update()* is abstractly represented as

update() → [view *reflects* subject.state];

which tells us that the *state* of the *subject* is adequately reflected by the *view* object.
The invariant clause of the *model-view* contract states that every change of the (state of the) *model* will be reflected by each *view*. The instantiation clause describes, in a rather operational way, how to initialize each object participating in the contract.

An object may be a component, but a component may consist of several objects

Since the number of pages in a paper is limited, design examples are often simplified to the extent that components are reduced to single objects. Also, most papers do not focus on the design as such. Instead the focus is on some new notation or language where the design is used as an illustration. This is also the case with the above example. However, in the contract example above and other examples in papers about contracts, component and object are synonyms. It is not just a simplification. Components and objects as synonyms is also reflected in the naming used in the contracts and by the reference to a single class as implementation of an abstract counterpart in a contract. In practise, eg, in Smalltalk, the model component is not just a single object, but several objects. For instance, it is usually not the model object itself which holds the set of views, but rather an object of a Collection class. The collection object is viewed as part of the model component.

There are some published design examples which are more extensive and give examples of components consisting of more than one object. For example, the article (Jacobson et al. 1995), and other work by the same authors, show many examples of OCS designs. These examples are not related to Smalltalk, as the MVC example is, but still presents designs with similar characteristics. The article (Jacobson et al. 1995) describes a simplified design of a telecom switching system. This example illustrates how objects are grouped into components. Figure F.3 below is taken from this article. It shows how the objects of a system are grouped into four components. The components can consist of more than one object. For instance, the Network handling component consists of three objects: the Network, Coordinator and Digit Information objects.

The observers of a component form the component's context

In the model-view contract there are two named components: the model and the view. However, there are more methods found in the classes than those corresponding to the messages sent from these two components. It is therefore obvious that the context of the model is expected to send more messages than those found in the view component. When viewing the contract as having two components, the components may be:

- the model component and
- a component including the view objects and other objects in the model's context

This interpretation of the components in the contract is based on the fact that the methods *attach()*, *detach()*; *setvalue()* and *getvalue()* are included in the description of the model component. Neither the model component itself nor the view component are specified to send these messages. Also, the view component has methods not called by itself or the model component, namely *draw()* and *subject()*. One can then assume that the model can receive the messages *attach()*, *detach()*; *notify()*, *setvalue()* and *getvalue()* from its context and the view can receive the messages *draw()* and *subject()* from other objects in the context. In other words, the contract states that the context of the model component can send more messages than the messages explicitly found in the model and view components.

The objects which form the context of a component are called *the observers* of the component.

The different kinds of actions

In the model-view contract there are two kinds of actions, namely message-send actions and assignment actions. In object component systems there may also be two other kinds of actions, namely actions which create new objects and there may be errors.

A message-send action can be described by giving the name of the receiving object, the message selector and the parameters of the message. For example, model!setvalue(123) can describe an action to some object named model. The message selector is *setvalue* and the value is 123.

An assignment action can be described by giving the name of the updated slot and the name of the object holding the slot and also the new value of the slot after the action is finished. For instance, model.state:=123 can describe an assignment action which assigns the value 123 to the slot names state in the object named model.

An object creation action can be described by the template used to create the new object and an error action can be described by naming the object where the error occurred. Such actions are further discussed below and defined in more detail in chapter 3.

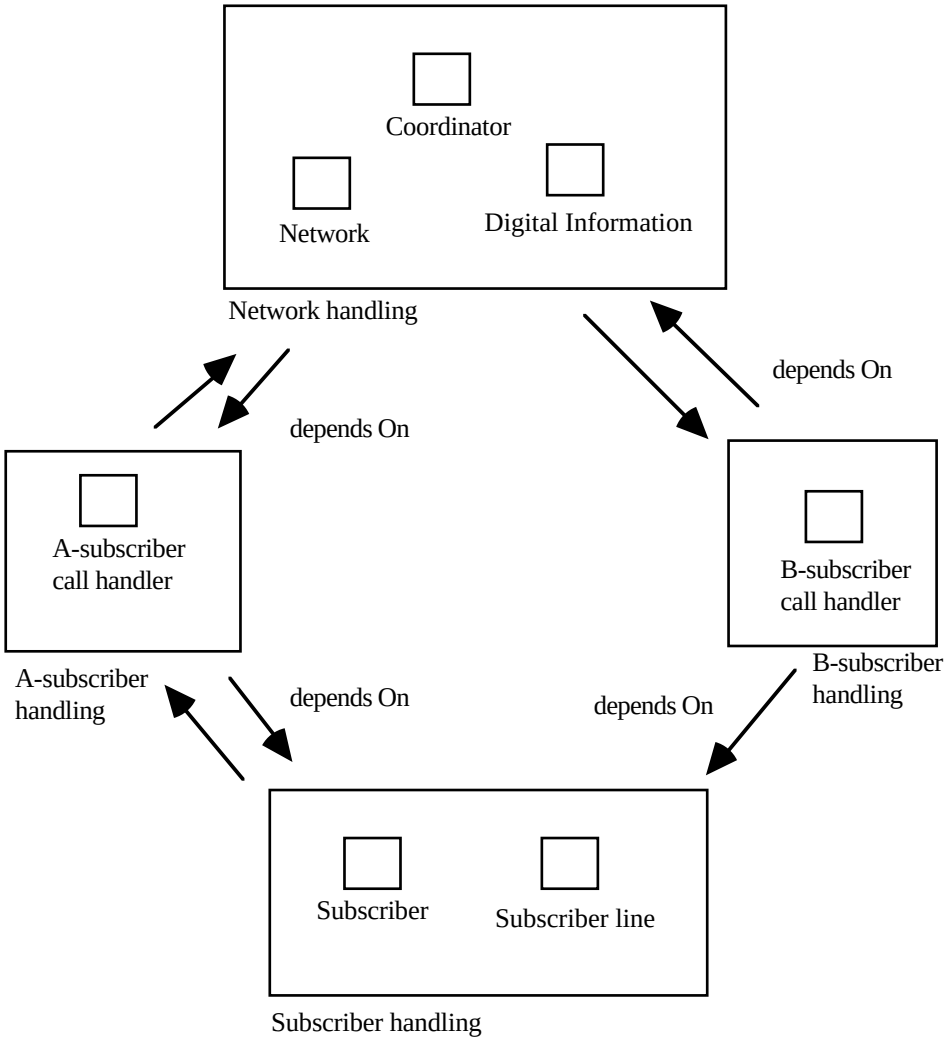


Figure F.3: Subsystem partitioning of the telecom example
The rectangles define subsystems grouping the enclosed objects into components.
The arrows define abstract relations between the components.
Figure from (Jacobson et al. 1995).

Refinements of contract parts

(Helm et al. 1990) uses a *contract specification* to specify how classes are related to their abstract counterparts in a contract. There are no theoretic or automatic support to show that a class actually implements a part of a contract. The relation between a part of a contract and a class implementing the part would correspond to the refinement relation introduced in chapter 1. A class implementing the model part of the model-view contract would then be a refinement of the model part relative to the context described in the contract.

2.3 Observable Behaviour

2.3.1 The observable behaviour of the model

The model-view contract of the previous section focuses on the behaviour of the model as observed by the view objects. The contract states that the observable behaviour of a model is as follows:

- a model has the ability to hold a set of objects, here denoted *views*
- a view object is added to *views* by sending the message *attach(v)* where v is the name of the view object
- a view object is removed from *views* by sending the message *detach(v)* where v is the name of the view object
- when the model receives the *value()* message all objects in the receiver's set of views will receive an *update()*- message

What follows is a more general discussion of the observability of the different kinds of actions in a component.

The observability of message-send actions:

A message-send action is observed by the receiver of the message. Observable message-send action is the focus of contracts as shown in the model-view contract example above. Observable message sending is also the focus of interaction diagrams found in many design notations. Both, Objectory, UML, OOram and others include such diagrams. An example describing message sending in the model-view contract follows:

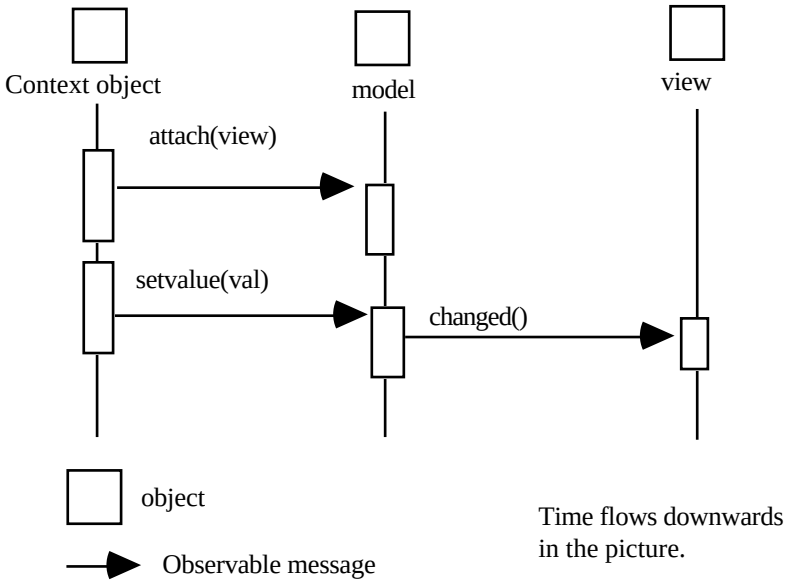


Figure F.4
An example of an interaction diagram which shows how objects collaborate by sending messages.

The observability of assignment actions

Observable assignment actions are not very common in OCS designs. Therefore, the various design notations rarely have syntax for describing variable access. Instead, data is transferred between objects by sending messages. This is also true for the contracts notation.

The model-view contract does not specify observable assignment actions since all assignment actions are only assignments to variables which are local to the object performing the action. Contracts only state properties of observable message-send actions and relations between the values of variables. If collaboration between objects which may access each others variables are to be described, the contract notation has to allow the description of observable assignment actions.

Observability of assignment actions is defined so that an assignment action is observed by the object holding the updated slot. If assignment actions are to be observed by one component while the assignment is done by an object in another component, objects have to be allowed to access variables in other objects. This is allowed in,

eg, C++ and Java by declaring variables as "public". It is also possible in languages like CLOS and SELF where one object may inherit, and thereby access, variables in another object.

The observability of object creation actions:

Object creation is not included in the contract example above. Most design methods do not include observable object creation actions in the various diagrams and notations. However, in many existing object-oriented systems, components create objects as part of their observable behaviour. For example, Smalltalk view objects create controller objects in a well documented way. Programmers can use this feature to make a view create the kind of controller they want.

In this thesis, observability of object creation actions are handled as described below. Other alternatives are left for further study.

If the creation action is to be observed, this is modelled by placing the template object in the context and thereby including the template in the observers of the component. In relation to the example with a view component having an observable object creation action, this corresponds to placing the template for controller creation as an object in the view's context.

If creation of some objects is not to be observed, this is modelled by placing the template in the component creating the object.

The observability of errors

The choice of an error model for a formalism will influence the results stemming from use of the formalism. The following error model is used in this thesis:

> an error action is observed if the error occurs when executing a sentence in the observing configuration
> otherwise, the action is not observed

This corresponds to the following error models in parallel and sequential systems:

> *Error model in a parallel systems:*
>
> Each method is seen as executing separately from the rest of the system. Then if one method terminates because there has been an attempt at performing an erroneous operation, the rest of the system may continue to execute. In such a case an error action in one component is only observed by other components as the absence of actions.

> *Error model in a sequential systems:*
>
> If there is an error in one part of the system no more operations are executed. Then the whole system stops since the execution of the program terminates. The last action in such a system will then be an error action. Usually, in practice, this error action is just indirectly observed by the fact that no more actions occur in the system or that the program stops and an error message is displayed.

The choice of error model for a formalism will influence the results stemming from the use of the formalism. This means that another error model which, eg, differentiates errors depending on what went wrong, might lead to another definition of observability of error actions. This in turn will lead to a different definition of similar observable actions. This might lead to other reliability requirements etc. and thereby give quite different results from what is presented in this thesis. Other error models is left for further study. However, as discussions below show, the reliability requirements seems to cover many other error models as well.

2.3.2 Hidden behaviour

The messages sent from an object in a component to another object in the same component are internal, and are part of the component's operational specification, the implementation, of a component. These details are therefore not part of the observable behaviour of the component. For example, the model sends itself the message *notify()*, and this will not be observed by the context. This action is therefore a *hidden* action which is not part of the model component's observable behaviour.

An action updating a variable in a component and where the action stem from execution of sentences in the component itself, is a hidden action. For example, when the model component updates its *views* variable, this is hidden from the context. Therefore, an implementation of the model part of this contract can delegate to some other object to hold the views. If the assignment action was observed, it would mean that a variable corresponding to *views* had to be present in all model implementations.

Creation of new objects by using templates which are considered to be within the component are not observed by objects outside the component. For instance, if the model component creates an object which is to handle the set of views, this creation action should not be observed by the context. If this action was observed, this would lead to restrictions on how the model component could be implemented. It would then be impossible to have components with more objects than described in a specification.

Error actions which are not observed are hidden. This means that actions which stem from execution of a sentence within the component is not observed by the component's context. This means that even when an error occurs in the model component, the model might still fulfil the model-view contract. The contract is fulfilled as long as the error does not influence the models collaboration with the objects observing the messages specified in the model-view contract. Therefore, from the perspective of the observers of the model-view contract, such an error action is hidden.

2.3.3 Observability of actions stemming from execution of sentences in the context

When sentences in the observing context are executed, the resulting actions may be observable to the context or not. It is not obvious how to categorise such actions stemming from execution of sentences in the observing objects. The solution presented in this thesis is based on practical experience with extensible object systems. In relation to observability of actions, the selected solution is to treat actions stemming from execution of sentences in the context in the same way as actions from sentences in the component.

A simple example which illustrates the consequences of this solution is found in the model-view contract above. The view component (the context of the model component) sends itself the message *draw()*. This message is observed by the view component since the view is part of the observers of the model. On the other hand, if the view is viewed as a component and the model and the other objects are the view's context, the *draw()* action is not observed.

For the next example we let the model be the component. Assume that the context of the model sends the model a message, eg, the *setvalue()* message. This action is not observed by the context since the receiver is not found among the observers of the model component, but rather in the model component itself. Since the model is the component, it is not part of the observing context. Therefore the *setvalue()* message is not observed. If the component/context is switched, ie, when the model is the observer of the context, the *setvalue()* message is observed.

2.3.4 New objects can be added to observers

According to the definition of an object system, each object is part of exactly one component. When new objects are created they must therefore be placed in some component. The question is then where to place new objects. There are three obvious alternative places to put the new objects:

In the component where the template object was found
In the component where the executed sentence which created the object was found
Somewhere else

The placement of new objects will influence the definition of observable behaviour because it will influence the choice of which objects make up a component and which objects will make up the observers of a component.

In the designs which have been studied as the background for the definitions of this thesis, the observability of an object creation action and the placement of a created object are always linked in the following way:

an object created by an observable object creation action is placed in the observing context and
an object created by hidden object creation action is placed in the component
where the executed sentence is found.

2.3.5 The observable behaviour of boundary objects may be non-deterministic

The view's observable behaviour is simply:

the view may send the message *getvalue()* to the model

The observable behaviour of the other objects in the context is indirectly specified by the list of methods defined for the model and view components. These are:

the context can send the model the messages:

attach(), *detach()*; *setvalue()* and *getvalue()* and maybe *notify()* (see discussion below)

the context can send the view the messages:

subject() and maybe *draw()* and *update()* (see discussion below)

It is normal to interpret the contract so that the methods listed can be called in any sequence any number of times. In other words, it is not determined in what order the methods will be called, ie, the context has non-deterministic behaviour. The context's behaviour will reflect the users behaviour and may then be seen as modelling aspects of the systems users. The context can therefore be viewed as a boundary component with non-deterministic behaviour.

In many OCS design methods, non-deterministic behaviour is used when a component's behaviour should not or can not be specified in detail, eg, a user has non-deterministic behaviour when it is not known in which sequence the user chooses to give commands to an application. In many designs the user's behaviour is therefore modelled by the non-deterministic behaviour of a user component, and the rest of the system should function without errors independently of which sequence of commands a particular user chooses.

Some OCS design methods expect the designer to specify boundary component behaviour more explicit than done in the above contract specification. For instance, Objectory focuses on giving exact specifications of actor objects' non-deterministic behaviour since this defines the system's functionality as seen from the users' point of view.

2.4 Observable Similarity

This section gives an informal, but detailed description of similar observable behaviour. First observably similar actions are described. Then it is described how actions from execution of sentences in the context as well as actions from execution of sentences in the component, are viewed relative to observable similarity. Next, it is described how similarity of behaviour is viewed when the specification has non-deterministic behaviour. The topic of the last part of this section is observable similarity of terminated components and components with sequences of hidden actions.

2.4.1 Observably similar actions

This section describe what it means for two actions to be observably similar. This is used in defining similar observable behaviour of components. Similar observable behaviour means sequences of observable actions which are pairwise observably similar.

Assume that the two actions to be compared are both observable actions. To be similar, the two actions must be of the same kind. This means that only message-send actions are similar to message-send actions, assignment actions are similar to assignment actions, etc. Furthermore we have:

> Two message-send actions are observably similar to each other when they are messages to the same object. Also, the message selector is the same in the two actions. The parameters which are names of observing objects are equal. Parameters which are not found as the name of an object in the context may be different. However, both parameters must be different from context object names.
>
> Two assignment actions are observably similar when they update the same slot in the same object. If, in either of the actions, the new value of the updated slot is the name of an observing object then the new value is the same in the two actions. If the new value is not found as the name of an object in the context, then both actions must have values which are different from context object names.
>
> Two object creation actions are observably similar when they create objects from the same template.
>
> Two error actions are equal when they are errors in the same object.

Below some examples are used to illustrate the consequences of this definition.

Message-send actions:

Consequences of requiring *message selectors* to be equal in message-send actions:

> In the definition of object-oriented systems in (Blair et al. 1991) it is said that an important characteristic of object-oriented systems is that execution is controlled through message passing, in (Blair et al. 1991) termed "dynamic binding". When there is dynamic binding in a system, the mechanism which selects what method to perform in response to an object receiving a message, is the basic mechanism to select alternative execution paths in a program. This selection mechanism is therefore the basic control mechanism in the system. Therefore, if the message selectors were different in two otherwise equal actions sent to, eg TextEditor in the two systems MyModel||TextEditor and TextModel||TextEditor, then TextEditor's execution may proceed differently in the two systems.
>
> One intention of defining observable equality is that when two message-send actions are observably similar, the receiving observer is expected to behave similarly. When the system has dynamic binding and we have this intention with our definition of observable equality, it is therefore natural to require that two observably similar actions have equal message selectors.
>
> It is also a common perception among component designers that *message selectors are equal in two observably similar actions*. Therefore this alternative is used in the present work while the alternative: allowing message selectors to differ, is left for further study.

Consequences of requiring *receivers* to be equal in message-send actions:

> If receivers were allowed to be different in two observably similar actions, the two receivers may respond in different ways. We would then also get as a consequence that TextEditor's execution would proceed differently in the two systems MyModel||TextEditor and TextModel||TextEditor after two observably similar actions with different receivers.

It is also a common perception among component designers that *receivers are equal in two observably similar actions*. Therefore this alternative is used in the present work while the alternative: allowing receivers selectors to differ, is left for further study.

Consequences of requiring parameters, which are context object names, to be equal:

One consequence of requiring parameters which are names of context objects to be equal, is that a context will receive the same context object names in the same parameter positions from a specification and its refinements.

The model-view contract does not give examples of messages from a component with parameters which are names of objects in the context. Such examples are not very common in OCS designs. In the Smalltalk system there are a few examples of messages which have such a functionality. Most uses of these messages are found in the class named Dictionary. The messages are used for retrieving objects which are stored in Dictionaries. For instance the Dictionary class has the following interface:

```
Dictionary:
        at: index put: item     "Stores the object referred to in the variable item under index"
        get: index              "Returns the object stored under index"
```

The consequence of the definition of observably similar actions, would in this case be that a context would find two versions of Dictionary observably similar if the same object name was returned after equal sequences of at:put: and get: messages. If one Dictionary would return a different object than the other, after the same get:-message with the same index as parameter, the two Dictionaries will not have the same observable behaviour.

Consequences of allowing parameters to be different when they are *not* the names of context objects:

Allowing parameters to differ when they are *not* names of objects in the context, allows a refinement to use different object names than the specification. It also allows a refinement to contain a different number of objects than the specification. In this way, the names of the objects used in a specification and the number of objects in the specification are not part of the observable behaviour of a specified component.

It is possible to make an illustration of this case based on the model-view contract. We let the view be the observing context and we let the model and the other objects in the system be the component called model. The view will receive one or more *subject()* messages with a parameter, namely the name of some model object. The view would observe two *subject()* messages as equal even when the parameters were different. This difference means that the name of the model object might vary in two versions of the model component.

The model-view contract does not give any examples of components with more than one object. It is therefore not straightforward to use this example to give an illustration of how the definition of observably similar message-send actions allows a refinement component to have a different number of objects than the specification. However, in the example referred to in figure 7 there are components with more than one object. For instance, the Network handling component consists of three object. The definition of observable equality would allow a refinement of the Network handling component to have one, two, three or any number of objects. However, there is a limit to the number of objects which a component can let the context know of. Such objects are called *visible objects* and are further discussed in the section on reliability requirements later in this chapter.

Assignment actions:

Let the view in the model-view contract be an observer of the other objects in a system. There is only one possibility for observable assignment actions when the view is the observer, namely assignment to the subject variable. Two observed assignment actions will assign values which are the names of some model objects. These names may vary. This allows two model components to have different object names for the objects representing the model component in the model-view collaboration.

Object creation actions:

Consider the case in Smalltalk where the view component creates a controller object. Two view components have observably similar object creation actions if they create a controller from the same class.

Error actions:

> If TextEditor observes an error action, the error action is the result of executing a sentence in one of the objects in TextEditor. Two observably similar error actions from TextEditor will then be errors which are the results of executing the same sentence in TextEditor, one error when the sentence is executed in TextModel||TextEditor and one error when the sentence is executed in MyModel||TextEditor.

Chapter 4 gives a formal definition of observably similar actions as defined above. This formal definition is given by a relation called the *observably equal actions relation*. The relation has a name with "equal" in it since it is an equality relation. This is shown in a proposition in chapter 4.

In chapter 5 it is shown that this definition of observable equality does not give reliability when used as a basis in the definition of a refinement relation. Chapter 5 therefore includes a modified version of the observably equal actions relation which can be used as a basis for defining a reliable refinement relation. This modified version is called the observably similar actions relation. The word "equal" is not used in the name of the modified definition since the new relation is just a pre-order.

2.4.2 Observable similarity from execution of sentences in the component and in the context

In what follows, *an action stemming from a sentence in the context is seen as observably unequal to an action stemming from a sentence in the component*. Using the model-view contract as an example, this means that, eg, a *notify()* message to the model sent from the model itself is viewed as observably different from a *notify()* message sent from the model's context.

The consequence of this decision can be illustrated using the two systems MyModel||TextEditor and TextModel||TextEditor where TextEditor is the observing context. The traces of the two systems are compared. Assume that there is one action from each system and the two actions are observably similar.

If there are two message-send actions, eg, two *update()* messages, which are observably similar relative to TextEditor, then either both actions stem from execution of a sentence in the TextEditor, or both stem from execution of sentences in the components MyModel and TextModel.

If the system is sequential, at most one sentence is executable at any one time. This means that if *update()* came from execution of a sentence in TextEditor, then both the observably similar actions must stem from execution of the same sentence in TextEditor.

If the system was a parallel system where objects can execute in parallel, then an *update()* message from the component TextEditor can stem from different sentences. This can happen if there are two sentences which can be executed and one is executed in TextModel||TextEditor and the other in the MyModel||TextEditor. In what follows it is assumed that *two actions stemming from different sentences in the context are seen as observably unequal*. This means that either both actions stem from execution of the same sentence in the observing context TextEditor, or both stem from execution of sentences in the components.

The consequences of choosing actions from different sentences in the context to be unequal is further discussed in the chapter 5. The conclusion is that when "clean and tidy" specifications of reliable refinements are made, an action from a sentence in the context will always be observably equal to the action from execution of the same sentence, no matter which component it is collaborating with. Therefore, this will not in practise make any difference in relation to determining whether or not a component is a refinement of some other component.

It is left for further study to look at the consequences of choosing an alternative definition of similarity of actions relative to where the executed sentences is found.

2.4.3 Similar observable behaviour to non-deterministic behaviour

The context's observable behaviour was specified above as being non-deterministic. Objects' behaviours are specified to be non-deterministic when the behaviour should not or can not be specified exactly.

A refinement of the context must have observable behaviour corresponding with some or all of the alternative behaviours given by the non-determinism of the context specification. In this view, a context which only

displays one of the alternative observable behaviours will be seen as having similar observable behaviour to the specification. This means that a user which always gives the same sequence of commands to an application is seen as a refinement of some actor (or environment role) describing what users can do in general.

Example:
The behaviour of the context towards the model is not specified in detail in the model-view contract. The context behaviour is perceived as being non-deterministic. Any implementation of the context which display one or more of the possible behaviours fulfils the specification since the model is expected to function without errors independently of which of the non-deterministic behaviours is displayed.

Another way of viewing this is that every sequence of observable actions displayed by the refinement must also be a possible sequence of observable actions given by the specification.

When a refinement only displays some of the alternative behaviours of the specification, then no new errors should be introduced in the rest of the system when combined with the refinement. One consequence of this is that if there are no error when the rest of the system is combined with the specification, it is expected to function without errors no matter which of the alternative behaviours is displayed.

2.4.4 Observing termination and sequences of hidden actions

Components which have terminated and components which execute giving a possibly infinite sequence of hidden actions are seen as equivalent in that none will give observable actions. There are alternatives to this view, but they are left for further study.

2.5 Reliability Requirements; Two Examples

To illustrate what is meant by reliability and reliability requirements, this section presents, very informally, two simplified reliability requirements:

A specification must specify all messages sent from the context to the component.
A specification must specify a component's visible objects.

The first example requirement does not apply to systems with non-deterministic behaviour. To describe the details of why this is so would be outside the scope of this section. However, the situation arises since a component can respond in different ways and thus control which messages it later will receive. The word "all" is therefore too general in such a case. This is further discussed in chapter 8 in the section discussing the reliability requirement "reliable message sending" in relation to traditional subtyping.

The other example requirement applies to all reliable specifications.

A specification must specify all messages sent from the context to the component

In order to know what a refinement must include and what a designer of a refinement may change, a contract must include something indicating the availability of methods. A small example illustrating this problem follows:

In the model-view contract, it is not clear whether the context is also allowed to call the model-method *notify()*. Knowing if *notify()* can be called from the context or not is important. If *notify()* is only used as part of expressing the *model-view* contract, an implementation need not include a notify()-method. However, if *notify()* is sent from the context, leaving a corresponding method out would lead to unwanted errors.

Choosing to leave out a method for *notify()* in the implementation of model might not create errors when combined with "nice" contexts not calling *notify()*. When a "nice" context is later replaced with a context made from the same contract, but which calls *notify()*, then an error will occur and the system does not function as planned. Therefore, the model-view contract's specification of the context's observable behaviour is *not reliable* in that it is possible to interpreted the contract in different ways which can lead to unanticipated behaviour.

Summing up this discussion: If the context specification is not reliable, the result is that it is possible to make an implementation of a model which works without errors when collaborating with a "nice" context. When the same model is used in another context, there is a chance of previously unseen errors appearing.

To avoid such errors it is therefore necessary that a specification specify all messages a context may send to the specified component.

This reliability problem is well known. Programming languages such as C++ and Java allow the programmer to define the availability of methods, and thereby specify the expected observable behaviour of the context of the objects of the class. This is done by marking the methods with a label indicating if the methods are private (only to be called from the component itself) or public (can be called from the context). A label named friend is used to limit the availability of methods, so that they can only be called from objects of some explicitly named classes.

Type checking is then used to ensure that only implemented methods are called - or is it to ensure that all called methods are implemented. This would depend on point of view. For instance it can be discussed whether it is the context or the component which put requirements on the other. If the component sets the requirements it must be checked that the context only calls implemented methods. If the context gives the requirements, it must be checked that the component implements all called methods. This thesis presents a context centred view where it is the context which sets the requirements. This is motivated by the fact that it is the context which is to function as planned, also when the component it replaced. Therefore a component must behave as expected by the context. This is as opposed to the common component centred view often presented in relation to type checking. In this view a component define what methods its context may call. A component can then be placed in all context which only call the specified set of methods.

These different views can also be exemplified by the following two statements concerning how components can be used when they have observable message-send actions:

Context centred view:
A component can replace another component if the component implements all the methods called by the context and send all the messages expected by the context.
Component centred view:
A component can be placed in any context which only calls the set of methods found in the component.

Because this thesis represents a component centred view, as opposed to the more common component centred view, this thesis has a stronger focus on the context than what is found in most related work which is usually based on the component centred view.

A specification must specify a component's visible objects.

Objects are message receivers, components are not:
In object component systems, objects are the receivers of messages while components can not receive messages. If there are more than one object in a component, the context can send messages to more than one object. For instance, if the model component consists of two objects, for instance called model and collection, the context could send messages both to the collection object and to the model object.

For example we might have a case where the collection object has methods to add and remove elements in a set, ie, the *attach()* and *detach()* methods are only found in the Collection class. The context must then send messages to the collection object to add and remove views. An error will then occur if the messages are sent to the model object. The error occurs since there are no corresponding methods in the model object. Furthermore, if the *setvalue()* and *getvalue()* messages should be sent to the model object, the context must be able to distinguish between the two objects in the model component.

Visible objects:
In a specification of a component with more than one object, it is therefore necessary to identify the different objects which will receive messages from the context. Objects which receive messages from the context are *visible objects*. If the context sends messages to the collection object, then the collection object is visible. We say that the collection object is *visible to* the context. We also say that the collection object is one of the *visible objects from the model component*.

In order to specify the model's observable behaviour it is also necessary to specify objects which are *visible to the model*, not only the model objects which are visible to the context. If the context's visible objects were not specified, it would be difficult, if not impossible, to specify which objects the model is expected to send messages to. That is why there are expressions for instantiating objects in the language used to define contracts. In the model-view context the view objects are visible from the model's context to the model since these objects get messages from the model component.

Hidden objects:
If there are objects in a component which do not receive messages from the context, but only receives messages from other objects in the same component, the objects are not visible. Such objects are *hidden objects*. If hidden objects occur in a specification of observable behaviours, the hidden objects are only used to express the observable behaviour. A refinement and its specification, therefore, need not have similar hidden objects as long as the refinement and specification have the same observable behaviour.

Examples of observers, visible and hidden objects:
In the model-view contract the following hold:

- The view objects and the other objects in the model's context are the observers of the model.
- The view named "someview" is visible *to* the model after the message *attach(someview)* is sent to the model and the attach-method executed. *someview* is visible from the context.
- The model named *somemodel* is visible *from* the model component after the view has received the *subject(somemodel)* message and the subject-method executed. *somemodel* is visible to the context.

There may be components with no visible objects. Such components usually model the environment which interacts with the system. Visible and hidden objects are formally defined in later chapters.

Observers, visible objects and component encapsulation:
The number of visible objects from a component may be used as part of a measure of a component encapsulation. We can say that a components functionality is more encapsulated the fewer visible objects there are from the component. Chapter 8 discusses component encapsulation and other results from the theoretic work which are relevant to practical OCS design.

Encapsulation as defined in the object-oriented tradition usually emphasises the encapsulation of the variables in single objects. Encapsulation of components consisting of more than one visible object is rarely discussed. With

the division of a component's objects into visible and hidden objects, and also in hidden and observed actions, it is possible to define various degrees of component encapsulation.

The conclusion from this small discussion is: since it is objects which are the receivers of messages, and not components, it is necessary to specify which object in a component should receive the different messages. The objects receiving messages from the context are visible objects.

Other reliability requirements

The first example of a reliability requirement is related to a well known problem which occurs when methods are missing from component implementations and/or when contexts send messages which components do not understand. Compile time type checking in object-oriented languages are done to help avoid this problem. However, as mentioned above and discussed in chapter 8, traditional type checking does not solve this problem for all systems.

The second example of a reliability requirement is not related to a well known problem in the same way as the first example requirement. However, there are some relationships between this second example requirement and some related work. This is also discussed in chapter 8. However, the related works do not argue for their solutions from a formal basis, but argue based on what are practical solutions when designing systems.

No related works focus on requirements for component specifications and implementations in order to get reliable substitution of components. There are some works which present solutions which touch the same area of concern, but no work presents the problem in general based on the characteristics of object component systems.

Compile time type checking is done to ensure reliable substitution for components in functional systems. Characteristics of functional systems and object component systems are quite different. Therefore, it is not surprising that traditional type checking does not ensure reliable substitution for OCS components. The interesting question is then: what is necessary to ensure reliable substitution of OCS components ?

Type checking of functional systems is a topic which has been and still is studied by many groups of researchers. There are many details to be sorted out in order to do type checking correctly so that it is ensured that no unanticipated errors will occur. Similarly, many details must be sorted out to find out how to ensure reliable substitution of OCS components.

This thesis is a first step in finding the necessary requirements on specifications and implementations in order to avoid unanticipated system behaviour when components are combined. In this first step, certain alternative views on objects, components and object behaviour have been chosen. The choices have influenced the results and different choices might give different answer to the question of what is necessary to ensure reliable substitution. Further work will show how different choices will influence the answer and perhaps show which choices will be the best alternatives for creating reliably substitutable components.

Chapter 5 presents a set of requirements necessary to get reliable substitution. The requirements are based on the definition of object-oriented concepts presented in chapters 3 and 4 and should therefore be read after these two chapters.

CHAPTER 3

Modelling OCS Properties Using Omicron

This chapter presents a simple object-oriented language for describing systems and components with behaviour as described in chapter 1. Since it is small and is used to define objects, the language is called Omicron after the short o of the Greek alphabet.

The Omicron language is a simple prototype-based language and therefore has only a very limited set of concepts, mainly name, slot, object and executable sentence. Most other concepts found in object-oriented languages can be modelled by using the Omicron concepts. Examples of how Omicron is used to model methods, templates for object creation, inheritance and some other object-oriented language features are explained below. For more examples see Ungar et. al. (Ungar et al. 1991) which gives a good overview of how traditional object-oriented language features can be simulated by constructs in more simple prototype-based languages.

Section 3.1 gives an informal introduction to the Omicron language. This section shows examples of how object-oriented concepts such as objects, methods, systems, and components, are represented in the Omicron language. It is shown how the object concepts of Omicron are used to model both objects and methods.

Section 3.2 shows how Omicron systems are executed. The result of executing an Omicron system is a sequence of actions. The actions describe the behaviour of the objects in the system. An Omicron expression is both the code of a system which a programmer writes and the representation of an executing system. How this is done is shown through examples in section 3.2. This section also shows how executing methods, self-reference and class features such as inheritance and templates for object creation are modelled in Omicron.

Section 3.3 shows how the model-view contract of chapter 2 can be defined using Omicron.

The complete syntax of Omicron is summed up in section 3.4. This section gives the formal semantics of the language. The formal semantics is given operationally through a set of transition rules. This gives the Omicron calculus which is used when reasoning about components' observable behaviour. The Omicron language and calculus is also called the Omicron formal framework, or just the Omicron framework.

A reader who is familiar with BNF syntax and language definition through operational semantics may read section 3.4 to get a complete understanding of the language. A reader with little training in BNF and formal language definitions may skip this section, except for the introduction. Section 3.4 also defines some basic notations for describing sequences of actions and the trace of an execution. This notation is used throughout the rest of the thesis. The notation is defined both informally and formally. This is done in the hope that both readers with and without training in formalisms can understand what is necessary in order to follow the discussions and conclusions in the rest of the thesis at an appropriate level.

Section 3.5 explains why the formal definition of the Omicron semantics was done as presented in section 3.4.

This chapter does not give an introduction to object-oriented concepts. It is assumed that the reader is familiar with object-oriented concepts such as object, method and message. Instead, the chapter presents how the object-oriented concepts central to the formalisation of object component system designs are modelled in the Omicron language. For definitions of object-oriented concepts see, eg, (Blair et al. 1991).

3.1 A Simple Object-Oriented Language

This section first gives some arguments for why Omicron is created. Then it argues for limiting the number of concepts in the language before presenting how object-oriented concepts such as objects and methods are represented in the Omicron language. This section also gives a formal definition of a system and a component. The presentation of Omicron will be developed further in later sections, where section 3.2 gives examples on how Omicron systems are executed, and section 3.4 gives a complete formal definition of Omicron.

3.1.1 Why the new language Omicron is created

Object-oriented technology featuring the concepts of object identity, instance creation, encapsulation, dynamic binding and inheritance is increasingly popular when designing systems. However, there is a lack of formalisation tools. As shown in chapter 9 on related work, the common way to make formal models of objects is to base them on models which are supported by existing calculi, ie, functional models (eg, lambda calculus) or models of communicating processes (eg, π-calculus (Milner et al. 1989a)). As discussed in chapter 9, this is not an ideal way to formalise objects as found in object component systems. This is mainly because it is not intuitive how definitions and proofs done in these calculi map to OCS concepts. The mapping is not intuitive since the concepts in these calculi and the OCS concepts are fundamentally different. The Omicron calculus differs from these calculi in that it is based on the OCS concepts. Definitions and proofs of properties of OCS components and systems can therefore be done by directly using OCS concepts.

There are a few formal models of objects which have concepts which are more in line with the OCS concepts. However, as said in chapter 9 on related work, all of them have some serious weakness in relation to showing reliable substitution of components with characteristics as described in chapters 1 and 2. Several of these formal models were studied and attempts were made to eliminate the weaknesses. However, as work progressed it became obvious that the weaknesses were due to fundamental properties in the models and were therefore quite inherent to the models. Each model would therefore need substantial rework to eliminate its weakness. It would therefore be much simpler to define a new formal model specially suited to reason about substitutability of OCS components. Consequently, a new formal model was created and it was named Omicron. Actually two formal models were made. Why two models is explained below.

When working to find a way to formalise OCS components, many alternative languages and rules for describing the semantics of the language were attempted before the two presented versions were selected. Most of the alternatives became unnecessary complex without being sufficiently better, or more interesting to justify presentation, except for these two alternatives.

The main reason most alternatives became unnecessary complex and only two versions were interesting to present is that there are basically two different ways to view the execution of a sentence in a program. A sentence may be seen as an atomic operation to be performed, or as an expression which is to be evaluated. The first alternative is found in process languages like CCS (Milner 1989) and the π-calculus, while the latter is found in functional languages such as ML (Milner et al. 1990). OCS designs have relationships to both of the these traditions. When defining the language for describing objects, one or the other alternative should be selected to keep the syntax and semantics simple. Therefore this thesis presents two versions of Omicron. In chapter 7 a language with *expression evaluation semantics* is presented, while the present chapter describes a language with *atomic operation* semantics. When the two alternatives are compared, it is clear that in many ways the atomic operation semantics gives the simplest calculus. The simplest version is therefore chosen as the main version and is presented below.

The formalist traditions of Europe (eg, the π-calculus), the pragmatic traditions of California (eg, SELF and Smalltalk) and the C tradition (eg, C++ and Java) have different concrete syntax preferences. Since the work presented here is strictly formal, the Omicron syntax follows the formalist tradition, thus easing the formalisation work. However, it has also taken some inspiration from the syntax of SELF. (This is the syntax for inheritance and input slots since there is no formalist tradition for the syntax of such slots.)

Omicron is not meant to be a practically usable object-oriented programming language in itself, since many features which usually are handled by runtime systems have to be handled manually in Omicron. When developing object-oriented systems, the specification, design and implementation may be done using more user friendly notations such as an object-oriented programming language or design notation. Examples of such notations are OOram, Syntropy and UML. Formal proofs can then be done by translating from the more user friendly form to Omicron. In this way proofs of, eg, refinement relations between specifications and realisations can be done.

3.1.2 Advantages of limiting the number of concepts

An advantage of limiting the number of concepts is that it reduces the number of concepts which must be defined. The number of concepts can be defined by for instance having a single concept to represent both object and method. This eliminates the need for a special syntax to distinguish between them and this reduce the number of syntactic elements whose semantics must be formally defined. Similarly, using objects as templates for other objects and allowing inheritance between objects eliminates the need for special syntax to distinguish classes and objects, while still retaining the class properties related to inheritance and object creation.

One reason for creating Omicron is to understand more about categorisation (*typing*) of objects by studying monotonic relations between OCS components. Therefore it is important to avoid traditional object-oriented typing and other such concepts which are introduced to help developers write correct code. Such concepts must be avoided since they have strong relations to equivalence and monotonic relation definitions. The formal language will therefore have no distinction between type, class and object.

In traditional object-oriented systems, objects are both described by sentences in a program and represented by bits in memory when the program is executed. This distinction between the program-description and the representation of an executing system is eliminated in Omicron and many other similar languages such as π and λ. This is done by having the language expressions represent both a program-description and an executable representation. How this is done is explained in section 3.2.

3.1.3 Names, slots, objects and methods

This section gives definitions of the OCS concepts name, slot and object as found in the Omicron language. For each defined concept, the Omicron syntax is shown. The definitions and syntax are illustrated by referring to the model-view contract in chapter 2. Below it is also shown how a traditional method is represented as an Omicron object.

Name

In the Omicron language there is no distinctions between what traditionally is the name of a variable, the name of a method and the name of an object (ie, the identity, key, unique name etc. of an object). All of these are just names. The set of names is denoted by $\mathcal{N}$.

Limiting the number of concepts in this way is an element in the basic assumption of not taking the reliability requirements for granted, as described in chapter 1. Limiting the number of concepts in this way also helps to simplify the formal definition of the Omicron syntax and thereby the reasoning done based on this definition.

However, when names are used in certain positions in Omicron expressions they are referred to as object names, slot names and slot values even if there is no syntactic difference. See below for details.

Slot

A slot is a pair of names denoted s→o. s is called *the name of the slot* and o is *the slot value*. As shown below, slots are both used as what is traditionally known as variables and sequences of slots are used as method dictionaries. This is similar to, eg, SELF.

Object

In Omicron an object may be defined as follows:

Example of an object description: e : ([s→o, w→m, t→j], s!w(t); s:=t; s := t clone;)

In this example e, s, o, w, m, t and j are all names. The example defines an object named e with three slots named s, w and t. Slots hold values which are names. In the example, the value of the s slot is the name o, the value of slot w is m and the value of t is j. The names found as slot values may also be found as object names and slot names in this or other objects in a system. The slot values may also be names which are neither found as object nor slot names in a system. Such names may for instance be used to represent constants.

The defined object has a body consisting of three *sentences*:

a message-send sentence	s!w(t)
an assignment sentence	s := t
a clone sentence	s := t clone

These sentences define actions the object performs when it executes. How sentences are executed is described in section 3.2.

Omicron slots are used as variables and sets of slots as method dictionaries. For example the model-object of the model-view contract of chapter 2 can be modelled in Omicron as follows:

model : ([state → v; views → o, setvalue → s, getvalue → g, attach → a, detach → d, notify → n])

where

the slots state and views correspond to the variables *state* and *views* as defined in the contract,
the rest of the slots correspond to the methods named *attach()*, *detach()*; *setvalue()*, *getvalue()* and *notify()*.

Also we have:

model is a name, referred to as the *object name*
s, g, a, d and n are names referring to objects modelling the actual methods,
v is a name representing the state of the model and
o is a reference to an object holding a set of views.

A complete Omicron description of the model-view contract is found further below in section 3.3.

In the operational specification of the model-view contract, there are executable sentences in methods, but not in the objects themselves. For instance there are sentences in the *attach()* and *detach()* methods which will change the *views* variable when they are executed.

Sentences in objects are not common in object-oriented languages, and even less used, although found in, eg, Simula and Beta objects. See, eg, (Birtwistle et al. 1973) for examples. In Omicron objects may have sentences. The main reason for this is that when objects can have sentences then methods can be objects as explained next.

A method is also an object

The example object named e above, could also be a method. The following example shows how a method in the model in the operational description of the model-view contract of chapter 2 can be defined using Omicron. The method is the one with the name *setvalue* and which has the parameter *val* of type V. Using Omicron, this method can be defined as follows:

s : ([self☆ → model, :val → nil] state := val; views!notify();)

where:

s is the name of the method, ie, an object name
the star (☆) in self☆ means that the method inherits slots from the object named model
(notation taken from SELF)
the colon in :val means that val is an input parameter (notation taken from SELF)
state := val; views!notify(); is a sequence of sentences defining the actions the object is
to perform when executed. Sentences will be further discussed in section 3.2 below

More examples of methods are given in section 3.2, subsection 3.2.4. Details of how methods and object executions are handled is described informally, but in more detail in section 3.2. A full formal definition is given in section 3.4.

The decision to have no syntactic distinction between an object and a method in Omicron was inspired by Beta. In Beta, classes and procedures are given a common syntax and are called Patterns.

3.1.4 Object systems and components

Object Systems

Definition: An object system

An object system is a set of objects where each object has a unique name.

Object systems are closed. With this we mean that the objects in the system only collaborate with each other.

Associating a unique name with each object is identified as a key property of objects in (Khoshafian and Copeland 1986). The unique name is seen as something different from the object's variable values (state) and the object's behaviour.

We let S.Dom denote the set of all object names in the object system S. This means that if 'model' is the name of an object in an object system S, then model $\in$ S.Dom is true. If model is not the name of an object in S, then model $\notin$ S.Dom. When there can be no misunderstandings, we let n $\in$ S denote n $\in$ S.Dom and n $\notin$ S denote n $\notin$ S.Dom for any name n.

Object names in theory and practise:
In practice there is a reuse of object names in that objects which do not exist at the same time can have the same name. This strategy gives some advantages in practise, mainly limiting the size of systems. To limit the size of systems, old unused objects are removed from the system in order to recover used memory space. The names of such removed objects can then be reused, while the uniqueness of object names at each instance in time is retained. Reuse of names is practical in that it limits the number of different object names which are necessary to ensure uniqueness. When the number of names is limited, then the size of the names is also limited. This in turn will also limit the size of object systems.

System size and memory use is not a primary concern when making theories. In theory it is simpler to assume that no object is ever removed from a system and all objects have unique names. This assumption is done in Omicron since it gives simpler definitions of the semantics of the language, ie, how systems are executed. It simplifies the descriptions of system execution since it is not necessary to describe how old unused objects are detected and removed or how names are recycled.

A component is a subsystem

A component consists of one or more objects. A component is a subset of the objects in a system and defines a subsystem:

Definition: Components

The objects in a system can be partitioned into components. A component, C, in an object system, S, is a non-empty subset of the objects in the object system, ie, $C \subseteq S \wedge C \neq \emptyset$.

An object in a system is part of one and only one of the components in the system. This means that the system S can be divided into the components $C_1,...,C_i$ if and only if the following holds:

$$\forall\, n : \mathcal{N},\ k,j : 1,\ldots,i \bullet \quad n \in C_j \quad \Leftrightarrow \quad n \in S \wedge (n \in C_k \Rightarrow k = j)$$

A component is an open system in that it interacts with its environment which is comprised of the objects in the other components in the system.

3.2 Execution of Omicron Systems

This section starts by presenting examples of how Omicron sentences are executed, something which results in sequences of actions. Then it shows how objects are used as templates for object creation and how inheritance is represented and handled. It also shows how methods are modelled as Omicron objects and how self reference is done. The last topic of this section is system errors.

Executing an Omicron sentence results in an action. The actions are of four kinds:

e->o!m(j_1,…,j_n)/k	a *message-send action* from a sentence in the object named e: o gets the message m with j_1,…,j_n as a parameters and where k is the name of the new method to be executed
e->o_1.s_1,…,o_n.s_n:=j	an *assignment action* from a sentence in the object named e: for i = 1..n, the slot s_i in the object o_i gets the value j
e->i.s:=k/o	a *clone action* from a sentence in the object named e: the object o is copied and given a new name k and the slot s in the object i gets the value k
e->error	an *error action* from execution of a sentence in the object named e.

These actions are explained below through examples. This section only presents examples of executing Omicron expressions and does not give all the details. All the details and possibilities are presented in the section 3.4 which gives a complete formal definition of the Omicron language.

3.2.1 Executing sentences in Omicron

In section 3.1 an object was defined as follows:

Example of an object description: e : ([s→o, w→m, t→j], s!w(t); s:=t; s := t clone;)

This object has a body consisting of three *sentences*:

a *message-send* sentence	s!w(t)
an *assignment* sentence	s := t
a *clone* sentence	s := t clone

These sentences results in actions which the object performs when it executes. To model execution in Omicron, an execution mark, denoted $, is introduced. The execution mark is placed in an object's body to show where the execution control is. By introducing the execution mark, Omicron configurations can represent both a program-description and an executable representation.

To make the above object an executing system, an execution mark can be placed in front of the object's sentences as follows:

e : ([s→o, w→m, t→j], $ s!w(t); s:=t; s := t clone;)

The next sentence executed is the one following the execution mark. After a sentence has been executed, the execution mark is advanced to the right of the executed sentence. Therefore, after execution of the first sentence, the object is described as follows:

e : ([s→o, w→m, t→j], s!w(t); $ s:=t; s := t clone;)

In this case, the next sentence to execute is s:=t.

The semantics of executing a sentence is described by an *action*. If error situations are ignored, the actions from execution of the example object above are:

e->o!m(j)/k	a *message-send action* from a sentence in the object named e: o gets the message m with j as a parameter, k is the name of the executing method which is the result of the message, this is explained in detail below
e->e.s:=j	an *assignment action* from a sentence in the object named e: the slot s in the object e gets the value j
e->e.s:=k/j	a *clone action* from a sentence in the object named e: the object j is copied and given a new name k and the slot s in the object e gets the value k

In general, a message-send action may have zero or more parameters. We then have a general form for message-send actions as follows:

$e->o!m(j_1,\ldots,j_n)/k$

Also, an assignment action may assign a value to one or more slots. We then have the following general form for assignment actions:

$e->o_1.s_1,\ldots,o_n.s_n := j$

When the execution of a sentence results in some actions, we say that the actions *stem from* the sentence.

Omicron has three kinds of sentences: message-send sentences, object creation sentences and if-sentences. The sentence s := t presented above was a simple version of the Omicron *if-sentence*. An if-sentences is written as follows:

s:=(v=w t f)

This reads: if the v and w slots hold the same name then s gets the name in t, if not, s gets the name in f. The sentence s := t may therefore be seen as a special case of the if-sentence: s := (t=t t t).

An example of an object with an if-sentence where the slot values of v and w are equal is:

e : ([s→j, v→k, w→k, t→i, f→j], $ s:=(v=w t f))

This will lead to the action

e->e.s:=i

If the object is defined as follows where the slot values of v and w are unequal:

e : ([s→j, v→k, w→l, t→i, f→j], $ s:=(v=w t f))

the action will be:

e->e.s:=j

Note that the names in the sentences are always slot names. This is also true for message selectors in message-send sentences. This is done in order to be able to model message selectors as slot values, something which is found in, eg, Smalltalk (Goldberg and Robson 1983).

It is not common to only use slot names in sentences. Particularly, syntax for message-send-sentences usually allow the programmer to explicitly write the message selector instead of the name of a slot whose value is the message selector. In this latter case, as found in Omicron, the programmer has to define a slot whose value is the message selector and then refer to this slot in the message-send sentence. This more cumbersome alternative does not limit the expressability of the language. Expressability is not limited since it is always possible to define a slot holding the message selector instead of writing the selector explicitly in the sentence.

3.2.2 Objects as templates for object creation

There are two ways of creating an object; one is to define the object explicitly and the other is to clone another object. Both alternatives are available in Omicron.

In all the examples above, objects were explicitly defined. However, in Omicron an existing object may be used as a template, usually denoted prototype, for the creation of other objects. Creating an object from a prototype object is usually called cloning the prototype. When an object is cloned, the new object is equal to the prototype object, except that the new object gets its own unique name.

By functioning as prototype objects, Omicron objects also function as classes do in languages like Simula, C++ and Smalltalk in that they are templates for object creation. This limits the number of concepts but does not limit the expressability.

In several object-oriented languages it is possible to send references to templates for object creation as parameters in a message. This is possible in languages like, eg, Smalltalk and SELF. In Smalltalk classes are templates as well as objects, and references to the templates may therefore be passed as parameters. In SELF, as in Omicron, objects are templates for other objects and their names may therefore be passed as parameters.

3.2.3 Inheritance between objects: Extension objects

By using extension objects a system can be designed so that certain objects can be used to represent classes and extension objects represent subclasses. Extension objects are described in (Blair et al. 1991) as follows:

> *Extension objects : In class based systems, a subclass defines some specialised structure and behaviour and inherits default structure and behaviour from its superclasses. How are such elements shared in classless systems ? Similarly, an* extension object *can be created that can share with one or more original prototypes. These objects, declared as shared from the view point of the extension objects, act as surrogates or proxies to which the extension objects will turn for assistance. More specifically, extension objects do not just share behaviour; they can share knowledge contained in their prototypes in a more general way; that is, the value of state variables can be shared between objects.*

In (Blair et al. 1991) the behaviour is not seen as the sentences in the body of an object, but rather as the set of methods available to the object. Extension objects directly correspond to Omicron objects with inheritance. Omicron objects can inherit slots from each other and slots are used both as traditional variables *and* for holding methods. Note that in Omicron and most other object-oriented languages, only slots (variables and methods) are inherited, not sentences.

To specify inheritance a special kind of slot is introduced: an *inheritance slot*. In SELF an inheritance slot is specified by a trailing star (in SELF syntax: slotName*). This is adopted in Omicron using slotName*.

An example of a configuration of two objects with an inheritance slot:

```
p : ( [x→o, w→m, t→j], ) ||
e : ( [h* →p], x!w(t); x:=t; x := t clone; )
```

where h is the name of the inheritance slot. The object named e will then inherit all slots of the object named p. The object e therefore inherits the slots x, w and t and the sentences in e may refer to these slots in p.

In Omicron there may be more than one inheritance slot in an object and the value of the inheritance slots may be changed just like any other slot's value. An example of how to make an exact definition of inheritance between objects is given in section 3.4.

3.2.4 Objects and methods in one concept

In Omicron methods are objects which are extensions of other objects, ie, they inherit from the object they operate on.

In Omicron a method is not executed by sending it a message in order to evoke its response. The execution of a method is initiated by what corresponds to a runtime system, which selects a method. A method is selected by checking that the selector of the message matches a slot name in the receiving object. If a match is found, the object referenced in the matching slot will be executed giving the method's response.

Methods may have any number of parameters. For this purpose the Omicron language includes syntax for defining *input-slots*. Such input-slots are defined by starting the slot name with colon (:), eg, :slotName as in SELF.

An example showing how messages are sent and received:
We have three objects, a sender, a receiver and a method defined as follows:

```
sender      : ( [ x→receiver, w→m ], $x!w(x);) ||
receiver    : ( [ m→method, y→o, t→j],  ) ||
method      : ( [:s☆→nil ], y:=t; )
```

When the sentence in the sender is executed, a message is sent to the receiver. The action will be:

sender->receiver!m(receiver)/method-copy

> meaning that the receiver gets the message m with the object name 'receiver' as parameter and the object which is created as explained below is given the name 'method-copy'

When the message m is sent to the receiver, the underlying execution mechanism looks for a slot named m. In this case the receiver has an m-slot with value 'method'. To model the execution of a method the method object is then copied and a $ inserted at the beginning of the sentences in the method copy:

```
method-copy : ( [:s☆→receiver], $y:=t; )
```

If there were execution marks in the original method, these are removed so that there will only be one execution mark at the time in an object. In the method-copy the value of the input-slot named s has been updated to be equal to the parameter to the message, here 'receiver'. In addition to being an input-slot, the s-slot is an inheritance slot. This combination of input and inheritance slot models how a method executes on behalf of the object who received the message. Therefore the method is an *extension object* to the receiver.

The s-slot in the example method is used to model what is a pseudo-variable in most object-oriented languages, eg, the pseudo-variable 'self' in Smalltalk, the implicit 'self' in SELF and 'this' in C++ and Simula. The main difference between Omicron and the mentioned languages on the matter of the 'self/this'-variable is that slots used as self/this must be explicitly defined as a parameter in Omicron methods. This is done automatically in the other languages.

As the message-send-sentence in the sender-object is executed, the execution mark is moved to the right. The execution mark will then be at the end of the sequence of actions. Then the sender becomes a terminated object which has no more sentences to execute:

```
sender      : ( [ x→receiver, w→m ], x!w(x); $)
```

After the message has been sent and the method copied, the configuration of objects looks like this:

```
sender      : ( [ x→receiver, w→m ], x!w(x); $) ||
receiver    : ( [ m→method, y→o, t→j],  ) ||
method      : ( [:s☆→nil ], y:=t; ) ||
method-copy : ( [:s☆→receiver ], $y:=t; )
```

where the sender has terminated since there are no more sentences to execute and the method-copy is ready to execute its assignment sentence since this is preceded by $. Making method-copies accommodates recursion. This gives expression power which includes while-constructions.

As mentioned above, the method-copy executes on behalf of the receiver. This means that the method-copy has access to the slots in the receiver so that these slots may be read and set by sentences in the method-copy. This access is given by the definition of the s slot which is both an input and inheritance slot. Therefore, after execution of y:=t in the method-copy the configuration of objects will look like this:

```
sender      : ( [ x→receiver, w→m ], x!w(x); $) ||
receiver    : ( [ m→method, y→j, t→j],  ) ||
method      : ( [:s☆→nil ], y:=t; ) ||
method-copy : ( [:s☆→receiver ], y := t; $)
```

There are now two terminated objects and the slot y in the receiver has the value j.

Method-copies are extension objects
The method-copy objects are extensions to other objects in that they manipulate the other objects state. If the method-copies are not extension objects, there is no way to influence an object's state by sending it a message.

This is because sending a message can not result in the execution of sentences which change the values of the object since a non-extension method-copy can not inherit slots from the object and is therefore not allowed to access the object's slots.

Methods and executing methods are Omicron objects
As shown above, when objects are allowed to inherit from other objects and when object can have input-slots, it is not necessary to have different syntax for defining methods, executing methods and objects. Methods are modelled as extension objects with a particular inheritance slot. When some object receives a message resulting in the execution of a copy of a method, the executing method's particular inheritance slot will hold a value which is the name of the receiver of the message. This models the automatic self-binding found in most object-oriented languages.

3.2.5 Self reference

It is common for objects to send messages to themselves. In a programming language, the object itself is usually referred to by a pseudo variable, for instance in C++ it is named *this* and in Smalltalk *self*. When self reference to an object is done in a method of the object, the reference is done by using inheritance slots which are also parameters as shown above. However, this does not allow self references within an object, particularly when it is created from a clone action and it does not make it possible to refer to the executing method itself from within the method. To get such a feature in Omicron, the special name **this** is introduced. This special name **this** is an alias for the name of the object where **this** is found. **this** may be viewed as a relative object name representing the object name of the surrounding object. For example (note how this is a different use of self-reference than what happens in a method where there is automatic self-binding of the receiver of a message):

```
sender      : ( [ x→receiver, w→m ], $ x!w(x); ) ||
receiver    : ( [ m→method ],  ) ||
method      : ( [:s☆→nil, self→this, w→m, n→method ], self!n(self) )
```

Here a new object is first created and then sent the message m with the receiver's name as the first parameter. We then get the action:

receiver!m(receiver)/method-copy

When the method is executed a method copy is created and we get a situation as follows:

```
sender      : ( [ x→receiver, w→m ], x!w(x); $) ||
receiver    : ( [ m→method ],  ) ||
method      : ( [:s☆→nil, self→this, w→m ], self!w(self) ) ||
method-copy : ( [:s☆→receiver, self→this, w→m ], $ self!w(self) )
```

When the method copy is executed it will send itself the message m with its own name as parameter and we get the action:

method-copy!m(method-copy)/method-copy2

Since the object named method-copy will inherit the slot named m from the object named receiver, this action will result in the creation of a new method copy:

```
method-copy2  : ( [:s☆→method-copy, self→this, w→m ], self!w(self) )
```

When this new copy is executed it will send itself the message m with its own names as parameter - etc. etc.

3.2.6 Error actions

Errors actions occur when sentences can not be executed. For example, when a message-send-sentences is executed it will give an error action when the receiver of the message is not found in the system and when the message is not understood, ie, there is no method corresponding to the received message. Error actions also occur when one or more slots referred to in an executed sentence does not exist and when the prototype object referred to in an executed object creation sentences does not exist.

An error action is described:

```
e->error
```

where e is the name of the object which has reached a sentence which is not executable

When an error occurs, the execution mark is removed from the object where the non-executable sentence was found. This terminates the execution of the object.

It is possible to define more than one kind of error actions, where the error actions will somehow reflect what went wrong. Defining and using such error actions in the reasoning about reliable substitution might lead to different results than what is presented in this thesis. However, as discussed in chapters 5 and 10, it seems that other, not too exotic, error models give quite similar results as presented in this thesis. More exotic error models is left for further study.

3.2.7 Summary of Omicron's object-oriented concepts

The concepts in the Omicron-language reflect a system view where a system is a configuration of objects. Each object has a state and a unique name within the configuration which is independent of the object's state. An object's state is divided into two parts: a set of slots and a body. A slot can store a reference to an object, i.e. the name of an object. Each slot has a unique name within the object. The set of slots both function as what is traditionally known as the object's variables and as the object's method dictionary. An object's body consists of a list of sentences. When the object is executed these sentences are executed and the execution of a sentence results in actions. An object can do the following:

- store names in slots,
- test names in slots,
- make clones of objects and
- send messages to objects.

An object can send messages to the objects referenced in the object's slots or slots the object inherits from other objects. A message has a selector and may contain parameters. The parameters are names which may or may not be names of objects. An object responds to receiving a message by copying the object named in the slot with the same name as the message selector. The sentences in the copy's body is then executed. The copied object functions as what is traditionally called a method. Some of a method-object's slots may be input slots where the message parameters will be stored in the copy before the copy is executed. Also, some slots are inheritance slots defining object inheritance.

3.3 Defining Part of the Model-View Contract Using Omicron

This section shows how the model-view contract can be expressed using Omicron. It also discusses some expressions and syntactic sugar which may be added to Omicron in order to make it easier to express designs such as the model-view contract from chapter 2.

Using Omicron, the model's behaviour can be expressed as follows. Note that there are no execution mark as the model does not have any behaviour unless the context sends it some message:

```
model :  ( [state → v, views → o, setvalue → s, getvalue → g, attach → a, detach → d, notify → n ] ) ||
s :      ( [this☆ → model, :val → nil ] state := val; this!notify(); ) ||
g :      ( [this☆ → model, :return → nil] return!return(state); ) ||
n :      ( [this☆ → model] views!update(); ) ||
a :      ( [this☆ → model, :v → nil ] views!add(v); ) ||
d :      ( [this☆ → model, :v → nil ] views!remove(v); )
```

To make the model definition complete it is also necessary to define how object names are stored in the object named o. The object named o is the visible object of a component which can store object names. There are many ways to define such a component. They all include implementing functions to add and remove names from the set or sequence. In addition, the component needs to implement an update-method. To implement such functionality in Omicron it is necessary to describe a large number of objects and it would be rather difficult to convince oneself and others that this was a correct implementation of set/sequence. This complexity in defining a set/sequence component in Omicron is mainly due to the lack of set and sequence notations.

To simplify the description of model's behaviour we can include a simple sequence-notation in Omicron. We can let all slot values be sequences and use s := +v to add the value of v to the sequence in s, and use s := -v to remove the value of v from the sequence in s. We can also define the message-send action so that all objects named in the receiver sequence will receive the message. The model can then be defined as above, but where the a and d objects are replaced with the following objects:

```
a :      ( [this☆ → model, :v → nil ] views := +v; ) ||
d :      ( [this☆ → model, :v → nil ] views := -v; )
```

Then we do not need a separate component for storing object names. It is also easy to convince oneself that views actually is a sequence.

The above sequence notation has, as an experiment, been included in a version of Omicron not included in this thesis. In the experiment version of Omicron rules defining the semantics of the add and remove notation were added to the rules of action in section 3.4.3. The existing rules of action were also modified to incorporate the sequence notation. This gave more complex rules of actions and also more rules of action. It also gave more complex versions of the definitions and proofs in later parts of this thesis. Therefore it was left out in the presented version of Omicron. However, incorporating the sequence notation was not difficult and showing the propositions in later part of the thesis seemed to be straightforward. However, everything became more complex and detailed. Therefore, reading (and writing) definitions became more time-consuming and all formal definitions, propositions and proofs of the propositions became much more difficult to understand. It is therefore left for further work and/or for the particularly interested reader to include sequence notation to the formal definition of Omicron.

Other functions such as integer addition has also been tried incorporated into Omicron in order to simplify the expression of certain kinds of models. This was also straightforward to do, but also introduced details which distracted from the focus of the present work.

The views' behaviour can be defined as follows:

```
view :   ( [viewstate → nil, update → u, draw → d, subject → su, return → r ] ) ||
u :      ( [this☆ → view ] this!draw(); ) ||
d :      ( [this☆ → view ] subject!getvalue(this); ) ||
r :      ( [this☆ → view, :st → nil ] viewstate := st; ) ||
```

su : ([this* → view , :m → nil] subject := m;)

In chapter 2 the following invariant was presented (where *reflects* is some informal relation):

∀ v ∈ views • [v *reflects* subject.state]

and also the instantiation rule:

∀ v ∈ views • subject.attach(v) & v.subject(subject);

These do not express properties of object behaviour which the Omicron framework is meant to handle. Instead, Omicron is created for reasoning about similarity of the observable behaviour of components. In this case reasoning with Omicron would be used to find out if a refinement and the specification of model will have the same observable behaviour. If they have the same observable behaviour, both would send *update()* messages to the same objects. A refinement and its specification should send the same objects the *update()* message for each possible sequence of other messages sent to the model and view objects.

Note that there are no type information in Omicron, since this is linked to reliability requirements, which in Omicron is not supposed to be part of the language.

Also note how the return from the *getvalue()* message of the model is handled. A return is done as a message-send, and is therefore not a special kind of action.

Part of the context's non-deterministic behaviour can be described as is done below. For simplification, we here introduce a special notation to avoid having to define very many slots. In stead of defining a slot to hold a name to be used in a sentence, the value of the (not defined) slot can be explicitly stated in the sentences by writing a # symbol in front of it. This is a way of introducing name constants into the language. For instance sending the message *hithere()* to some object named in the rec-slot is written:

sender : ([rec->someobj] rec!#hithere();)

and we can also write:

sender : ([ ] #someobj!#hithere();)

which would give the same action as when executing the object above. This simplification means that the objects need only name the slots which hold variables and methods and inheritance slots.

p : ([v→nil, met1→m1, met2→m2, met3→m3, met4→m4]) ||
m1 : ([s*→p] $ v := #view **clone**; #p!#met1();) ||
m2 : ([s*→p] $ #model!#attach(v); #p!#met2();) ||
m3 : ([s*→p] $ #model!#detach(v); #p!#met3();) ||
m4: ([x → 123] $ #model!#setvalue(x); #p!#met4();)

This defines a context which will create new view objects and send the messages *attach()*, *detach()*, and *setvalue()* to the model in an arbitrary sequence infinitely many times. A refinement of this context may therefore create any number of views and send the three different messages in any sequence and send the messages any number of times.

3.4 Formal Definition of Omicron

This section gives a formal definition of the simple object-oriented language Omicron introduced in the previous sections.

Section 3.4.1 defines the syntax and formal semantics of the Omicron language. Section 3.4.2 gives formal definitions of inheritance between objects and present functions which are used in later sections and chapters.

The formal semantics is given operationally through a set of transition rules in section 3.4.3. Omicron actions are formally defined in this section through the definition of a transition relation of the form:

$$C \xrightarrow{\alpha} C'$$

Intuitively, this transition means that the configuration C can evolve into C', and in doing so perform the action α. To define this transition relation, many helpful functions and notations are defined in section 3.4.2. These include many finer details which are not necessary to comprehend in order to get the general ideas of Omicron.

Section 3.4.4 defines some basic notation for describing sequences of actions, derivations of configurations and the traces from execution of a configuration.

Section 3.4.5 shows some properties of Omicron configurations. It is also shown that a derived configuration is uniquely determined by the action, that actions from execution of the same sentence will be equal every time it is executed and no rules of action are applicable to a terminal configuration and at least one rule of action is applicable if the configuration is not terminal.

Section 3.4.6 includes notation which is used when describing configurations defined by combining other configurations. This is used in definitions and proof leading up to the formally stated substitution proposition.

3.4.1 Omicron syntax

This section describes the syntax of the Omicron language through the use of extended BNF (BNF is defined in appendix A). Terminal symbols are given in **bold font**.

Configuration	::=	$\text{Object}^{*}_{\|\|}$	
Object	::=	name : **(**Slots**,** Body**)**	-- Definition of an object
Slots	::=	**[** $\text{SlotDef}^{*}_{,}$ **]**	
Body	::=	$\text{Sentence}^{*}_{;}$ | $\text{Sentence}^{*}_{;}$ **$** $\text{Sentence}^{*}_{;}$	
Sentence	::=	name **:=** name **clone** |	-- Clone sentence
		$\text{name}^{+}_{,}$ **:= (**name = name name name **)** |	-- If-sentence
		name **!** name **(** $\text{name}^{*}_{,}$ **)**	-- Message-send sentence
SlotDef	::=	slotName → Val	
Val	::=	name | **this**	
slotName	::=	name |	-- Plain slot
		:name |	-- Input slot
		name☆ |	-- Inheritance slot
		:name☆	-- Input and Inheritance slot
name	::=	char^{+} but no colon (:) first and no ☆ last and not equal to 'this'	
char	::=	**a** | … | **z** | **A** | … | **Z** | **0** | … | **9** | **+** | **-** | ***** | **/** | **_** | **:** | =	

where $\text{Object}^{*}_{||}$ means $\text{Object}_1 \,||\, \ldots \,||\, \text{Object}_n$ and $\text{name}^{+}_{,}$ means one or more names separated by comma. The symbol $ is called the execution mark. Since objects may be executing in parallel, there may be more than one execution mark in the system, but only one in each object. The transition rules of section 3.2.3 define how execution marks are handled. The special word **this** is a special name which is an alias for the name of the object where **this** is found. **this** may be viewed as a relative object name representing the object name of the surrounding object.

Note that there may be three types of slots: plain slots, input slots and inheritance slots. A slot may be both an input slot and an inheritance slot. The plain slots have no other purpose than to store names. Input slots are slots for storing input names (parameters) when the object is executed as a result of some object receiving some message. The third type of slots are inheritance slots.

A *syntactically correct* configuration is well-formed if each object in the configuration has a unique name and the slot names within each object are unique.

A *system* is a configuration which is closed in that the objects in the system only collaborate with each other.

3.4.2 Formalisation of configurations

The set of all configurations is denoted $\mathscr{C}$. The set of all Slots is denoted $\mathscr{M}$ (also referred to as slot maps), the set of all names is denoted $\mathscr{N}$. The following syntactic conventions are used:

e,f,i,j,o,p	$\in \mathscr{N}$	- object names
k,l	$\in \mathscr{N}$	- names used as new object names
v,w,s,t,u	$\in \mathscr{N}$	- slot names
m	$\in \mathscr{N}$	- name used as message selector
$\bar{v}$	$\in \mathscr{N}^*$	- a list of input slot names
$\bar{p}$	$\in \mathscr{N}^*$	- a list of object names used as parameters (slot values)
A,B,C,D	$\in \mathscr{C}$	- well-formed configurations of objects
M	$\in \mathscr{M}$	- slot map, ie, the Slots part of an Object
S		- object bodies, ie, sequences of sentences and possibly $

A configuration may be seen as a mapping from object names to object definitions and Slots as a mapping from slot names to names. To simplify the definition of the formal semantics of Omicron this fact is used in that the configuration and slot definitions are expressed in a Map-type notation defined in appendix A. The following meta function defines the translation from Omicron syntax to Map-type notation:

Translation function signature:

[[Omicron Slot Map or Configuration]] == Omicron syntax in Map type notation

Translation function definition:

Slot Map translation:

[[$[s_1 \rightarrow t_1, \ldots, s_n \rightarrow t_n]$]] == $\text{init}()[s_1 \rightarrow t_1] \ldots [s_n \rightarrow t_n]$

Configuration translation:

[[$e_1 : (M_1, S_1) \parallel \ldots \parallel e_n : (M_n, S_n)$]] == $\text{init}()[e_1 \rightarrow ([[M_1]], S_1)] \ldots [e_n \rightarrow ([[M_n]], S_n)]$

In a mapping the right most pair is the most significant and it gives $\perp$ outside its domain. Note that $\perp$ is not a legal character in Omicron expressions.

In addition the following map notation is used:

C.Dom	= the domain of the configuration C, ie, the object names mapped from in C C == e : $(M_e, S_e) \parallel$ f : (M_f, S_f) gives C.Dom == {e, f}
C.Names	= all names found in C C == e : ([x→i],) || f : ([y→e, w→m], y!w()) gives C.Names == {e, x, i, f, y, w, m}
C.Values	= the set of values of the slots in configuration C C == e : ([:$x^{\star}$→i, :z→j], S_e) || f : ([$y^{\star}$→j, w→k], S_f) gives C.Values == {i, j, k}
C(o)	= the (Slot, Body)-part of the object o in the configuration C C == e : $(M_e, S_e) \parallel$ f : (M_f, S_f) gives C(e) == (M_e, S_e)
C(o).Slots	= the slot map of the object named o C == e : $(M_e, S_e) \parallel$ f : (M_f, S_f) gives C(e).Slots == M_e
C(o).Slots(s)	= the value of the slot s in the slot map of the object named o This is referred to as *getting the value* of the slot s in the object o. If o is not in C.Dom or s is not found in the slot map, the result is the object name $\perp$. C == e : ([x→i], S_e) || f : (M_f, S_f) gives C(e).Slots(x) == i and C(e).Slots(y) == $\perp$

C(o).inputs = the sequence of input slot names in the slot map of the object named o
C == e : ([:x$^{☆}$→i, y→j, :z→j], S_e) || f : (M_f, S_f) gives C(e).inputs = <x, z>
C(o).Body = the Body of the object named o
C == e : (M_e, S_e) || f : (M_f, S_f) gives C(e).Body == S_e

We also define notation for expressing that a name is in the domain of a configuration as follows:
o ∈ D == o ∈ D.Dom

A special definition of Map-lookup is needed to handle **this** as slot value. In appendix A maps are defined so that we have:

C(o).Slots(s) == if the value found in slot s is **this** then o is returned
else the value of the slot s is returned.

The following sequence notation is used:

<a_1, …, a_n>	a sequence of the n items a_1 to a_n
#	is a function whose value is the length of a sequence
sq1 & sq2	denote sequence sq1 concatenated with sequence sq2
sq_i	denote the i'th element of the sequence sq

The notation C(o).Slots(s) is extended to sequences of slot names by:

C(o).Slots(<s_1, …, s_n>) == < C(o).Slots(s_1), ..., C(o).Slots(s_n) >

The following function, denoted @, is used to test if the result of one or more slot lookups succeeds or not. It is defined as follows:

@⊥	== false
@v	== true if v ≠ ⊥
@<v_1,..,v_n>	== @v_1 ∧ ... ∧ @v_n

For example @(C(o).Slots(s)) returns true if o is an object name in C and s is a slot name in o's Slot map, false otherwise.

Updating a slot is defined as follows using the Map notation and the @-function:

C[p.s:=j] == *if* @(C(p).Slots(s)) *then* C[p→(C(p).Slots[s→j], C(p).Body)] *else* C.

If the slot s is found in the slot map of the object named p in C, then the s-slot is updated with the value j. This is denoted *setting the value* of the slot named s in the object named p to the value j. If s is not found in a slot map of p, then the result is no change to C.

The above functions are lifted to sequences of slot names as follows:

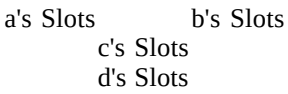
@C(o:<s_1,…,s_n>) == @C(o:s_1) ∧…∧ @C(o:s_n)
C[<o_1.s_1,…,o_n.s_n >:=j] == <C[o_1:s_1:=j],…,C[o_n:s_n:=j]>

Inheritance:

If an object has an inheritance slot it will inherit slots of the object referred to in the inheritance slot. To inherit slots means that sentences in the body of the object may use the names of the slots in the inherited slot map. An object may have several inheritance slots in its slot map, and also an inherited slot map may contain inheritance slots. One object's total slots is therefore all the slots found in the slot maps in the map graph defined by the values in the inheritance slots of the maps. This is referred to as an object's *inheritance graph*.

Example:
a: ([],) ||
b: ([],) ||
c: ([s1$^{☆}$ → a, s2$^{☆}$ → b, t→e],) ||
d: ([s$^{☆}$ → c, w→m], t!w()) ||
e: ([],)

The inheritance graph of the object named d is:



When looking up a given slot name in a Slot map the search is always started in the Slot map of the object where the slot name is referenced. E.g. for the slot t in d the lookup is started in d's Slot map. If the slot name is not found in the first Slot map the search is continued at the next level in the graph, ie, in c's map.

We let C(o).supers denote a function returning the sequence of the values of the inheritance slots in the slot map of the object named o. When

$C == o : ([:x^{☆} \rightarrow i, y^{☆} \rightarrow j, :z \rightarrow j], S_e) \parallel f : (M_f, S_f)$ then we have

$C(o).supers = \langle i, j \rangle$

An owner-function is used to find the owner of a particular slot in the inheritance graph of an object. If there is no owner in the inheritance structure, the owner function returns $\perp$.

In general there are many alternative ways to traverse an inheritance graph and some versions are discussed in (Chambers et al. 1991). The variations mostly stem from how Slot maps on the same level in a graph are traversed, how Slot maps which are included several times in a single graph are handled and how cyclic structures are tackled. In the example above owner(C, d, t) would return the c independently of which traversing algorithm is chosen. The owner-function definition found below uses a breadth first and leftmost first version. This definition of the owner-function will never terminate for cyclic inheritance structures where the slot is not found before a cycle is traversed once. In general it is possible to define owner-functions which will terminate for cyclic inheritance graphs.

owner(C,<>,s) == $\perp$

owner(C,<o>,s) == *if* o $\notin$ C.Dom *then* $\perp$ *else*
if @C(o).Slots(s) *then* o *else* owner(C, C(o).supers, s)

owner(C,<o_1, ..., o_n>,s) == *let* x = owner(C, <o_1>, s) *in*
if x = $\perp$ *then* owner(C, <o_2, ..., o_n>, s) *else* x

This definition of the owner-function says that owner(C,<o>,s) returns the name of the object in the inheritance graph of o where s is found or $\perp$ when s is not defined anywhere in the inheritance graph of o. When the list of object names only contain a single name we write owner(C, o, s) instead of owner(C, <o>, s).

In Omicron the inheritance graph of an object may be changed during execution since inheritance slots may get new values just like all other slots. An inheritance slot may also be an input-slot which then is defined by the syntax: :slotName$^{☆}$.

Next, we introduce some short-hand notations where o:s is used to denote a slot s in an object named o or in an object in the inheritance graph of o:

C(o:s) == C(owner(C, o, s)).Slots(s)

We say that when @C(o:s) is true then *the slot s has an owner*. This functions is lifted to sequences of slot names as follows:

C(o:<s_1,…,s_n>) == <C(o:s_1),…,C(o:s_n)>

Below the notation $\overline{o.s}$ is used to denote $o_1.s_1$, ..., $o_n.s_n$.

3.4.3 Formal operational semantics of sentences

The formal semantics of Omicron sentences is defined by a set of transition rules creating an operational semantic definition following the tradition started by (Plotkin 1981) and also used in describing the semantics of the π-language, eg, in (Milner et al. 1989b).

The syntax of the transitions is given below together with an informal description of the semantics. The operational semantics is defined through a transition relation where a transition rule is given for each type of Omicron sentence.

Definition: Transition
A transition is of the form:

$$C \xrightarrow{\alpha} C'$$

Intuitively, this transition means that the configuration C can evolve into C', and in doing so performs the action α. The set of all such actions is denoted $\mathscr{A}$. In Omicron there are four types of actions a configuration can perform. The actions are described formally through the transition system below.

The transitions may be applied to a configuration in any order, as long as the transition is legal by the premises in the rules of action below. The rules of action are not confluent, ie, the result configuration after applying a set of transitions to a configuration may depend on the order in which the transitions are applied. This is in correspondence with the intuition that the result of executing a parallel program depends on the execution order.

By definition || is ACI (associative, commutative and has the empty configuration as identity), ie, $C_1||C_2$ is equal to $C_2||C_1$ for any configurations C_1 and C_2.

Definition: Transition relation and rules of action
The transition relation, denoted $\xrightarrow{\alpha}$, is the smallest relation between Omicron configuration expressions satisfying the *rules of action* given below. All the names in the rules are meta-variables. Informal descriptions of the rules are given below and the rules are given in the frame on the following page.

The rules are given by transition for the three different kinds of sentences which may be found after the execution mark $: if-sentence ($\bar{s}$:=(v=w t u)), clone sentence (s := t clone) and message-send sentence (s!w($\bar{t}$)).

Comments to the rules and actions:
The system denoted C in the rules has the form C == C'||e:(M,$S_1$$sentence;S_2) where S_1 and S_2 are sequences of sentences, possibly empty.

The difference between the original and derived configurations common in all rules, except the error rule, is the movement of the execution mark. The rules do not imply much difference in the initial and derived version of the part of the configuration denoted C'. The only rules which might affect C' are the CLONE and IF rules which update slots. The CLONE and SEND rules create new objects, something which does not affect C'.

All rules except the error rule have relatively weak requirements, requiring that the slots referred to in the executed sentence have owners. In addition the rules make requirements on the values of the different slots. These requirements depend on the kind of sentence they apply to. This is further discussed below.

The IF-rules:
The IF-rules are applicable when there is an execution mark in front of an if-sentence and all the slots have owners. The IF-true rule is applicable if the slots referred to as v and w in the rule have equal values and the IF-false rule is applicable if the values are different.

Executing an if-sentence gives an *assignment* action of the form e->o_1.s_1,…,o_n.s_n := i, meaning that this action stems from execution of a sentence in an object named e and the slots s_1,…,s_n in the respective objects o_1,…,o_n are updated with the value i. The set of all assignment actions is denoted $\mathscr{A}_{:=}$.

The CLONE-rule:
The rule is applicable when there is an execution mark in front of a clone sentence, all the slots have owners and the slot referred to as t in the rule has a value which is the name of an object in the configuration. The execution of a clone sentence gives a *clone* action of the form e->o.s := k / j, which means that the action stem from execution of a sentence in an object named e, the object j is copied and the copy is given a new name k and the slot s in the object o gets the value k. The set of all clone actions is denoted $\mathscr{A}_{clone}$.

The SEND-rule:
Execution of a message-send sentence gives a *message-send* action of the form e->o!m(i_1,…,i_n)/k, meaning that the actions stem from execution of a sentence in an object named e and o gets the message m(i_1,…,i_n) and k is the name of a new object created as result of the message send. The set of all message-send actions is denoted $\mathscr{A}_!$.

A message-send action creates a new object and the new object will have an execution mark. Also, no object or execution mark is removed from the existing configuration. The new object will therefore execute in parallel with all objects which executed in the initial configuration C.

In an action e->o!m($\bar{p}$)/k, o is called the *receiver* of the message, m is called the *selector* of the message and $\bar{p}$ is a list of names called the *parameters* of the message. The object named j in the SEND rule denoting C(o:m) is called a *method.*

The rule is applicable when there is an execution mark in front of a message-send sentence, all the slots have owners and both a receiver object and a method object is found. It is also required that the m-slot of the receiver (o) holds the name of an object having the same number of input slots as there are parameters in the message.

The rule models the reception of a message as follows:
The method is copied, resulting in a *method copy* which is given a new name (k) and its input slots (C[k].inputs = $\bar{v}$) are updated with the message's parameters $\bar{p}$ (ie, ($\bar{v} \rightarrow \bar{p}$)). Also, any old execution marks are removed and a new execution mark, $, is inserted into the body of the method-copy.

Note that there are never more than one $ in any object body since
- when an object is copied then $ is removed from the body before $ inserted in the beginning and
- the SEND rule is the only rule that inserts execution marks.

IF-true rule:

$$\frac{@C(e{:}\bar{s}\ \&< v, w, t >)\ \wedge\ C(e{:}v)\ =\ C(e{:}w)}{C'\,||\,e{:}(M, S_1\$\bar{s}{:=}(v = w\ t\ f); S_2) \xrightarrow{e\text{->}\overline{o.s}:=j} (C'\,||\,e{:}(M, S_1; \bar{s}{:=}(v = w\ t\ f)\$S_2))[\overline{o.s}{:=}\ j]}$$

where j=C(e:t) and o_i = owner(C,e,s_i) for each $o_i.s_i$ in $\overline{o.s}$

IF-false rule:

$$\frac{@C(e{:}\bar{s}\ \&< v, w, f >)\ \wedge\ C(e{:}v)\ \neq\ C(e{:}w)}{C'\,||\,e{:}(M, S_1\$\bar{s}{:=}(v = w\ t\ f); S_2) \xrightarrow{e\text{->}\overline{o.s}:=j} (C'\,||\,e{:}(M, S_1; \bar{s}{:=}(v = w\ t\ f)\$S_2))[\overline{o.s}{:=}\ j]}$$

where j=C(e:f) and o_i = owner(C,e,s_i) for all $o_i.s_i$ in $\overline{o.s}$

CLONE rule:

$$\frac{@C(e{:}s)\ \wedge\ C(e{:}t)\ \in\ C}{C'\,||\,e{:}(M, S_1\$s{:=}\ t\ \text{clone}; S_2) \xrightarrow{e\text{->}o.s:=k/j} (C'\,||\,e{:}(M, S_1; s{:=}\ t\ \text{clone}\$S_2)\,||\,k{:}C(j))[o.s{:=}\ k]}$$

where j = C(e:t), o = owner(C,e,s) and k is any name such that k $\notin$ C.Names

SEND rule:

$$\frac{@C(e{:}\bar{t})\ \wedge\ C(e{:}s)\ \in\ C\ \wedge\ C(o{:}m)\ \in\ C\ \wedge\ \#C(j).\text{inputs} = \#\bar{t}}{C'\,||\,e{:}(M, S_1\$s!w(\bar{t}); S_2) \xrightarrow{e\text{->}o!m(\bar{p})/k} C'\,||\,e{:}(M, S_1; s!w(\bar{t})\$S_2)\,||\,k{:}(C(j).\text{Slots}[\bar{v} \rightarrow \bar{p}],\ \$C(j).\text{Body}^{-\$})}$$

where o = C(e:s), m = C(e:w), j = C(o:m), $\bar{v}$ = C(j).inputs, $\bar{p}$ = C(e:$\bar{t}$),
k is any name such that k $\notin$ C.Names and
C(j).Body$^{-\$}$ means that any existing $ in the body is removed

ERROR rule:

$$\frac{\text{no other rule of action is applicable to the e - object}}{C'\,||\,e{:}(M, S_1\$\text{sentence}; S_2) \xrightarrow{e\text{->error}} C'\,||\,e{:}(M, S_1; \text{sentence}; S_2)}$$

The rules of action

The ERROR-rule:
The ERROR rule is applicable when there is an execution mark in front of a sentence and no other rules apply to this sentence. One reason for an error is that there are slots which do not have owners. Another reason is that there are slots which should have values which are names of objects in the configuration, but which are not. This would create erroneous actions such as messages to non-existent object or cloning of non-existent object.

Error actions have the form e->error where e denote the name of an object with an execution mark but where none of the rules of action apply. The set of all error actions is denoted $\mathscr{A}_{error}$.

The error rule says that the execution mark is removed from the object holding the erroneous sentence. This reflect a system-view where an object with an erroneous sentence terminates. Each object is seen as executing separately from the rest of the system in that only the object with the erroneous sentence terminates and the rest of the system continues execution.

The error rule with the informal requirement "no other rule of action is applicable to the e-object" can be replaced by a set of rules with formal requirements. The rules in the set replacing the informally expressed error rule would be equal to the IF, CLONE and SEND rules except that the requirements would be negated and the actions would be error actions instead of assignment, clone and message-send actions.

The rules of action preserve syntactic correctness

The following proposition shows that the rules of action preserve syntactic correctness. This is an important property since it ensures that execution of a system does not give syntactically incorrect expressions.

Proposition P.3.1 : The rules of action preserve syntactic correctness
Applying a rule of action to a syntactic correct configuration gives a syntactic correct configuration.

Proof:
Done by cases for the different rules:
IF-rules : The new version of the object named e has legal syntax since only the execution mark is moved and a slot gets a new value which is a name.
CLONE rule : The new version of the object named e has legal syntax since only the execution mark is moved and a slot gets a new value which is a name. The new clone has legal syntax since the original is assumed to have legal syntax and the new name is unique within the configuration.
SEND rule : The new version of the object named e has legal syntax since only the execution mark is moved. The new method-copy has legal syntax since it is assumed that the method was syntactically correct and any existing execution mark is removed before an execution mark is placed in the beginning of the body and the input slots get new values which are names. Also, the method copy gets a new name which is unique within the configuration.
ERROR rule : The new version of the object named e has legal syntax since only the execution mark is removed.
❑

3.4.4 Basic notations and definitions

This section gives some basic definitions and notations which are used when expressing properties of and reasoning about Omicron configurations.

The first definition formally defines the notation previously used for sequences of action.

Definition: Sequences of transitions and actions: $\xrightarrow{\bar{\alpha}}$, $\bar{\alpha}$,

$C \xrightarrow{\bar{\alpha}} C'$ denotes a sequence of zero or more transitions $\xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n}$ from C to C' as defined by the rules of action and $\alpha_i \in \mathscr{A}$.

The next definition defines the term "derivations of a configuration". The derivations of a configuration C are all configurations which can be the result of executing zero, one or more sentences in C.

Definition: Derivation of a configuration

The configurations derived by one or more transitions from a configuration C, are denoted the derivations of C.

$$\text{Derivations}(C) == \{ C' \mid \exists\ \bar{\alpha} : C \xrightarrow{\bar{\alpha}} C' \}$$

For this parallel version of Omicron the convention of interleaving semantics is followed. This means that the execution of different objects (or machines) are *interleaved*, ie, the effect of (a pair of machines) executing one operation each simultaneously is the same as executing the two operations in some arbitrary order.

Also, each action is seen as atomic. Therefore the result of executing a parallel Omicron system may be correctly described by a sequence of (atomic) actions. All possible executions of a parallel system can be described by a set of sequences of actions. We call this set the *traces* of the configuration. The traces can be formally defined as follows:

Definition: The traces of a configuration

The traces of a configuration C is the set of all sequences of zero or more transitions $\xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n}$ from C to some C' as defined by the rules of action and $\alpha_i \in \mathscr{A}$.

$$\text{Traces}(C) == \{\ \bar{\alpha} \mid \exists\, C' :\ C \xrightarrow{\bar{\alpha}} C'\ \}$$

We let $\alpha \in \text{Traces}(C)$ abbreviate $< \alpha > \in \text{Traces}(C)$.

In the above definition, the traces of a system is defined as the prefix closed set of sequences of actions from executions of the system. This means that each sequence of actions in a trace contains zero or more actions. Each sequence describes the actions resulting from execution of the system from the start of "execution time" until some arbitrary later "time" in the execution of the system.

Definition: Description and execution parts of actions

In the following, actions are denoted by small Greek letters, typically α and β, and an action is seen as consisting of two parts:

$\alpha = \alpha.\text{exe} \rightarrow \alpha.\text{dcs}$
where $\alpha.\text{exe}$ is the name of the object where the executed sentence which gave the action was found and
$\alpha.\text{dsc}$ is the rest of the action - the description part of the action.

If $\alpha.\text{exe} \in C$ then C is said to be the *owner* of the executed sentence. It is also said that α is from execution of a sentence in C, or just α from C.

We let $o \in \alpha.\text{dsc}$ mean that o is a name found in the description part of the action α, and
$<o_1, \ldots, o_n> = \alpha.\text{names}$
means that the names $o_1, \ldots, o_n$ are the names found in α in the sequence they occur in $\alpha.\text{dsc}$.

3.4.5 Some properties of configurations of objects

The two next observations state properties which relate actions and derived configurations. They state different aspect of the fact that equal actions give equal derived configurations.

Observation O.3.1 : A derived configuration is uniquely determined by the action

When two transitions from the same configuration are equal then the result configurations are equal, ie, the resulting configuration is uniquely determined by the action. This can be formally stated:

$$C \xrightarrow{\alpha} C' \wedge C \xrightarrow{\alpha} C'' \Rightarrow C' \equiv C''$$

Observation O.3.2 : Each object is deterministic and gives equal derived configurations

When the exe-part of two actions from the same configuration are equal then they are actions from execution of the same sentence. This is because there is only one sentence with an execution mark in front of it in each object and therefore at most one rule of action is applicable to any object. Then the actions are equal and the transitions are equal. This can formally be stated:

$$C \xrightarrow{\alpha} C' \wedge C \xrightarrow{\beta} C'' \wedge \alpha.\text{exe} = \beta.\text{exe} \Rightarrow \alpha \equiv \beta \wedge C' \equiv C''$$

This property expresses that each object has a deterministic behaviour.

Next we show that no rules of action are applicable to a configuration which has terminated, and at least one rule of action is applicable if the configuration has an execution mark in front of a sentence. This shows that the rules of action may be viewed as describing execution of Omicron systems in an appropriate way. First we formally define termination by giving a definition of terminal configurations.

Definition: Terminal configurations

A configuration is terminal if all object bodies either have an execution mark at the end of its sentences or have no execution mark. This means that there are no object bodies of the form S_1 \$ S_2 where S_2 is non-empty. The set of all terminal configurations is denoted $\mathscr{C}_{Term}$.

Proposition P.3.2 A configuration is terminal iff no rules of action are applicable

$C \in \mathscr{C}_{Term} \Leftrightarrow Traces(C) = \{<>\}$

Proofs:

Proof of $\Rightarrow$: When all objects in C either have the form e: (M, S \$) or when \$ is not in the body of the object, then there are no actions from execution of sentences in the configuration and then the traces of the configuration only include the empty action sequence <> and the proposition holds.

Proof of $\Leftarrow$:

We show that when there are non-empty action sequences in Traces(C) then C is not a terminal configuration.

When $C \notin \mathscr{C}_{Term}$ then there exists one or more objects with an execution marks in front of a list of sentences. It must therefore have one of the following forms and then a rule of action is applicable:

Alternative	*Applicable rules*
(M, S_1 \$ s := t clone; S_2)	The CLONE rule or the ERROR rule is applicable.
(M, S_1 \$ $s_1 \ldots s_n$:= (v=w t u); S_2)	One of the IF rules or the ERROR rule is applicable.
(M, S_1 \$ s!w($t_1 \ldots t_n$); S_2)	The SEND rule or the ERROR rule is applicable.

❑

3.4.6 Combined configurations

This subsection presents definitions which are used to express and prove the substitution proposition. In the substitution proposition there are transitions from composed configurations, say B||D $\xrightarrow{\beta}$ B'||D'. The derived configuration is also seen as being a composition of configurations, here B'||D' where B' and D' are called the primed versions of B and D respectively.

To simplify the expression of derivations of composed configurations, we define how the prime configurations are derived from some initial composed configurations and a related action.

When some action from B||D creates a new object, the definition of primed configurations specifies whether the new object is to be found in B' or D'. For example we can have a configuration as follows:

B = b : ([s->nil, t->o], \$s := t clone;)

D = o : ([m->p],)

After an action β = b->b.s := k/o from execution of the sentence in B in B||D the derived configuration will be:

b : ([s->k, t->o], s := t clone; \$) || o : ([m->p],) || k : ([m->p],)

where a new object, in this case named k, is added to the configuration. A question is then if the new object is considered part of D' or part of B'. In general, if we have some action $\beta \in$ action(B||D) and

β.dsc = i.s := k/o or β.dsc = o!m($\bar{p}$)/k

the question is whether $k \in D'$ or $k \in B'$. We choose the following convention which models the ideas described in chapter 2:

i.s := k/o : $o \in D \Rightarrow k \in D'$
o!m($\bar{p}$)/k : $o \in D \Rightarrow k \in D'$

which says that

if an object is a clone of an object in D, then the object is part of D', and
if the object is a method copy created as result of some object in D getting a message,
then the new object is part of D'.

In all other respects, D' is equal to D except that it is updated according to the rules of action. This means that if β is from execution of a sentence in an object in D, ie β.exe $\in$ D, the corresponding object in D' has the

execution mark one sentence to the right. Also, if the action updates slots in D, the corresponding slots in D' have the new values.

Correspondingly, this is also the case for objects in B and the objects found in B'.

Below, the prime of a configuration is formally defined. In this definition, and also in definitions in later chapters, the following function is used:

Definition: The NewNames function

Given an action sequence $\bar{\alpha}$ and a set of object names O. We define the new names function to return the set of names of all objects created by the actions in $\bar{\alpha}$ and which are

- created by cloning objects in O, or created by cloning clones of objects in O, or cloning clones of clones of objects in O etc. and
- method copies which are created when the receiver is an object in O or a clone of an object in O, or a clone of a clone of an object in O etc.

The function is formally defined as follows:

NewNames(<>, O) == Ø

NewNames(<α> & $\bar{\alpha}$, O) ==

case α.dsc *of*
i.s:=k/o, o!m($\bar{p}$)/k : *if* o ∈ O *then* NewNames($\bar{\alpha}$, O∪{k}) ∪ {k}
else NewNames($\bar{\alpha}$, O)
otherwise : NewNames($\bar{\alpha}$, O)

We let NewNames($\bar{\alpha}$, D) abbreviate NewNames($\bar{\alpha}$, D.Dom)

Definition: The prime of a configuration

The prime of D, denoted D', is defined relative to a configuration B and an action sequence $\bar{\beta}$ ∈ Traces(B||D). We define it as follows:

$$\text{prime}(D, B, \bar{\beta}) == D' \Leftrightarrow B||D \xrightarrow{\bar{\beta}} B'||D' \wedge D'.Dom = (D.Dom \cup \text{NewNames}(\bar{\beta}, D))$$

This notation is used in the rest of this thesis in order to denote derivations of the different parts of a configuration.

If the object names in a configuration are not unique, then the configuration is not well formed as defined above. Therefore, when composing configurations, the names of the objects in the configurations must not overlap. We say that configurations with non-overlapping domains are combinable and define this formally as follows:

Definition: Combinable configurations

Given a set of configurations $C_1,\ldots, C_n$, The configurations are combinable if they have non-overlapping domains, ie, $\forall$ i,j ∈ {1…n} • i ≠ j ⇒ C_i.Dom ∩ C_j.Dom = Ø. This ensures that the combined configuration $C_1||\ldots||C_n$ is well-formed and we have:

$C_1||\ldots||C_n \in \mathscr{C}$

When configurations are compared, it is usually expected that objects in one configuration, say, D can be able to send messages to or clone objects in another configuration, say, B or vice versa. To be able to have such actions it is necessary that slots in D hold values which are names of objects in B, and vice versa. We say that the B object names which are found as D slot values are visible to D. As the system of composed configurations is executed, B can send messages to D with B object names as parameters and D can clone B objects. In this way, new B object names can become visible to D. Visible object names are formally defined as follows:

Definition: Visible object names

The set of visible object names of a configuration B relative to a configuration D are those object names in B or derivations of B which, at some point during an execution from B||D, will occur as slot values in D or some derivation of D:

$$\text{Visible}(B, D) == \{ o \mid \exists B', D', \bar{\beta} : B||D \xrightarrow{\bar{\beta}} B'||D' \wedge o \in B'.Dom \wedge o \in D'.Values \}$$

We say that an object with name o in a configuration B or derivations of B is *visible* to an other configuration D if $o \in$ Visible(B, D). We also say that o is one of the visible objects of B and we say that o is visible *from* B *to* D.

3.5 Alternative Semantic Descriptions

This subsection presents some alternative to formally define Omicron's semantics along the lines of Plotkin's work (Plotkin 1981). However, before presenting the alternatives, a comment is made about why the rules of action refer to a complete system and not parts of such systems.

External and internal actions

In most definitions of language semantics using the Plotkin-style, the rules of action are usually organised as follows:

Typically there are three and three rules grouped together. In a group of three there are two rules which may be applied to parts of a system and therefore define what is perceived as internal actions in the part. For example there is one rule for receiving a message and one for sending a message - schematically written:

requirement 1	requirement 2
A -x-> A'	B -y->B'

which says that when requirement 1 holds then a legal transition will be A -x-> A' where x denote some internal action. Similar for the other schematic rule. The third rule then defines an external action which occur when the two parts are combined:

requirement 3

A||B -z-> A"||B"

where requirement 3 is typically that the transitions A -x-> A' and B -y->B' are legal. Then A" and B" are defined based on A' and B'. z defines the external action.

This approach was also originally taken when defining the semantics of Omicron. For instance there was a rule quite similar to the SEND-rule which defined an internal action with an "unsent message" and where the rule stopped the execution of the object which sent the message. Then another rule said that if there was some object in another component which was willing to receive such a message, then the execution will continue and the result would be an "external message-send action". The message-send action in the first rule would then be seen as an internal error action if there were no willing receiver. There would also be a rule defining internal message-send actions where the sender is allowed to continue execution. There would also be rules describing reception of messages. One would define the reception of a message where the receiver and method is found and the number of input slots correspond with the number of parameters in the message. This rule would not say that the method copy will start executing. Based on this example, the SEND-rule above would then be replaced by at least 4 transition rules.

Other complexities in the modular rules of actions were the result of inheritance between objects. When there are inheritance, the receiver might not be possible to describe by only looking at a part of the system since this part might not include the slot references in the message-send sentence. Then the possible transitions from some configurations would include actions with all possible objects as receivers. Then there would be rules such as "If A can send a message to any object, and the slot name in the executed sentence in A which gave this action is found in a configuration which A is combined with, then the only legal transition of the combined configuration will be a message to the object named in the slot". This would correspond to a requirement 3 as shown above and in this case the requirement 3 refer to the slot names in A which are involved in defining the x-transition. This became quite similar to stating requirement 1 as part of requirement 3, and in this way most of the requirements in rules of type 1 and 2 above were repeated in the requirements in rules of type 3. This gave substantially more complex requirements of type 3 then the requirements in the above present rules of actions.

The result of organising the rules of action in this more modular manner was, as the small examples show, more rules of actions and more complex requirements in the rules. This way of organising the transition rules was therefore not followed. In addition, the definition of internal and external actions which usually helps in simplifying definitions, propositions and proofs were not a similar help in relation to defining and showing reliable substitution. This might be because the formalism is in this thesis used in a situation where the focus is on sets of components forming complete systems, and not as in the more common situation where the focus is on single components which are to be placed in some system together with unspecified components.

Denotational semantics

When defining Omicron, one attempt was made at using denotational semantics to define what the language expressions mean. This was rather simple for the sequential version of Omicron. However, for the parallel version of Omicron it involved using power sets of states. The introduction of power sets adds complexity to the definitions and to propositions and proofs. Also, denotational semantics are better suited for expressing objects' state and reasoning about such states, rather than expressing and reasoning about object behaviour.

Temporal logic

In the initial stages of the work presented in this theses, some attempts were made at using temporal logic to express and reason about object behaviour. However, when Plotkin style operational definition of languages' semantics were tried, the Plotkin style gave much simpler expressions and proofs. It also allowed a more direct and intuitive relationship between objects as described in object-oriented languages and objects in the formal model.

Other alternatives

Other formal basis such as, evolving algebras and dynamic algebras, could be used. However, this would mean that the concepts in the language must be translated to concepts used in relation to evolving/dynamic algebras. Also, reasoning would have to be done based on the algebras.

Conclusion

The advantage of using Plotkin style formalisation of Omicron, is that it is a nice way to get a direct and intuitive model of the concepts of the language. It is not necessary to rewrite or translate from objects, methods, messages, object creation etc. to something else. Also, it is not necessary to introduce any concepts which are not modelled in the language. In addition, reasoning can be done directly based on the concepts in the language.

If one of the other alternatives were used, it would give a less direct and intuitive relationships between objects and similarity of objects as found in object component systems and in objects and similarity as found in the present formal model.

CHAPTER 4

Observable behaviour and Refinement relations

Previous chapters have just used the term "similar observable behaviour" informally. This section will elaborate on this topic and give a formal definition of observable behaviour and of observably equal action sequences. This chapter also defines similarity of components through a refinement relation between components.

This chapter argues that the defined refinement relation is in line with the intuitive understanding of similar components as presented in chapter 2. In chapter 5 it is shown that the defined refinement relation does not give reliable substitution. How the definition must be strengthened in order to get reliable substitution is the topic of the next chapter, chapter 5.

In the previous chapter, Omicron actions were defined as follows:

e->o!m(j_1,…,j_n)/k	a *message-send action* from a sentence in the object named e: o gets the message m with j_1,…,j_n as a parameters and where k is the name of the new method object to be executed
e->o_1.s_1,…,o_n.s_n:=j	an *assignment action* from a sentence in the object named e: for i = 1..n, the slot s_i in the object o_i gets the value j
e->i.s:=k/o	a *clone action* from a sentence in the object named e: the object o is copied and given a new name k and the slot s in the object i gets the value k
e->error	an *error action* from execution of a sentence in the object named e.

In the following, actions are denoted by small Greek letters, typically α and β, and an action is seen as consisting of two parts:

$\alpha = \alpha.exe \rightarrow \alpha.dcs$
where α.exe is the name of the object where the executed sentence giving the action was found and
α.dsc is the rest of the action.

Section 4.1 defines observable and hidden actions.

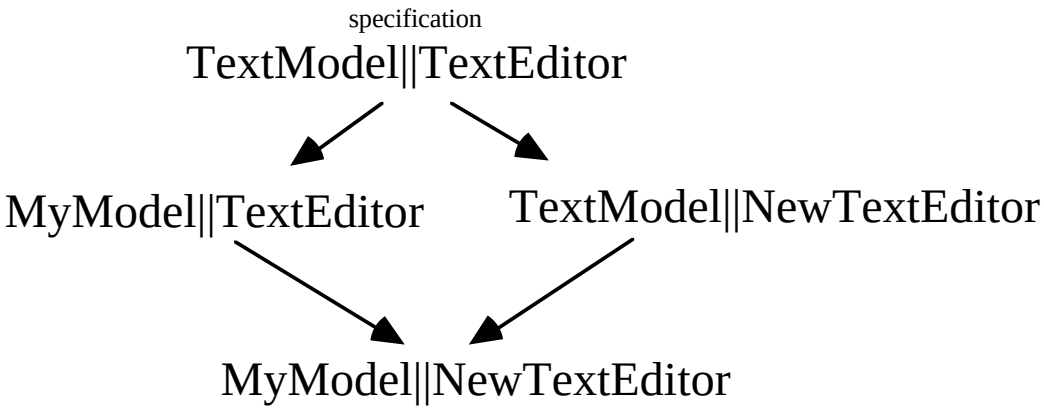
Section 4.2 defines observable equality actions.

Section 4.3 defines a refinement relation between configurations.

4.1 Observable and Hidden Actions

Actions stemming from the execution of a sentence in a component are either hidden or observed. These aspects are defined relative to the set of observers, where the observers are the objects making up the context of the component. This section gives formal definition of observable and hidden actions.

The definitions might be easier to understand if the text editor example of chapter 2 is kept in mind. This example can be depicted as follows:



The two refinements combined

The below definitions may be viewed as focusing on MyModel as a component and TextEditor as the observing context.

Observable actions are defined relative to a context consisting of one or more objects. The definition of observability of actions may therefore be viewed as focusing on the observability of the actions from MyModel||TextEditor relative to the observers in TextEditor. These actions may stem from execution of sentences in any part of the system. If MyModel is to be a refinement of TextModel, it is these observable actions of MyModel||TextEditor which must be similar to the observable actions in TextModel||TextEditor relative to TextEditor. Similarity of observed actions is defined in section 4.2.

Below the configuration denoted D, is used to represent an observing context of objects. This would correspond to TextEditor in the above example. The configuration denoted A||D can be exemplified by MyModel||TextEditor and the action α and the action sequence $\overline{\alpha}$ can be thought of as being actions from execution of MyModel||TextEditor. A set of object names O is used to denote the set of observing objects in a context, where the set O corresponds to the names of the objects found in TextEditor. Below we say that an object is named in O if the name of the object is an element in the set O.

4.1.1 Observable actions and action sequences

Observable actions

Section 2.3 motivated a definition of observable actions and gave an informal definition. Below a formal definition of observable Omicron actions is given based on the definitions in section 2.3.

In the definition of observable actions, actions stemming from execution of sentences in the context and actions stemming from execution of sentences in the component are treated equally with respect to observability. Observable actions can therefore stem from execution of sentences in any part of a system. An observing context will therefore observe actions from execution of sentences in both the component and the context itself.

Definition: Observable Action

Given an action α and a set of object names O. The action α is observable from the objects named in O if the action changes slots in an object named in O, clones an object named in O, is a message send to an object named in O or an error action from an object named in O. The set of actions observable from the objects named in O is denoted obs(O) and is formally defined[7]:

$$
\begin{array}{lll}
obs(O) == & \{\ e\text{->}o!m(\bar{p})/k & \mid o \in O\ \} \cup \\
 & \{\ e\text{->}o_1.s_1,\ldots,o_n.s_n := j & \mid \exists\ i \in \{1\ldots n\} : o_i \in O\ \} \cup \\
 & \{\ e\text{->}o.s := k/j & \mid o \in O \vee j \in O\ \} \cup \\
 & \{\ e\text{->}error & \mid e \in O\ \}
\end{array}
$$

Given a configuration C, then obs(C) abbreviates obs(C.Dom). We say that "α is observable from C" if $\alpha \in$ obs(C). Similarly, we say "α is not observable from C" if $\alpha \notin$ obs(C). Note that C will have observable actions $\alpha \in$ obs(C) where α.exe $\in$ C, while there may also be actions where α.exe $\in$ C such that $\alpha \notin$ obs(C). This is in line with the informal definition of section 2.3.

By definition, observable actions will always be observable from some object. An object is always a part of one and only one component and, therefore, an action will always be observable from some component. This also means that an action can not be unobserved from all components in a system. Message-send actions and error actions will be observable to one object and will therefore only be observable from one component. Assignment actions can be observable to n objects where n is between 1 and m and where m is the number of slots assigned to by the action. Such actions can therefore be observable from both the component itself and to the objects in the context making up the observers of the component. Object creation actions is observable from one or two components. It is observable from one component if the updated slot and the copied template object are found in the same component. It is observable from two components if the object with the updated slot and the copied template object are in two different components. Object creation actions can therefore also be observable from both the component and the context of the component.

Below the observable trace of a sequence of actions is defined. The observable trace is defined relative to a set of object names O. The defined function will return all the actions in the actions sequence $\bar{\alpha}$ observable from objects named in the set O.

Definition: Observable trace of a sequence of actions

Given a sequence of actions $\bar{\alpha}$ and a set of object names O. The observable trace of the action sequence $\bar{\alpha}$ is the sequence of actions observable from the objects named in O. This is denoted $\bar{\alpha}$/obs(O).

The notation $\bar{\alpha}$/Obs(O) is taken from (Dahl 1992) and denote all O-observed actions in the action sequence $\bar{\alpha}$ in the same order as they are found in $\bar{\alpha}$.

Hidden behaviour and silent actions

The messages sent from an object in a component to another object in the same component are not part of the observable behaviour of the component. They are only part of the component's operational specification, the implementation of a component. Such actions are called hidden actions. In general, hidden actions are those actions which do not change any objects in the context.

Next we formally define when an action is hidden as seen from some observing objects. A hidden action does not change the objects from which it is hidden. We formally define this as follows:

Definition: Hidden actions relation

Given an action α and a configuration of objects A||D and where $\alpha \in$ Traces(A||D). We define the hidden actions relative to a configuration D, denoted $\alpha \oplus$ D, as a relation such that

$$\forall\ A, D, \alpha \bullet A||D \xrightarrow{\alpha} A'||D' \wedge \alpha \oplus D \Rightarrow D' \equiv D$$

where $\equiv$ denotes textual equality

---

[7] This definition of observable actions is different from, eg, Hoare's definition of observable actions. In (Hoare 1978) Hoare defines obs(C) as those actions *from* C which are observed by others. This definition, however, defines obs(C) as those action which are *observable from* C. With the definition given here it is possible to distinguish between different observers, something which Hoare's definition does not. The distinction in necessary when considering systems consisting of more than two parts, ie, systems created by composing three or more components.

This definition says that an action α from A||D is hidden to D, denoted $\alpha \oplus D$, if and only if the execution of the sentence giving α do not lead to any changes in the objects in D, ie, that D' ≡ D when D' is given by $A||D \xrightarrow{\alpha} A'||D'$.

An action is *silent* if it is an action from execution of a sentence in a component and the action is not observable from the component's context. This means that an action which updates a variable in the component of the executed sentence is silent. Also, an assignment action which updates a slot in the component of the executed sentence is silent. A silent object creation action will update a slot and create an object from a template in the component of the executed sentence. Error actions are silent if they stem from execution of a sentence which is *not* in an observing object.

What follows is a formal definition of a silent action. It is then shown that a silent action is also a hidden action.

Definition: Silent actions

We say that an action α is *silent* relative to a configuration D, denoted $\alpha \otimes D$, if it is not observable from the configuration and the action stems from execution of a sentence in an object not in the configuration. This can be formally defined:

$$\alpha \otimes D == \alpha.exe \notin D \wedge \alpha \notin obs(D)$$

This notation can be lifted to sequences of actions as follows:

$<\alpha_1, ..., \alpha_n> \otimes D = \alpha_1 \otimes D \wedge ... \wedge \alpha_n \otimes D$

Observation O.4.1.1 : Non-observed actions are either silent or from execution of a sentence in observers

When we have $<\alpha>/obs(O) = \emptyset$ then we have $\alpha \otimes O \vee \alpha.exe \in O$. This means that an action which is not observed is either a silent action or an action from execution of a sentence in an observing object.

We have that silent actions are observably equal to an empty action sequence, ie,:

$$\overline{\alpha} \otimes O \quad \Leftrightarrow \quad \overline{\alpha}/obs(O) = <>$$

Proposition P.4.1.1 Silent actions are hidden actions

$\forall$ A, D, α •
$\alpha \in$ Traces(A||D) $\Rightarrow (\alpha \otimes D \Rightarrow \alpha \oplus D)$

Proof:

Assume $\alpha \otimes D$ and $A||D \xrightarrow{\alpha} A'||D'$. We show that D' ≡ D and then by definition of hidden actions this gives $\alpha \oplus D$.

Cases for the different kinds of actions α:

e->o!m($\overline{p}$)/k : Since the action is not observable from D then o ∈ A

Since the action came from a send sentence in A and the receiver is an A-object, the method-copy will be placed in A by definition of primed configurations, and then there will be no change in D.

e->o.s:=k/j : Since the action is not observable from D then o ∈ A and j ∈ A

Since o ∈ A the updated slot is in A. Because the clone original is an object in A, then by the definition of primed configurations, the new clone is placed in A. Then both the update slot and the new clone is in A and then D is not changed.

e->$o_1.s_1,\ldots,o_n.s_n$:=j : Since the action is not observable from D then for i ∈ {1..n} we have $o_i \in A$

Since $o_i \in A$, the updated slots are in A, and then D is not changed.

e->error : Since the action is not observable from D then e ∈ A

Since the action came from A, the terminated object is in A and this will not change D.

❑

Proposition P.4.1.2 Hidden actions are silent actions except for trivial assignment

For all configurations A and D and action α where $\alpha \in$ Traces(A||D) and where α is not a trivial assignment action, ie, where the slots in D get the same values as they had, we have that if α is a hidden action, then α is a silent action, ie, we have :

$$(\alpha \otimes D \Leftarrow \alpha \oplus D)$$

Proof:
We can then show that except for trivial assignment we have for all cases where $\alpha \otimes D$ do not explicitly hold we have $D' \neq D$ where $\neq$ denote textual inequality. This means showing

$\neg (\alpha \otimes D) \Rightarrow \neg (\alpha \oplus D)$

This gives $\alpha \oplus D \Rightarrow \alpha \otimes D$. By definition of silent action we have

$\neg (\alpha \otimes D) = \neg (\alpha.exe \notin D \wedge \alpha \notin obs(D)) = \alpha.exe \in D \vee \alpha \in obs(D)$

We therefore show:

$\alpha.exe \in D \vee \alpha \in obs(D) \Rightarrow D' \neq D$ where $\neq$ denote textual inequality.

If $\alpha.exe \in D$ then the execution mark is moved and we do not have $D' \equiv D$ and then $D' \neq D$.

Cases for different actions when $\alpha \in obs(D)$:

e->o!m($\bar{p}$)/k : Since the action is observable from D then $o \in D$

Since the receiver is a D-object, the method-copy will be placed in D by definition of primed configurations, the new method copy will be added to D giving $D' \neq D$.

e->o.s:=k/j : Since the action is observable from D then $o \in D$ or $j \in D$

If $o \in D$ the updated slot is in D giving $D' \neq D$. Because the clone original is an object in D, then by the definition of primed configurations, the new clone is placed in D giving $D' \neq D$.

e->$o_1.s_1,\ldots,o_n.s_n$:=j : Since the action is observable from D then for $i \in \{1..n\}$ we have $o_i \in D$

Since $o_i \in D$, the updated slots are in D giving $D' \neq D$ provided the values of $o_1.s_1,\ldots,o_n.s_n$ where different from j.

e->error : Since the action is observable from D then $e \in D$

Since the terminated object is in D the execution mark in the object is removed from D giving $D' \neq D$.

❑

Note that when an action is non-observed, it is not generally a hidden or silent action. This is because when an action α is hidden or silent relative to an observing configuration D, then by definition of silent and hidden actions we have $\alpha.exe \notin D$. However, by definition of observable actions we may have actions α where $\alpha.exe \in D$ which are non-observed, ie, $\alpha \notin obs(D)$.

4.2 Observable Equality

This section gives a definition of observably equal actions. First the definition is given and then the definition is compared with the definitions in chapter 2.4.

As in the previous section, the definitions in this section may also be easier to understand if the text editor example is kept in mind. The definitions below may then be viewed as focusing on the observable equality of TextModel and MyModel relative to TextEditor. The big letter O is used to denote the set of observer objects. In the example this corresponds to the objects found in TextEditor. In the following, two action names are used. These are α and β. Referring to the above example, α can be thought of as stemming from execution of a sentence in MyModel||TextEditor and β from execution of a sentence in TextModel||TextEditor. In general, β is then an action from execution of a configuration consisting of two component specifications and α is from execution of a sentence in a configuration consisting of a specification configuration combined with a possible refinement of the other specification.

4.2.1 Definition of observably equal actions

Below we define observably equal actions. The motivation for the definition is found in section 2.4. The below formal definitions are compared with these definitions further below. This formal definition is given by a relation called the *observably equal actions relation*. The relation is an equivalence relation. This is shown in proposition P.4.2.1 below.

Definition: Observable equality relative to a set of object names; $\sim_O$

Two object names, e,f, are observably equal relative to a set of object names O, denoted $e \sim_O f$, if:

- either they are equal
- or none of them are names in O

This can be formally defined:

$$e \sim_O f \quad == \quad e \in O \vee f \in O \Rightarrow e = f$$

The definition of observable equality can be lifted to sequences of names as follows:

$$\langle e_1,..,e_n\rangle \sim_O \langle f_1,..,f_m\rangle \quad == \quad n = m \wedge \forall\, i : \{1..n\} \bullet e_i \sim_O f_i$$

An action α is said to be observably equal to an action β relative to a set of object names O, denoted $\alpha \sim_O \beta$, iff $\alpha.exe \sim_O \beta.exe$ and $\alpha.dsc \sim_O \beta.dsc$. This is formally stated:

$$\alpha \sim_O \beta \quad == \quad \alpha.exe \sim_O \beta.exe \wedge \alpha.dsc \sim_O \beta.dsc$$

where observable equality of the description part of actions is defined as follows:

$$o!x(\overline{q})/k \sim_O p!y(\overline{p})/l \quad == \quad o \in O \vee p \in O \Rightarrow \langle o, x, k\rangle = \langle p, y, l\rangle \wedge \overline{q} \sim_O \overline{p}$$

$$\overline{o.s} := i \sim_O \overline{p.t} := j \quad == \quad \#\overline{o.s} = \#\overline{p.t} \wedge \forall\, n \le \#\overline{o.s} \bullet o_n \in O \vee p_n \in O \Rightarrow o_n.s_n = p_n.t_n \wedge i \sim_O j$$

$$i.s:=k/o \sim_O j.t:=l/p \quad == \quad (i \in O \vee j \in O \Rightarrow i.s = j.t \wedge k \sim_O l) \wedge (o \in O \vee p \in O \Rightarrow \langle o, k\rangle = \langle p, l\rangle)$$

$$error \sim_O error \quad == \quad true$$

The definition of observable equality of actions relative to a set of object names can be lifted to sequences of actions as follows:

$$\overline{\alpha} \sim_O \overline{\beta} \quad == \quad \forall\, i \le \#\overline{\alpha} \bullet \alpha_i \sim_O \beta_i$$

If two actions are observably equal and one of the actions observable, then both must be observable. This is so because it is required that all the names in the actions which are related to the observability of the actions, must be equal. For instance, it is required that the receivers in message-send actions are equal, and it is the receiver

which determines whether an action is observable or hidden. This corresponds with how observability is defined in chapter 2.

The consequence of this definition is that an observable action and a hidden action are never observably equal. If this was not so, we would have a situation where two actions, one observable from a context and the other hidden to the context, would be observably equal. This would somewhat contradicts the idea of distinguishing between observable and hidden actions.

Proposition P.4.2.1: Observable equality is an equivalence relation

Observable equality is an equivalence relation since we have that the relation is:

reflexive: $x \sim_O x$
commutative: $x \sim_O y \Rightarrow y \sim_O x$
transitive: $x \sim_O y \wedge y \sim_O z \Rightarrow x \sim_O z$

for any x, y, z which denote either three names, three name sequences, three description parts of actions or three actions

Proof:
First we show the case when x, y and z are names:
Reflexive:
$e \sim_O e$ is true since e is always equal to e and then we have $e \in O \vee e \in O \Rightarrow e = e$
Commutative:
$e \sim_O f \Rightarrow f \sim_O e$ is true since the definition is symmetric with respect to e and f.
Transitive:
$e \sim_O f \wedge f \sim_O g \Rightarrow e \sim_O g$ holds since we have:
$e \in O \vee f \in O \Rightarrow e = f$ and $f \in O \vee g \in O \Rightarrow f = g$
and then if $f \in O$ then we have $e = f$ and $f = g$ and then $e = g$ which gives $e \sim_O g$,
and if $f \notin O$ then $e,g \notin O$, since if e and/or g is in O and the premise holds, then
we would have $e = f$ and $f = g$ which can not be true if f is not in O while the others are.
Thus, when $e,g \notin O$ then we have $e \sim_O g$.

The proof of equivalence for name sequences, the description parts of actions and actions follows straight forward from the definitions.
❑

Below the formal definition of observably equal actions is compared with the informal definition of observable similarity given in section 2.4. This is done by repeating a part of the formal definition and comparing this with the corresponding part of the presentation in section 2.4. Note that in section 2.4 it was assumed that the actions were observed. In the below this means that it is assumed that typically $o, p \in O$.

Observably equal message-send actions
The formal definition of observably equal actions says:

$$o!x(\overline{q})/k \sim_O p!y(\overline{p})/l \quad == \quad o \in O \vee p \in O \Rightarrow \langle o, x, k\rangle = \langle p, y, l\rangle \wedge \overline{q} \sim_O \overline{p}$$

The informal presentation of observably similar actions given in section 2.4 says (with references to the formal definition in parenthesis):

> Two message-send actions are observably similar to each other if they are messages to the same object ($o = p$). Also, the message selector is the same in the two actions ($x = y$). The parameters which are names of observing objects are equal ($q_i = p_i$). Parameters which are names of object in the component may be different ($q_i \notin O \wedge p_i \notin O$).

We also require that the names of the method copies are equal if the message-send action is observed. This is further discussed in relation to observation O.4.3.2 in a later section in this chapter.

Observably equal assignment actions:
The formal definition of observably equal assignment actions says:

$$(\overline{o.s} := i) \sim_O (\overline{p.t} := j) \quad == \quad \#\overline{o.s} = \#\overline{p.t} \wedge \forall n \leq \#\overline{o.s} \bullet o_n \in O \vee p_n \in O \Rightarrow o_n.s_n = p_n.t_n \wedge i \sim_O j$$

The informal presentation of observably similar actions given in section 2.4 says (with references to the formal definition in parenthesis):

Two assignment actions are observably similar when they update the same slot in the same object ($o_n.s_n = p_n.t_n$ for all $n \leq \#\overline{o.s}$). If, in either of the actions, the new value of the updated slot is the name of an observing object then the new value is the same in the two actions (i = j). If the new value is not found as the name of an object in the context, then both actions must have new values which are different from context object names ($i \notin O \wedge j \notin O$).

Observably equal object creation actions:
The formal definition of observably equal object creation actions says:

$$i.s:=k/o \sim_O j.t:=l/p \quad == \quad (i \in O \vee j \in O \Rightarrow i.s = j.t \wedge k \sim_O l) \wedge (o \in O \vee p \in O \Rightarrow <o, k> = <p, l>)$$

The informal presentation of observably similar actions given in section 2.4 says (with references to the formal definition in parenthesis):

Two object creation actions are observably similar if they create objects from the same template (In the formal definition, templates are objects in observers and we therefore get the following: $o \in O \vee p \in O \Rightarrow o = p$).

In addition, the Omicron object creation actions update slots. Similarity of this part of the actions are handled as observably similar assignment actions. Therefore it is required that k = l when $o \in O \vee p \in O$. This requirement is also discussed in relation to observation O.4.3.2 in a later section in this chapter.

Observably equal error actions:
Two error actions are observably equal if they stem from execution of sentences in the same observer object. They are also observably equal if they are from execution of sentences in objects which are not observers. This corresponds with the description of similarity of error actions in section 2.4 which says:

Two error actions are equal when they are errors in the same object ($\alpha.exe \sim_O \beta.exe$).

4.2.2 Observably equal action sequences

This subsection defines a relation between sequences of actions. The relation is called the observably equal action sequence relation, and $\overline{\alpha} =_O \overline{\beta}$ denotes that the action sequence $\overline{\alpha}$ is observably equal to the action sequence $\overline{\beta}$ relative to a set of object names O.

Definition: Observably equal action sequences

Given two action sequences $\overline{\alpha}$ and $\overline{\beta}$ and a set of object names O. We define a binary relation called *an observable equality between action sequences*, denoted $=_O$, for any two sequences $\overline{\alpha}$ and $\overline{\beta}$ as follows:

$$\overline{\alpha} =_O \overline{\beta} \quad == \quad \overline{\alpha}/obs(O) \sim_O \overline{\beta}/obs(O)$$

This definition says that for each O-observed action in $\overline{\alpha}$ there must be an observably equal action in $\overline{\beta}$. Below, $\alpha =_O \overline{\beta}$ is used as short hand notation for $<\alpha> =_O \overline{\beta}$. Also we let

$$\overline{\alpha} =_{D'} \overline{\beta}$$

denote $\overline{\alpha} =_O \overline{\beta}$ where O = D'.Dom = D.Dom $\cup$ NewNames($\overline{\alpha}$, D)

Proposition P.4.2.3 The observably equal action sequences relation is an equivalence relation

The observably equal action sequences relation defines an equivalence relation since we have that the relation is:

reflexive: $\overline{\alpha} =_O \overline{\alpha}$

commutative: $\overline{\alpha} =_O \overline{\beta} \Rightarrow \overline{\beta} =_O \overline{\alpha}$

transitive: $\overline{\alpha} =_O \overline{\beta} \wedge \overline{\beta} =_O \overline{\gamma} \Rightarrow \overline{\alpha} =_O \overline{\gamma}$

Proof:
The proposition holds by the definition of the relation and since the observably equal actions relation is an equivalence relation as shown in proposition P.4.2.1.
❑

4.3 A Refinement Relation

This section defines a refinement relation between two Omicron configurations which are viewed as components. The refinement relation is defined relative to a third configuration containing the observers of the component configurations.

The motivation for the formal definitions below is the refinement relation presented in chapter 1 and 2. As defined in chapter 1, a refinement relation relative to a context is said to hold between two configurations, for example MyModel and TextModel, when MyModel will not display any observable behaviour which is not also displayed by TextModel as observable from TextEditor. Therefore, to check this, the two systems MyModel||TextEditor and TextModel||TextEditor are compared by the sequences of observable actions they display when executed. The Traces()-function defined in chapter 3 returns the set of all sequences of actions from execution of a system. This function is therefore used in the below formal definition of the refinement relation. For the refinement relation to hold between MyModel and TextModel then for each action sequence in the traces from MyModel||TextEditor there must be an action sequence in the traces from TextModel||TextEditor which is observably equal. This seems like a straight forward definition. However, there are some aspects to consider in order to make a relation which is not too strict and not too weak. These aspects are discussed below before the formal definition of the refinement relation is presented.

Adding new objects to observers

New objects may be placed in the component or in the observing context. If new objects are placed into the set of observing objects, this will have consequences for which actions in the traces are observed. This in turn will influence whether or not a component will be seen as refinements of an other components. Therefore positioning of newly created objects is essential when defining refinement relations.

In what follows, the link between observability of action and placement of new object presented in chapter 2 was:

> an object created by an observable object creation actions is placed in the observing context and
> an object created by hidden object creation actions is placed in the component
> where the executed sentence is found.

This link was taken into account in the definitions of the NewNames and prime configuration functions found in section 3.4.6. The NewNames function is used in the definition of the refinement relation for traces in order to add new objects to the set of observers. The objects which are added to observers are all the clones of observers and method copies when the receiver is an observer.

Note that when two action sequences are observably equal relative to both O and the objects created by the actions in the sequences (here denoted Q), ie, $\overline{\alpha} =_{O \cup Q} \overline{\beta}$ where Q = NewNames($\overline{\alpha}$, O), then all names of new objects in O are equal in the two action sequences, ie, NewNames($\overline{\alpha}$, O) = NewNames($\overline{\beta}$, O).

Non-deterministic behaviour

As explained in chapter 2, objects' behaviours are specified to be non-deterministic when the behaviour should not or can not be specified exactly. In parallel systems, such as Omicron configurations, objects performing actions in parallel is also a source for non-determinism. This is because the relative execution speed of the objects may vary non-deterministically. Because of this non-determinism, the traces of some specification B||D, ie, Traces(B||D), may contain several different sequences of actions since a component may have non-deterministic behaviour.

As also explained in chapter 2, a refinement does not have to display all the alternative behaviours of a specification. The consequence of this is that we must require that for each action sequences from the system consisting of a refinement, eg, A, and the context, eg, D, there must be an observably equal action sequence from the system consisting of specifications, eg, B||D, ie,

$$\forall\ \overline{\alpha} : \text{Traces(A||D)}\ \exists\ \overline{\beta} : \text{Traces(B||D)} \bullet \overline{\alpha} =_{O} \overline{\beta}$$

where O = D.Dom $\cup$ NewNames($\overline{\alpha}$, D)

Ending collaboration between components and their contexts

A component which has terminated or which executes giving a possibly infinite sequence of hidden actions is said to have ended collaboration with the context. When A ends its collaboration with D in a system A||D this means that A does not have any actions which are observable from D. We can formally define this as follows:

Definition: Ending collaboration

Given two configurations A and D. Then we say that A has ended collaboration with D, denoted endColab(A, D) when A does not have any actions which are observable from D, ie, all actions from A are hidden to D. This if formally defined as follows:

endColab(A, D) == $\forall\ \overline{\alpha}$: Traces(A||D) • $\#\overline{\alpha} > 0 \wedge \overline{\alpha}.\text{exe} \in A \Rightarrow \overline{\alpha} \otimes D$

where $\overline{\alpha}.\text{exe} \in A$ means $\forall\ \alpha : \overline{\alpha}$ • $\alpha.\text{exe} \in A$

and we also define ending collaboration after some initial action sequence $\overline{\alpha}$ as follows:

endColab(A, D, $\overline{\alpha}$) == endColab(A', D')

where A' = prime(A, D, $\overline{\alpha}$) and D' = prime(D, A, $\overline{\alpha}$)

In most component developer's view as presented in chapter 2 components which have terminated and components which execute giving a possibly infinite sequence of hidden actions are seen as equivalent in that none will give observable actions. This means that a refinement should not end collaboration with the context before the specification terminates. This is motivated by the requirement that a specification will not specify actions which the refinement does not display. If we require for A to be a refinement of B relative to D that:

$\forall\ \overline{\alpha}$: Traces(A||D) $\exists\ \overline{\beta}$: Traces(B||D) • $\overline{\alpha} =_O \overline{\beta}$

where O = D.Dom $\cup$ NewNames($\overline{\alpha}$, D)

then A may end collaboration with D before B does. This is because when A has ended collaboration with D then we have $\overline{\alpha}$/Obs(O) = <> and then we can always find some $\overline{\beta}$ such that $\overline{\alpha} =_O \overline{\beta}$. This is done by letting $\overline{\beta}$ be the empty action sequence which is found in the traces of every configuration. This is not in line with component developers view of similar components as described in chapter 2. It must therefore be required that if a refinement ends collaboration with the context, then the specification component should have an alternative behaviour which ends collaboration with the context. This is formally stated:

endColab(A, D, $\overline{\alpha}$) $\Rightarrow$ endColab(B, D, $\overline{\beta}$) where $\overline{\alpha} \in$ Traces(A||D) and $\overline{\beta} \in$ Traces(B||D) and $\overline{\alpha} =_{D'} \overline{\beta}$.

A refinement relation

Based on the above discussion, the refinement relation between configurations is defined as follows:

Definition: Refinement relation; $A \leq_D B$

Given two systems A||D and B||D. The configuration A is a refinement of B relative to D, denoted $A \leq_D B$, if the traces of A||D is a refinement of the traces of B||D relative to D. This can be formally defined as follows:

$A \leq_D B$ == $\forall\ \overline{\alpha}$: Traces(A||D) $\exists\ \overline{\beta}$: Traces(B||D) •

$\overline{\alpha} =_{D'} \overline{\beta} \wedge$ (endColab(A, D, $\overline{\alpha}$) $\Rightarrow$ endColab(B, D, $\overline{\beta}$))

This definition says that for all action sequences in the traces Traces(A||D), there is an observably equal action sequence in the traces Traces(B||D) relative to the object names in D and the names of the new observing objects. Also, if A ends collaborating with D after the action sequences $\overline{\alpha}$, then B can end collaboration with D after the action sequence $\overline{\beta}$.

In the way Omicron configurations can be defined, there may be an infinite number of action sequences in the traces of the configuration. Also, an action sequence may have an infinite number of actions. If such infinity is found in the traces, then it would take infinite time to decide by execution or simulation of execution if a component with infinity in its traces is a refinement of some other component. However, if the component with infinity in its traces is not a refinement of some other component, then this is decideable by comparing executions or simulations of executions in finite time since it takes a finite time to find the action for which no observably equal action is found. Refinement is therefore not in general testable by execution of components or simulations of such components, since such a test might take infinite time. Since refinement is not in general

testable for OCS components, it is important to have a formal theory for such components which can be used in showing properties such as refinement. It is then important that proofs in the theory of properties such as refinement can be done in finite time.

When defining OCS components using Omicron, the possible traces of a component is deterministically defined. For instance it is deterministic weather the execution of a component will give new actions or if the component has stopped and therefore will give no new actions. In the first case there is one or more sentences in the component which has an execution mark in from of it. In the latter case, there are no such sentences in the configuration. Because of this deterministic property, a function such as endColab() can be defined by a prefix closed set of finite sequences of actions. This allows proofs of properties, such as refinement of Omicron configurations, typically to be done by induction which are proofs which can be done in finite time.

Proposition P.4.3.2: The refinement relation is a pre-order

The refinement relation is a pre-order because the relation is:

reflexive: $A \leq_D A$

transitive: $A \leq_D B \wedge B \leq_D E \Rightarrow A \leq_D E$

Proof:

Reflexive: Obvious since the observably equal action sequence relation is reflexive (by P.4.2.3).

Transitive: We have by definition of the refinement relation:

$$\forall\ \bar{\alpha} : \text{Traces}(A||D)\ \exists\ \bar{\beta} : \text{Traces}(B||D) \bullet \bar{\alpha} =_{D'} \bar{\beta} \wedge (\text{endColab}(A, D, \bar{\alpha}) \Rightarrow \text{endColab}(B, D, \bar{\beta}))$$

and

$$\forall\ \bar{\beta} : \text{Traces}(B||D)\ \exists\ \bar{\gamma} : \text{Traces}(E||D) \bullet \bar{\beta} =_{D''} \bar{\gamma} \wedge (\text{endColab}(B, D, \bar{\beta}) \Rightarrow \text{endColab}(E, D, \bar{\gamma}))$$

where $D'' = D.\text{Dom} \cup \text{NewNames}(\bar{\beta}, D)$

Trivially we then have $\text{endColab}(A, D, \bar{\alpha}) \Rightarrow \text{endColab}(E, D, \bar{\gamma})$. Next we show $\bar{\alpha} =_{D'} \bar{\gamma}$.

Since the action sequences $\bar{\alpha}$ and $\bar{\beta}$ are observably equal, then by definition of NewNames, we have D'.Dom = D''.Dom. Then we have

$$\forall\ \bar{\alpha} : \text{Traces}(A||D)\ \exists\ \bar{\beta} : \text{Traces}(B||D),\ \bar{\gamma} : \text{Traces}(E||D) \bullet \bar{\alpha} =_{D'} \bar{\beta} \wedge \bar{\beta} =_{D'} \bar{\gamma}.$$

Since the observably equal action sequence relation is transitive (by P.4.2.3) we have $\bar{\alpha} =_{D'} \bar{\beta} \wedge \bar{\beta} =_{D'} \bar{\gamma} \Rightarrow \bar{\alpha} =_{D'} \bar{\gamma}$ and we then have

$$\forall\ \bar{\alpha} : \text{Traces}(A||D)\ \exists\ \bar{\gamma} : \text{Traces}(E||D) \bullet \bar{\alpha} =_{D'} \bar{\gamma}$$

and the proposition holds for this case.

❑

Observation O.4.3.1 : The refinement relation is neither an equivalence relation nor monotonic

The refinement relation is not symmetric since all sequences of actions in Traces(A||D) are considered, while only a selected set of the action sequences in Traces(B||D) are used to establish the relation between the two traces.

An equivalence relation could be defined as follows:

$$A =_D B \ == \ A \leq_D B \wedge B \leq_D A$$

This relation is left for further study, while the refinement relation is the focus of the present work since this relation is more used in practice.

Also, we do *not* have:

$$A \leq_D B \ \Rightarrow \ \forall\ C : A||C \leq_D B||C$$

or

$$A =_D B \ \Rightarrow \ \forall\ C : A||C =_D B||C$$

Therefore the refinement relation and an equivalence relation based on the refinement relation are not monotonic relations.

A simplifying assumption concerning names of new objects:

The actions include the names of new objects. For two actions to be observably equal, the names of new objects must be equal. For example we have α = e->i.s:=k/o from A||D where o is the name of an object in D. If an action β is to be observably equal to α relative to D we must have β = f->j.t:=k/o. When we require:

$$\forall \alpha : \text{Traces}(A||D\sigma)\ \exists \beta : \text{Traces}(B||D) \bullet \alpha =_D \beta$$

for A to be a refinement of B, then there will never be any refinement of B if there are D-observable message-send and clone actions in A||D. This is due to the requirement that for every action from A||D there must be an observably equal action in B||D. For example if there is a clone action in A||D which clones an object in D, then there will be infinitely many possible actions from A||D since there are infinitely many unique object names. One or more of these actions would then be an action where the name of the new object is equal to an object named in B, since it is only required that names of new objects in A||D are not found as object names or slot values in A||D. When we have $\alpha =_O \beta$, we require that the names of new objects are equal and then β will hold a new object name which is a name already in B. Then β will not be a legal action from B||D. Therefore we do not have a β for all α, and then never $A \leq_D B$ if A sends messages to D or clones objects in D.

There are several alternative ways of making definitions and proofs which eliminates this problem. One example is allowing differences in new object names in observably equal actions. This would not complicate the definition of observably equal actions. However, the extra complications are found in relation to expressing and proving properties which are necessary when making inductive proofs of properties of reliable refinements. The complications will occur in the proofs in chapters 5 and 6 which refer to the below observation, O.4.3.2.

One other way to avoid this problem is to require a refinement to have the same number of objects as the specification configuration and that the object names are equal. This is not in accordance with the intuitive understanding of refinements we have presented previously where a refinement and its specification should be able to have different numbers of objects. As mentioned, there are also several other ways of avoiding the above problem. However, each alternative contribute with complexity in the definitions and proofs. Therefore, a simplifying assumption is introduced:

Observation O.4.3.2 : Simplifying assumption about names of created objects

> Names of new objects found in actions are assumed to be different from every object name, slot name and slot value in the configurations associated with the expression where the new names are found.

This means that if we, eg, have as above $\forall \alpha : \text{Traces}(A||D\sigma)\ \exists \beta : \text{Traces}(B||D) \bullet \alpha =_O \beta$, then we assume that all names of new objects in actions in α and β are not found as object names, slot names or slot values in A, B or D. Obviously, since there are infinitely many names, it is always possible to find new names which are not used as object name or slot value in any of the involved configurations. Making this assumption therefore ensures that all names of new object which are found in α from A||D can also be used as names of new objects in B||D.

The main reason for choosing the above alternative and thereby leave some of the complexity out, is that it corresponds well with what happens in practice. In practice, the difference in new objects' names is not considered when comparing equality of components as long as the object names are unique. The uniqueness of names are taken care of by the underlying runtime system, thus removing the burden of naming object from the programmer. By our simplifying assumption, an equal burden is removed from the person doing the proofs of propositions and lemmas used in showing the substitution proposition.

CHAPTER 5

Reliability Requirements

The definitions in chapter 4 were based on the intuitive notions of similar observable behaviour found among component developers as described in chapter 2. This leads to a refinement relation which is not reliable as defined in chapter 1. The present chapter shows how and why this definition of configuration refinement is not reliable. This chapter defines a set of *reliability requirements* for Omicron configurations and shows how the refinement relation defined in chapter 4 can be strengthened so that we get a reliable refinement relation which ensures reliable substitution of components.

Section 5.1 gives an introduction to reliability of Omicron configurations while section 5.2 introduces *name substitutions* which are used to specialise a configuration when combining it with different other configurations. This allows components to have different object names, while still having observably equal behaviour.

The lack of reliability is shown by giving examples of components where the properties expressed in the substitution proposition do not hold. The examples are given in sections 5.3 and 5.4. From the problems related to showing reliability properties for the simple examples in section 5.3, certain requirements on configuration expressions are introduced. There requirements are denoted *reliability requirements.*

In section 5.4 the examples show how the definition of observable equality as defined in chapter 4 must be modified to get reliability. This is summed up in the definition of observable similarity. The observably similar actions relation is transitive and not an equivalence relation like the observably equal actions relation. This is because the observably similar actions relation is not reflexive or commutative. However, the relation is reflexive in what might be considered normal cases.

Observable similarity is used to define a reliable refinement relation, something which is done in section 5.5. This section also shows that the reliable refinement relation is in line with the developers notion of similarity, but in addition sets new requirements on components if they are to be reliable.

In chapter 6 it is shown that the reliability requirements combined with the new refinement relation are sufficient to show the substitution proposition of chapter 1.

More practical consequences of the reliability requirements and the reliable refinement relation with specialisation are presented in chapter 8. The consequences are discussed and they are summed up as a set of rules for making reliable specifications and reliable refinements of such specifications.

5.1 Reliability of Refinements

The proof that the refinement relation is a reliable refinement relation is done by showing the substitution proposition of chapter 1. This means that, eg, it is possible to prove the simple version of the substitution proposition:

$$\forall A, B, C, D \bullet A \leq_D B \wedge C \leq_B D \Rightarrow A \leq_C B \wedge C \leq_A D$$

where $A \leq_D B$ means that A is a refinement of B relative to D, etc. Because of the universal form of the above proposition it is only necessary to prove:

$$\forall A, B, C, D \bullet A \leq_D B \wedge C \leq_B D \Rightarrow A \leq_C B$$

as $C \leq_A D$ follows by symmetry. The configurations denoted B and D may be viewed as specifications. The configuration denoted A is a refinement of B and C is a refinement of D.

Because of the form of the substitutability proposition, we can say that the configurations denoted A and C "play equal roles" and B and D "play equal roles" in the sense that A and C are refinements and B and D are specifications. Configurations playing equal roles will have to meet the same reliability requirements and take the same place in propositions and proofs in relation to proving the substitution proposition. Remembering this symmetry will help in understanding the formalisations and discussions in the following sections as, eg, something which is stated for A will also hold for C and statements about *B relative to A and C* could just as well be statements about *D relative to C and A*.

To show that A is a refinement of B relative to D, ie, $A \leq_D B$, the traces of the two configurations A||D and B||D are compared. It is then shown that for every action sequences in traces of A||D there is an observably similar action sequence in traces of B||D. Similarly, to show $C \leq_B D$ the traces of B||C and B||D are compared. When the conclusion of the substitution proposition holds then for every action sequences in traces of A||C there is an observably similar action sequence in traces of A||D. Observational similarity of C and D is then defined relative to the objects in A. Note that the premise states properties of the traces of B||D, A||D and B||C, while the conclusion states properties of the traces of A||D, B||C and A||C. This means that properties of the traces of the configuration A||C is only stated in the conclusion of the proposition. Remembering this will also help in understanding the formalisations and discussions in the following sections.

To simplify, the following discussions, definitions and propositions are based on the simple version of the substitution proposition shown above, in which four configuration names are used: A, B, C and D. This naming scheme will be followed below in that whenever some property is defined and that property is typically relevant for specifications, the configurations are denoted B or D. If the property is typically relevant for refinements, the configurations are denoted A and C. There are no a priory assumptions on configurations with these names. The consistent use of names is only meant to help in understanding the definitions, discussions, propositions and theorems.

Keeping the definition of reliable refinements as abstract as possible

In the definition of the refinement relation in chapter 4 there are no restrictions on how to make operational specifications of observable behaviour. The operational specifications are Omicron configurations and this creates total freedom in using Omicron when expressing observable behaviours. All details in the operational specifications which are irrelevant to the observers are not taken into account when determining whether a component is a refinement of some other component. The resulting definition of similar components may therefore be seen as abstracting away details of components which are irrelevant to the observers.

There are different alternative requirements which may be added to make the definition of the refinement relation reliable. The different alternative reliability requirements can put restrictions on

- Omicron configurations and/or
- on relations involving actions

If restrictions are put on Omicron configurations then this will restrict how implementations and specifications of behaviours are done. To get as much freedom as possible when expressing observable behaviours, it is therefore best to put requirements on relations involving actions rather than on the Omicron configurations which operationally describe the actions. The reliability requirements preferred in this thesis are therefore those which avoid putting restrictions on Omicron configuration. In other words, it is preferred to set requirements on relations involving actions.

However, this chapter shows that in order to that to get reliability, there is no way to avoid some restrictions on configurations. These are found in section 5.3 below. In chapter 8 it is shown that many of these restrictions correspond with published advice on how to make good object-oriented designs while other restrictions introduce and motivate new design advice. The restrictions on Omicron configurations may be supported by a properly defined language where the restrictions can be statically checked. However, as discussed in this chapter and in chapter 8 it is not trivial to define such a language and it is not evident that all the reliability requirements can be statically checked even with a properly defined language. Defining such a language is therefore left for further study. It can also be noted that traditional class based languages ensure several of the reliability requirements, but on the other hand put unnecessary restrictions on configuration expressions.

Some reliability requirements can be avoided by establishing properties of the configurations which are formed by combining refinement configurations. Referring to the above named configurations, this means establishing properties of A||C. By using this alternative we would loose the property found and expressed in the substitutions proposition. This property is that the refinement relation can be established separately for each refinement configuration, and when these relations have been established, the refinements can be safely combined. This is often called the *compositionality property* of components. However, as explained in section 5.4 there are a few simple requirements which must be put on A and C in order to get reliability. This is the requirement called reliable names formally defined by the RelNames-function.

5.2 Configuration Specialisation

This section introduces *name substitutions* which are used to *specialise* a configuration when combining it with different other configurations allowing components to have different object names, while still having observably equal behaviour. This section also define what it means to be a reliable substitution and a configuration with safe names are defined. These properties are necessary in order to ensure predictable behaviour when configurations are specialised and combined with other specialised configurations.

5.2.1 Name substitutions

The following configurations give an example of a situation where it is useful to have name substitutions and configuration specialisation:

B = b : ([m->p],) || p: ([:y->j, v->n], y!v();)
A = a : ([m->p],) || p: ([:y->j, v->n], y!v();)
D = d : ([s->b, w->m, x->d, n->q], $s!w(x);) || q : ([ ],)

In this case we have:

Traces(B||D) = < d->b!m(d); k_p ->d!n() >
Traces(A||D) = < d->error>

This does not give A $\leq_D$ B. The only difference between A and B is the name of one of the objects. In general we then get that we do not have A $\leq_D$ B if the names of visible objects from A and B to D differ, ie, when

D.Values $\cap$ A.Dom $\neq$ D.Values $\cap$ B.Dom

In order to specialise D to collaborate with A instead of B we introduce name substitutions:

Definition: A name substitution

A name substitution[8] is an operation mapping names to names. Name substitutions are denoted by small Greek letters and the set of all name substitutions is denoted $\mathscr{Su}$.

The substitution operation is written post fix, eg, Cσ, and binds stronger than the other operations so that, eg, A||Cσ means A || (Cσ).

A name substitution has the form $\{a_1/b_1\}\ldots\{a_n/b_n\}$ where {a/b} means 'b' is replaced by 'a'. For b-names not listed in the {a/b}-sequence the substitution mapping is equal to the identity function. For the substitution to be well-defined, all b_i in the sequence must be unique. {a/b}* denote a sequence of {a/b}.

Given a substitution $\sigma = \{a_1/b_1\}\ldots\{a_n/b_n\}$ then
keys(σ) = $\{b_1\ldots b_n\}$, values(σ) = $\{a_1\ldots a_n\}$ and names(σ) = keys(σ) $\cup$ values(σ).

To make D have A-names in stead of B-names we *specialise* D by applying a name substitution which substitute B-names with A-names, eg, we have A||Dσ where $\sigma \in$ (B.Dom $\rightarrow$ A.Dom). Similarly we use a substitution $\rho \in$ (D.Dom $\rightarrow$ C.Dom) to specialise B to collaborate with C instead of D.

For short we write $\sigma \in$ B $\rightarrow$ A for $\sigma \in$ (B.Dom $\rightarrow$ A.Dom).

The role of configurations and substitutions which will be used when expressing a reliable version of the substitution proposition is illustrated in figure F.5.1 below.

5.2.2 Substitution of observer names

If a substitution σ has keys which are equal to names of the observers, ie, D.Dom in the above example, and the keys are new names, then the names of observers are changed when D is specialised with σ. If observer names are changed when configurations are specialised, then this must be taken into account when defining observable equality of actions. This creates extra complexity in the definition of observable equality. However, by requiring that we have B||D $\in \mathscr{C}$, ie, B.Dom $\cap$ D.Dom = Ø, and that the substitution substitute from B-names to A-names, ie, $\sigma \in$ B.Dom $\rightarrow$ A.Dom, then the D-names will never be changed.

[8] Name substitutions are defined in some more detail in appendix A.

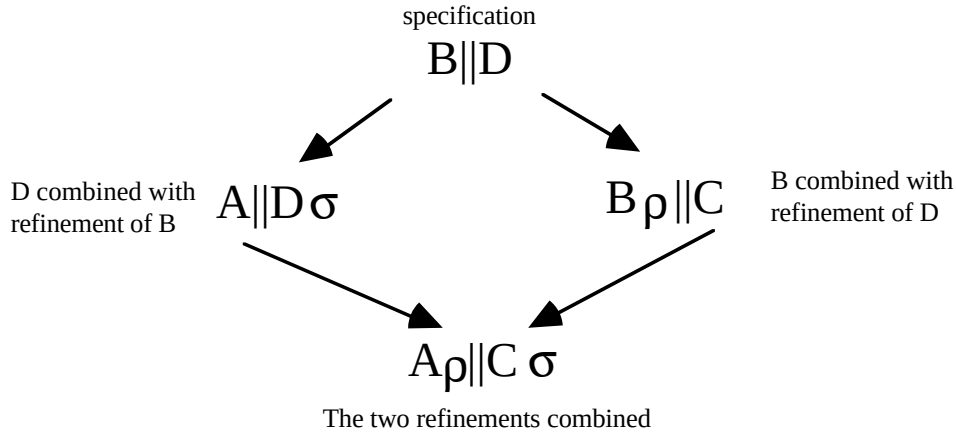


Figure F.5.1 : Configurations and substitutions which will give reliable behaviour provided the reliability requirements presented in this chapter hold.

Observation O.5.2.1 : Observing objects' names are never keys in the substitution

When we have

$B||D \in \mathscr{C}$ which gives $B.Dom \cap D.Dom = \emptyset$

and when we have

$\sigma \in B \to A$,

then we have

$(\forall i \bullet i \in D \Rightarrow i = i\sigma)$

since when $i \in D$ and $B.Dom \cap D.Dom = \emptyset$ then $i \notin B$ and then $i \notin keys(\sigma)$ since $\sigma \in B \to A$.

We then also have, eg,

$D.Dom = D\sigma.Dom$ and $(\forall i \bullet i \in D.Values \wedge i \in D \Rightarrow i = i\sigma)$

and we also have

$E||D \in \mathscr{C} \Leftrightarrow E||D\sigma \in \mathscr{C}$ for any configuration E since when $D.Dom = D\sigma.Dom$ then $E.Dom \cap D.Dom = E.Dom \cap D\sigma.Dom$

When A and Dσ are combined, then we must require safe names in the configuration A||Dσ. By observation O.5.2.1 (observing objects' names are never keys in the substitution) then $A||D \in \mathscr{C}$ gives $A||D\sigma \in \mathscr{C}$.

Definition: Reliable substitution relative to configurations

When we have $A||D\sigma$, $B||D \in \mathscr{C}$ and $\sigma \in B \to A$ then we say that the substitution σ is *reliable* relative to A, B and D and we denote this RelSubst(σ, A, B, D).

Proposition P.5.2.1 Reliable substitutions preserve substitution reliability

$\forall$ A, B, C, D, σ, ρ •

$A||D, B||D \in \mathscr{C} \wedge \sigma \in A \to B \wedge \rho \in D \to C$

$\Rightarrow$

$B\rho||D, A\rho||D\sigma \in \mathscr{C} \wedge \sigma \in A\rho \to B\rho$

Proof:

By proposition O.5.2.1 (observing objects' names are never keys in the substitution) we have $A.Dom = A\rho.Dom$, $B.Dom = B\rho.Dom$ and $D.Dom = D\sigma.Dom$. $B||D \in \mathscr{C}$ gives $B.Dom \cap D.Dom = \emptyset$, and we have $A||D, B||D \in \mathscr{C} \wedge \sigma \in A \to B$, this gives $B\rho||D, A\rho||D\sigma \in \mathscr{C} \wedge \sigma \in A\rho \to B\rho$.

❑

5.2.3 Substitution of slot names

In Omicron configurations there is no syntactic difference between slot names and object names. Therefore, there may be slot names in D which are equal to object names in B. If such a slot name was found as a key in the substitution this would create unintentional change in behaviour. Here is an example:

> What would have been observably equal messages from A and B to D will be a message send from B to D and an error action in A||Dσ as the name of the receivers method-slot was changed by the substitution.

This is demonstrated by the following example:

B = b : ([m->p],) || p: ([:y->j, v->n], y!v();)
A = a : ([m->p],) || p: ([:y->j, v->n], y!v();)
D = d : ([a->b, b->m, x->d, n->q], $a!b(x);) || q : ([ ],)

Consider the substitution {a/b}. Here a and b are also found as slot names in the object named d and we then get
Dσ = d : ([a->a, a->m, x->d, n->q], $a!a(x);) || q : ([ ],)

and we then have
Traces(B||D) = < d->b!m(d); k_p ->d!n() >
Traces(A||Dσ) = < d->error>
because the slot name b was replaced with a. In this case there are two slots with the same name in a single object, but in general such a substitution may lead to two slots with the same name in an inheritance graph of an object. When there are two slots with the same name, then the slot which comes first in the lookup sequence will be used in all cases. In the case above we will therefore get the value b from D(d:a). We may then get differences in behaviour as in the above case and then not A $\leq_D$ B, even if the difference in behaviour has nothing to do with the definitions of A and B.

This problem with slot name substitution may be viewed as a problem related to how substitutions are used when specialising configurations. The problem arises when the substitution is applied to a configuration, such as is done by Dσ. There are several ways of getting around this problem. Here are some examples:

- Define how substitutions are applied to configurations, and in the definition say that slot names are not changed. The problem with this approach is that it is not statically decideable whether a slot value is a slot name or an object name and therefore should not or should be changed. Typically, a slot value could both be the name of a slot in D and the name of an object in B.

- Require that slot names in D are not equal to B and A names found in σ, ie,
  D.SlotNames ∩ names(σ) = Ø
  Similarly for slot names in B relative to the substitution from D-names to C-names. The drawback with this solution is that it will complicate many definitions which the reliable refinement definition builds on. These extra elements in the definitions do not actually correspond to problems which are found in practice, but are mainly of theoretical interest since it stems from the use of substitutions to specialise configurations.

- A simple solution is to require disjoint sets of names for slots and objects, ie, splitting the set of names $\mathcal{N}$ into the sets $\mathcal{ON}$ for object names and $\mathcal{SN}$ for slot names. We then require:

  $$\mathcal{ON} \cap \mathcal{SN} = \emptyset \quad \text{and} \quad \mathcal{N} = \mathcal{ON} \cup \mathcal{SN}$$

  Naturally, we require that the object names used in the configurations are in $\mathcal{ON}$, for instance that all object names in B are in $\mathcal{ON}$, ie, B.Dom $\subseteq \mathcal{ON}$. Slot names in the configurations will then be in $\mathcal{SN}$.

  The advantage of this approach is that it simplifies the definition of a reliable refinement relation while it sets a restriction which is found in almost every object-oriented system. It is common in all object-oriented programming languages such as C++ and Smalltalk[9]. In such systems the object names are created and maintained by the underlying runtime system, while slot names correspond to variable names and procedure names which are defined by programmers.

  An important property of this approach is that it does not restrict the values of slots to only contain object names; slot values may also be slot names.

To avoid such slot name substitution and with as little extra complexity as possible, but without making unnecessary restrictions in practice, the last approach is chosen.

Definition: Configurations with safe names
Given a set of configurations $C_1,\ldots, C_n$, The configurations have safe names if they have non-overlapping object names and the set of slot names and the set of object names of the configuration are

[9] In Smalltalk there is a tradition for sending what corresponds to slot names as parameters. This is further commented on in chapter 8.

disjoint, ie, $C_1||\ldots||C_n \in \mathscr{C}_S \wedge ON(C_1||\ldots||C_n) \cap SN(C_1||\ldots||C_n) = \emptyset$ where ON() and SN() is defined as follows:

$ON(C) == C.Dom$
$SN(\ C_1 \ ||\ C_2) == SN(C_1) \cup SN(C_2)$
$SN(\ i : M, S) == SN(M) \cup SN(S)$
$SN([s_1 \text{->} i_1, \ldots, s_n \text{->} i_n) == \{s_1,\ldots,s_n\}$
$SN(S_1\ \$\ S_2) == SN(S_1) \cup SN(S_2)$
$SN(S_1\ ;\ S_2) == SN(S_1) \cup SN(S_2)$
$SN(s!w(p_1,\ldots,p_n)) == \{s, w, p_1,\ldots,p_n\}$
$SN(s := v = w\ t\ f) == \{s, v, w, t, f\ \}$
$SN(s := t\ clone) == \{s, t\}$

When the configurations have safe names we write this as follows:

$C_1||\ldots||C_n \in \mathscr{C}_{Safe}$

When the configurations have safe names, and the substitution is reliable, then slot names are not changed by the substitution:

Observation O.5.2.2 : Reliable substitutions do not change slot names in configurations with safe names

When we have configurations A, B, and D where

$A||D,\ B||D \in \mathscr{C}_{Safe}$

and a substitution σ where

$\sigma \in B \rightarrow A$

then we have that

all object names in the configurations are in $\mathscr{ON}$, and

slot names are in $\mathscr{SN}$ where $\mathscr{SN} \cap \mathscr{ON} = \emptyset$,

and then only B-names will be substituted, while all slot names are unchanged by the substitution σ in $D\sigma$.

The following is the BNF-syntax definition for Omicron configurations with safe names. Oname is used for object names while Sname is used for slot name. The difference between this definition and the definition in chapter 3 is that all occurrences of "name" is replaced by "Sname" except in the line marked (*). In this line, "name" is replaced by "Oname". In the line marked (!). Oname is added to the definition of Val. The line marked (+) defines Oname.

	Configuration	::=	$Object^*_{\|\|}$	
(*)	Object	::=	Oname : **(**Slots**,** Body**)**	-- Definition of an object
	Slots	::=	**[** $SlotDef^*_{,}$ **]**	
	Body	::=	$Sentence^*_{;}$ | $Sentence^*_{;}$ **$** $Sentence^*_{;}$	
	Sentence	::=	Sname **:=** Sname **clone** |	-- Clone sentence
			$Sname^+_{,}$ **:= (**Sname = Sname Sname Sname **)** |	-- If-sentence
			Sname **!** Sname **(** $Sname^*_{,}$ **)**	-- Message-send sentence
	SlotDef	::=	slotName → Val	
(!)	Val	::=	Oname | Sname | **this**	
	slotName	::=	Sname |	-- Plain slot
			:Sname |	-- Input slot
			Sname✯ |	-- Inheritance slot
			:Sname✯	-- Input and Inheritance slot
	Sname	::=	$char^+$ but no colon (:) first and no ✯ last and not equal to 'this'	
(+)	Oname	::=	$char^+$ not equal to 'this'	
	char	::=	**a** | … | **z** | **A** | … | **Z** | **0** | … | **9** | **+** | **-** | ***** | **/** | **_** | **:** | =	

5.3 Reliability Requirements

This section presents and motivates a set of reliability requirements. Each subsection presents some simple configurations and attempts to prove the substitution proposition for each simple case. However, we are able to show that in each of the examples there is a property or lack of property in the configurations which leads us to conclude that the proposition does not hold.

Each case starts by presenting the configurations. Then the actions from the configurations are presented before the case is discussed. The configurations are named A, B, C and D and the actions from these configurations are named as follows:

$\alpha \in$ action(A||D) and $\beta \in$ action(B||D) and $\gamma \in$ action(B||C) and $\delta \in$ action(A||C)

We assume $A \leq_D B$ and $C \leq_B D$ and then check if we have $A \leq_C B$ and $C \leq_A D$. In the examples below we do not have $A \leq_C B$ and $C \leq_A D$. To get reliable substitution we must therefore add requirements on how to define reliable Omicron configurations. Each subsection concludes by describing necessary requirements which must be fulfilled in order to prove the substitution proposition for situations similar to the subsection's example.

The sufficiency of the requirements is shown by the propositions and the theorems in chapter 6.

5.3.1 Inheritance slots

External inheritance in a refinement configuration means that an object in a refinement configuration inherits from an object in a specification configuration. For example an object in A inherits from an object in D as follows:

```
B = b : ([s->o, w->m, x->i], $s!w(x);)
A = a : ([q*->d], $s!w(x);)
D = d : ([s->o, w->m, x->i], ) || o: ([m->p], ) || p: ([:y->j], )
C = o: ([m->p], ) || p: ([:y->j], )
```

In the above case we have

α = a->o!m(i), β = b->o!m(i) and γ = b->o!m(i)

which gives α.dsc = β.dsc = γ.dsc and, $\alpha \in$ obs(D), $\beta \in$ obs(D) and $\gamma \in$ obs(C). This gives $A \leq_D B$ and $C \leq_B D$. Since A inherits from D there will also be inheritance from A to C. The action δ comes from A||C and we have δ = a->error since there is no object named "d" in A||C and therefore the slots are not found. To be able to ensure $C \leq_A D$, we must be able to ensure that δ will be an action which is not observable from A.

In this case it can not be ensured that δ will be an action observably similar to γ when α is observably similar to β. This is because the result of executing a sentence in A in A||C where there is inheritance from A to C, can not be known unless the internals of C is known. This is because we do not know where the slots referred to in the sentence in the object named "a" will be found unless the internals of C are known. The proposition assumption involving C is $C \leq_B D$. This only gives properties of the actions from C observable from B in B||C and not any information about C's internals. It is therefore not in general possible to determine what the action δ from execution of the sentence in the object named "a" will be. In the case when A is combined with C, δ will be an error action since there is no object named in C.

To show the substitution proposition in such cases where there is inheritance from A to D we have two choices:

- to disallow A to inherit from D. We call this *no external inheritance* in A.

- to state properties of internals of C which are used in relation to A inheriting from C in the premise of the substitution proposition. These properties must be stated relative to the internals of D which are used in relation to A inheriting from D.

The first property formally stated:

$$NoExt^*(A, D) == \forall A', D' \bullet A||D \xrightarrow{*} A'||D' \Rightarrow noExt(A', D')$$

where noExt is defined by

$noExt(A) == \forall o, s \bullet @A(o:s) \Rightarrow owner(A, o, s) \in A$

where owner(A, o ,s) returns the name of the object in the inheritance graph of o where a slot named s is found (defined in chapter 3).

Note that when there is no external inheritance in A, then all inheritance slots hold names of objects in A. Then A will not inherit from objects in C or any other objects not in A. However, since slot values can be changed during execution of the configurations, then NoExt*(A) can not be checked by looking at static properties of A.

The second property formally stated:

$$\text{SafeInheritance*}(A, C, D) == \forall A', C', D', A'', C'', D'', \overline{\alpha}, \overline{\delta}, \alpha, \delta \bullet$$
$$A||D \xrightarrow{\alpha} A'||D' \wedge A'||D' \xrightarrow{\overline{\alpha}} A''||D'' \wedge A||C \xrightarrow{\delta} A'||C' \wedge A'||C' \xrightarrow{\overline{\delta}} A''||C'' \wedge$$
$$\alpha.exe = \delta.exe \Rightarrow \text{SafeInheritance}(A', C', D')$$

where SafeInheritance is defined by

$$\text{SafeInheritance}(A, C, D) == \forall i, s \bullet i \in A \wedge s \in A.Slots(i) \Rightarrow (A||D)(i:s) = (A||C)(i:s)$$

where A.Slots(i) denote the names of the slots in the object named i in A
and (A||C)(i:s) denote the value of the slot names s as found in the inheritance graph of the object named i.

Both these formal definitions are a bit more general than strictly necessary, since they cover all slots, and not only those slots which are actually used in a given case. Even in the more specific cases, establishing SafeInheritance* would require knowledge of both A and C. We would then loose the compositionality property found and expressed in the substitution proposition. This property allows A and C (refinements) to be safely combined without first having to establish properties of the combined configuration A||C. However, by introducing the SafeInheritance* property this is no longer true, and the basic intention is broken. Therefore, this alternative is not taken as a reliability requirement. Instead the noExt()-function is taken as a reliability requirement for configurations which are to be reliable refinements.

The *no external inheritance* alternative means that objects in refinements cannot inherit from objects in specification configurations. This might at first glance be associated with inheritance from classes in class libraries. However, we talk here about *object inheritance* and not *class inheritance*. When classes are not changed during execution, then one may eliminate the problem with external inheritance. The problem is eliminated by considering any classes which are inherited between specifications and refinements as copied into the different configurations; one copy for each configuration.

We have then established that a requirement such as NoExt* is necessary if we want reliability without having to establish properties of the combined configuration A||C.

Observation O.5.3.1 : In configurations with no external inheritance, an action can only update slots within the configuration where the executed sentence is found

Observe that when we have no external inheritance in a configuration, all assignment actions from if-sentences in the configuration will be silent actions. This is because all owners of a slot referred to in a sentence will be found within the configuration where the sentence is found.

A consequence of this is that observable traces will not include assignment action from execution of sentences in refinement configurations.

Similarly clone actions from execution of sentences in configurations with no external inheritance will only update slots in the configuration. Such actions are observed only if they clone objects which are in the observer configuration. Otherwise, they are hidden.

When we have an action $\alpha \in \text{Traces}(A||D)$ and there is no external inheritance A and the action α is from execution of a sentence in A and is observable from objects in D, ie,

$\alpha \in \text{Traces}(A||D) \wedge \text{noExt}(A) \wedge \alpha.exe \in A \wedge \alpha \in \text{obs}(D)$

then the only kinds of action α can be is

a message-send action to an object in D or
an action which creates an object in D and updates a slot in A.

This is because there is no external inheritance, and therefore we can not have actions which update slots in other configurations.

Observation O.5.3.2 and shorthand notation : All updated slots come from the same part of the configuration

By observation O.5.3.1 (observed actions from a configuration with no external inheritance), an if-sentence in a configuration with no external inheritance can not update slots outside the configuration. This means that when we have actions of the form

α = e->$o_1.s_1 \ldots o_n.s_n$:= j

and e $\in$ A and there is no external inheritance in the configuration A, then all o_i will be object names in the same configuration, ie, $o_1 \ldots o_n \in$ A. Since all o_i are names in the same configuration then the notation may be simplified by denoting the actions e->o.s := j where o.s corresponds to $o_1.s_1 \ldots o_n.s_n$.

The next proposition shows an important property of applying a reliable substitution to a configuration with no external inheritance. It shows that no external inheritance is preserved when a configuration is specialised with a reliable refinement, eg, if we have noExt(C) and a reliable substitution σ, then we also have noExt(Cσ). The next proposition also shows that the same owner of a slot will be found in C and Cσ, ie, owner(C, i, s) = owner(Cσ, iσ, sσ). The proposition also shows that the value of a given slot will be equal whether the substitution is applied to the configuration as a whole, or to the value found in C, ie, Cσ(i:s) = (C(i:s))σ for all object names i in C and all slot names s in C.

Proposition P.5.3.1 Reliable substitutions give same slots and preserve "No external inheritance"

$\forall$ A, B, C, i, s, σ •
noExt(C) $\wedge$ $\sigma \in$ B $\rightarrow$ A $\wedge$ B||C, A||C $\in \mathscr{C}_{Safe} \wedge$ @C(i:s)
$\Rightarrow$
noExt(Cσ) $\wedge$
owner(C, i, s) $\in$ C $\wedge$ owner(C, i, s) = owner(Cσ, iσ, sσ) = owner(Cσ, i, s) $\wedge$
Cσ(i:s) = (C(i:s))σ

Proof:
Observation O.5.2.1 (observing objects' names are never keys in the substitution) and observation O.5.2.2 (reliable substitutions do not change slot names in configurations) give:
since i is an object name in C then i = iσ and
since s is a slot name in C then s = sσ.
This gives
owner(Cσ, iσ, sσ) = owner(Cσ, i, s).
noExt(C) gives that all object names in the inheritance graph of the object named i are objects in C. By observation O.5.2.1, these names are not changed by the substitution. We then have
owner(C, i, s) $\in$ C and owner(C, i, s) = owner(Cσ, i, s) and noExt(Cσ)
Then, by definition of slot lookup, the same slot is found for C(i:s) and Cσ(i:s). We then have
Cσ(i:s) = (C(i:s))σ
❑

5.3.2 If-sentences

Assume that we have the following case where we have an if-sentence in a method in C:

B = b_1 : ([m->p],) || p : ([],) ||
b_2 : ([],)

A = a : ([m->p],) || p : ([],)

D = d : ([s->b_1, t->b_2, w->m], $ s!w();)

C = c : ([s->b_1, t->b_2, w->m, x->q], $ s := (s=t x s); s!w();) ||
q : ([m->r],) ||
r : ([],)

The substitution:
σ = {a/b_1} {a/b_2}
is used to specialise C and D for combination with A, and the configurations fulfil the reliability requirements of all previous sections in this chapter.

When executing the configurations we then get the following actions and action sequences:

α = < d->a!m() >
β = < d->b_1!m() >
$\bar{\gamma}$ = < c->c.s:=b_1, c->b_2!m() >
$\bar{\delta}$ = < c->c.s:=q, c->q!m() >

In $\bar{\gamma}$ we have the action c->c.s:=b_1 because $b_1 \neq b_2$ and then c.s get the value of s which is b_1.
In $\bar{\delta}$ we have the action c->c.s:=q because a = a and then c.s get the value of x which is q.

We then have $\bar{\delta} \leq_C \bar{\gamma}$, $\bar{\gamma} \leq_B \beta$ and $\alpha \leq_D \beta$, but we do not have $\bar{\delta} \leq_A \alpha$. To avoid situations such as this we must require that every if-test in C will give the same result both when combined with A and B. This must also hold for derivations of C from A||Cσ and B||C.

Assume that we have an if-sentence s := (v=w t f). If C(i:t) = C(i:f) then it is not necessary to consider the values of C(i:v)σ and C(i:w)σ in order to ensure reliability. In the case where C(i:v) = C(i:w), applying the substitution will by definition give C(i:v)σ = C(i:w)σ something which ensures reliability. This leaves one case where it is necessary to set extra requirements to get reliability, namely when C(i:t) $\neq$ C(i:f) and C(i:v) $\neq$ C(i:w). To get reliability we must then require that C(i:v)$\sigma \neq$ C(i:w)σ.

To ensure C(i:v)$\sigma \neq$ C(i:w)σ we can either put requirements on the substitution σ or put requirements on the values of the v and w slots. If we put requirements on the substitution we must require that all values must be unique, since we have that all keys are unique. The result of this is that a refinement configuration must have the same number of visible objects as the specification configuration. This is not a preferred solution since it gives unnecessary restrictions on refinement configurations.

The requirement on slot values must ensure that when C(i:t) $\neq$ C(i:f) and C(i:v) $\neq$ C(i:w) then C(i:v)$\sigma \neq$ C(i:w)σ. We have that the substitution only substitutes, in this case, B object names and the values of the substitution are all A object names. If we also require that each slot values in C are either not the name of an object in A or it is the name of both an object in A and an object in B, then C(i:v)$\sigma \neq$ C(i:w)σ can be ensured by requiring that at least one of the values of v and w in C is different from object names in B, ie,

$C(i:v) \notin B \vee C(i:w) \notin B$

This reliability requirement can then be formulated as follows:

Definition: Reliable if-sentences in a configuration C when combined with a configuration B

We say that a configuration C has reliable if-sentences when combined with a configuration B if there are no executable if-sentences in C which will compare two slot values in C which are object names in B. This means that there are no if-sentences of the form $s_1,\ldots, s_n$:= (v=w t f) after an execution mark \$ where the values of both v and w are object names which are found in the configuration B. This can formally be defined:

RelIfSentence(C, B) ==
$\forall$ i : C, v, w : $\mathcal{N}$ •
($\exists$ $s_1,\ldots, s_n$, t, f, S_1, S_2 •
C(i).Body $\equiv$ S_1 \$ $s_1,\ldots, s_n$:= (v=w t f); S_2) $\Rightarrow$
(C(i:t) $\neq$ C(i:f) $\vee$ C(i:v) $\neq$ C(i:w)) $\Rightarrow$ (C(i:v) $\notin$ B $\vee$ C(i:w) $\notin$ B)

This must also hold for all derivations of B||C. This is handled below.

In practice this requirement means that if-sentences in a configuration should only compare names of objects which are found within the configuration itself. This requirement is not commonly used when making object oriented systems. Use and checking of this requirement is discussed in chapter 9 on related work.

5.3.3 "Message not understood" errors

Assume that we have a case where D gets a message with a message selector not matching any slot name in its inheritance graph:

B = b : ([s->o, w->n], \$s!w();)
A = a : ([s->o, w->m], \$s!w();)
D = o : ([],)
C = o: ([n->p],) || p : ([],)

This is often called a message not understood error.

In this case we have the actions α = a->error, β = b->error and γ = b->error. This gives A $\leq_D$ B and C $\leq_B$ D. We also have δ = a->o!n() which stem from execution of a sentence in A and is observable from C. Since γ is not observable from C, we do not have A $\leq_C$ B and the proposition do not hold for this case.

This problem is caused by an object in D receiving a message with a selector which does not match with a slot name in the inheritance graph of the receiver, while there in C is a matching slot. To avoid this problem, we can require that there will never be an error action in A||D caused by "method not found" when the receiver is an object in D. This means that an appropriate method will always be found when an object in D receives a message. We call this reliable message sending from A to D and define it formally as follows:

Definition: Reliable message sending from a configuration A when combined with a configuration D

We say that A has reliable message sending relative to D if, when an executing sentence is found in A in a system A||D, there will never be errors because a method object can not be found when the receiver is an object in D. This can formally be defined by

RelMessageSend(A, D) ==

$\forall$ i : A, t, w : $\mathcal{N}$ •

(($\exists$ S_1, S_2, $\bar{p}$ •

A(i).Body $\equiv$ S_1 \$ t!w($\bar{p}$); S_2) $\wedge$ A(i:t) $\in$ D) $\Rightarrow$ D(A(i:t):A(i:w)) $\in$ D

where A(i).Body means the sentences in the object named i in A.

An alternative to requiring reliable message sending would be to look at all message sends from A to D and compare them with message sends from A to C. This would mean that actions from A||C have to be considered in order to establish A $\leq_C$ B. This means that we would have to check the collaboration properties of A||C explicitly. As discussed in chapters 1 and 2, this is something we want to avoid so that components can be developed separately in space or time. This is one of the motivations for having reliable refinements and defining the substitution proposition. We therefore prefer the reliable message sending requirement.

The reliable message send requirement is linked with the error model. The error model defined for Omicron defines "message not understood" errors as not observed, even when the receiver is an observer. Alternative error models can eliminate the need for the reliable message send requirement. We have considered several alternatives which eliminate the need for this requirement. A typical example of such an error model is a model where error actions hold all information about the erroneous actions. In the case of message-send actions this means data about the receiver, the message selector and parameters corresponding to the names which would be found in a corresponding successful message send action, eg, the action e->error(o!m($\bar{p}$). Observability of such erroneous actions is defined as for observability of successful actions. In relation to the reliable message send requirement, an action where the receiver is an object in D would be observed by D. Observable equality of erroneous actions is defined as for non-error actions in order to get reliable behaviour of refinements and avoid the reliable message send requirement. This means that an error action from A which is a "message not understood" error with a receiver in D, is observably similar to an error action from B provided the receiver is the same D object and the message selector is equal in the two actions. Also, similarity requirements for the parameters must be the same as for message-send actions. This gives:

error(o!x($\bar{q}$)) $\sim_O$ error(p!y($\bar{p}$)) == o $\in$ O $\vee$ p $\in$ O $\Rightarrow$ <o, x> = <p, y> $\wedge$ $\bar{q}$ $\sim_O$ $\bar{p}$

The consequence of this is that reliable refinements of B and B must have observably equal "message not understood" error actions, particularly try to send messages with the same erroneous message selector to the same D objects. Viewing B as a specification, this would means that a specification will define what "message not understood" errors reliable refinements should create. In practice, such specification would not be very common. It can be assumed that it is more common to actually require also reliable message sending from the specification configuration B, ie, RelMessageSend(B, D), meaning that there will never be error actions in B due to method not found in D. However, note that requiring RelMessageSend(B, D) does not eliminate the need for requiring RelMessageSend(A, D) for a reliable refinement A. However, by the above definition of error actions and similarity of such actions, RelMessageSend(A, D) is ensured when the refinement relation holds and therefore does not have to be separately established.

The conclusion from trying other error models is that there are two alternatives to ensuring reliable behaviour in the case of "message not understood" errors. The first alternative is to require reliable message sending in reliable refinements. The other is to have specifications which specifies what "message not understood" errors reliable refinements should create, and this is usually none. It is then necessary to define an error model and define observability and observable equality of error actions in such a way that it is ensured that a specification and its

refinements create corresponding "message not understood" errors where the receiver is the same object and the message selectors are equal. Compared to the first alternative, the latter alternative gives much more complex definitions of error actions, observability and observable equality. Since the latter alternative also seems to be very uncommon in practise and rather seen as peculiar, the first alternative was chosen for the present work.

5.3.4 External methods

An external method for an action is a method which is found in an other configuration than the configuration where the receiver object of the action is found. For example an object in C receives a message and the method is found in A as follows:

B = p : ([],)
A = p : ([x->a, y->n], x!y()) || a : ([n->q],) || q : ([],)
D = d : ([],)
C = c : ([s->o, w->m], $s!w();) || o : ([m->p],)

Here we have $A \leq_D B$ and $C \leq_B D$. We have $A \leq_D B$ since A||D and B||D have no action and we have $C \leq_B D$ since D||B has no actions and B||C has a single action, γ = c->o!m()/k_p, which is silent in that is it from execution of a sentence in C and only observable from C.

We will also have an action δ = c->o!m()/k_p from A||C, which is not observable from A. The configuration derived from C and γ will be:

C' = c : ([s->o, w->m], s!w();$) || o : ([m->p],) || k_p : ([], $)

where k_p is a copy of the object names p in B. The configuration derived from C and δ will be:

C'' = c : ([s->o, w->m], s!w();$) || o : ([m->p],) || k_p : ([x->a, y->n], $x!y())

where k_p is a copy of the object named p in A.

There will be no next action from B||C', nor from B||D or A||D. However from A||C'' there will be a next action. In this case it will be a message-send-action, k_p->a!n()/k_q, which is observable from A. We would then not have $C \leq_A D$, even if we have $A \leq_D B$ and $C \leq_B D$. We can then conclude that the substitution proposition does not hold in this case.

This problem is created by the refinement C having external methods, ie, methods which are found in the context and not in the component itself. In order to avoid such problems, and thereby be able to prove the substitution proposition, we must require that all methods which are copied as result of an object receiving similar messages are similar in some way. One simple way to achieve this is to require that the methods are found locally in the component. This ensures that the methods are equal. We call this *reliable method lookup* and define it formally as follows:

Definition: Reliable method lookup in a configuration A when combined with a configuration D

We say that a configuration A has reliable method lookup when combined with a configuration D if for every message-send action from A||D where the receiver is in A, the method-object to be copied is an object in A. This can formally be defined by

RelMethodLookup(A, D) ==
$\forall$ e, o, m, $\bar{p}$ • e->o!m($\bar{p}$) $\in$ Traces(A||D) $\wedge$ o $\in$ A $\Rightarrow$ A(o:m) $\in$ A

where e->o!m($\bar{p}$)/k $\in$ Traces(A||D) means that e->o!m($\bar{p}$)/k is an action which is the result of executing a message-send sentence in the object named e in the configuration A||D

To get reliable substitution, we must also have reliable method lookup for all derivations of A||D. This is included in below definitions leading up to the definition of the refinement relation.

Other alternatives which will ensure that the methods are "similar enough" to ensure reliable substitution is left for further study.

The next proposition shows that when we have a message-send action where the receiver is an object in a configuration with reliable method lookup, below denoted A, the same method object is found in A and in Aρ when ρ is a reliable substitution.

Proposition P.5.3.2 The same method is found in a configuration and its specialisation

$\forall$ A, D, ρ, e, o, m, k, $\bar{p}$ •
RelMessageSend(A, D) $\wedge$ RelMethodLookup(A, D) $\wedge$ noExt(A) $\wedge$ A||D $\in \mathscr{C}_{Safe} \wedge \rho \in D \rightarrow C \wedge$
e->o!m($\bar{p}$)/k $\in$ Traces(A||D) $\wedge$ o $\in$ A $\Rightarrow$ A(o:m) $\in$ A $\wedge$ Aρ(o:m) = A(o:m)

Proof:
Since we have noExt(A) $\wedge$ A||D $\in \mathscr{C}_{Safe} \wedge \rho \in D \rightarrow C$ then proposition P.5.3.1 gives
@A(o:m) $\Rightarrow$ Aρ(o:s) = (A(o:m))ρ.
o $\in$ A $\wedge$ RelMessageSend(A, D) gives by definition of reliable message sending @A(o:m) and we then have
Aρ(o:s) = (A(o:m))ρ.
o $\in$ A $\wedge$ RelMethodLookup(A, D) gives by definition
A(o:m) $\in$ A.
A(o:m) $\in$ A $\wedge$ noExt(A) $\wedge$ A||D $\in \mathscr{C}_{Safe} \wedge \sigma \in D \rightarrow C$ gives A(o:m) = (A(o:m))ρ
and we then have
Aρ(o:s) = A(o:m)
ie, the same method object is found in both A and Aρ and the proposition holds.
❑

5.3.5 Summing up requirements on reliable configurations

The reliability requirements are named *no external inheritance*, *reliable method lookup*, *reliable if-sentences* and *reliable message sending*. How these relate to common practice is discussed in chapter 8. A brief summary is:

> The two first are commonly found in class-based languages, where classes can not change during runtime. "Reliable if-sentences" are not supported by any language or notation, but is common practice in, eg, Smalltalk and SELF. Reliable message sending is partially supported in most object-oriented programming languages.
>
> If the language for defining configurations does not include statements for assigning values to inheritance slots and slots referring to methods, then noExt and RelMethodLookup can be checked statically. If they hold in the initial configuration, the reliability requirements will continue to hold for the primed configurations since the inheritance and method slots can not be changed by execution of sentences in the objects. However, Omicron allows update of inheritance and method slots, so external inheritance and non-local methods can be introduced by execution of sentences. Reliability of message sending and if-sentences is not simple to check statically in a system where the slot values change as result of executing sentences. Results such as the work presented in (Palsberg and Schwartzback 1994), suggest that it might be possible to ensure reliable message sending by type inference. It may also be possible to check reliability of if-sentences through some form of inference and/or typing of slots. This is left for further study.

The reliability requirement on a possible refinement configuration is summed up in the following definition:

Definition: Reliable (A, D)

Reliable(A, D) ==
noExt(A) $\wedge$ RelIfSentence(A, D) $\wedge$
RelMessageSend(A, D) $\wedge$ RelMethodLookup(A, D)

The following predicate is introduced in order to be able to assert reliability for configurations which are the results of transitions from a configuration:

Definition: Reliable(A, D, $\bar{\alpha}$)

Reliable(A, D, $\bar{\alpha}$) == (A||D $\xrightarrow{\bar{\alpha}}$ A'||D' $\Rightarrow$ Reliable(A', D')

5.4 Observable Similarity

This section defines observable similarity of actions and action sequences. This observable similarity is not an equality relation since it is not commutative. We use the word similar to denote a non-commutative relation. This is how the word is used in definition of relations in the π-calculus by Miner et.al. The commutative relation is called bisimilar in the π-calculus tradition.

Observable similarity of action sequences will in a following section be used in the definition of a reliable refinement relation between configurations.

5.4.1 Observably equal names as parameters

What follows is an example of objects sending messages with object names as parameters. In the example the configuration named A sends more names to D and C than B does. This means that a possible refinement sends more names to the objects in the context then the specification component does. This makes it impossible to prove the substitution proposition and must therefore be avoided. The problem is avoided by strengthening the definition of observable equality as shown below.

Here is an example where the refinement A send more names to the contexts D and C than the specification B:

B = x : ([s->o, w->m, v->n, x->b], $s!w(x, x);) ||
b : ([m->p],) || p : ([],)

A = y : ([s->o, w->m, x->a, y->b], $s!w(x, y);) ||
a : ([m->p],) || p : ([],) ||
b : ([m->r],) || r : ([i*->y], s!w(x))

D = o : ([x->i, y->j, m->q],) ||
q : ([w->m, :v->nil, :w->nil], v!w(); v!w();)

C = o : ([x->i, m->q,],) ||
q : ([w->m, :v->nil, :w->nil], x!v(); x!w();)

Here A sends two names, a and b, where B sends only one, b.
The action sequence from A||D is:

$\overline{\alpha}$ = < y->o!m(a, b)/k_q, k_q->a!m()/k_p, k_q->a!m()/k'_p, >

and from B||D:

$\overline{\beta}$ = < x->o!m(b, b)/k_q, k_q->b!m()/k_p, k_q->b!m()/k'_p, >

and from B||C:

$\bar{\gamma}$ = < x->o!m(b, b)/k_q, k_q->b!m()/k_p, k_q->b!m()/k'_p, >

This gives A $\leq_D$ B as both $\overline{\alpha}$ and $\overline{\beta}$ have one action which is from execution of A and B sentences which are observable from D and the bodies of the actions are observably equal. This also gives C $\leq_B$ D as both $\bar{\gamma}$ and $\overline{\beta}$ have two action which is from execution of C and D sentences which are observable from B and the bodies of the actions are observably equal.

The action sequence from A||C is infinite:

$\bar{\delta}$ = < y->o!m(a, b)/k_q, k_q->a!m()/k_p, k_q->b!m()/k_r,
k_r->o!m(a,b)/k'_q, k'_q->a!m()/k'_p, k'_q->b!m()/k'_r,
k'_r->o!m(a,b)/k''_q,..... >

and we do not have A $\leq_C$ B or C $\leq_A$ D. Therefore the assumptions of the substitution proposition hold, while the conclusion does not hold for these configurations. The reason for this problem is that A sends more object names to C and D than B does. Thereby, A gets more visible object names than B relative to C.

As in some previous sections, we have two alternative approaches to get around this problem. One is to put requirements on the relations between internals of C and D and the other is to change the requirements on observably equal actions and action sequences. As argued previously, the preferred alternative is to put requirements on relations involving actions, rather than put requirements on configuration internals.

We therefore solve the problem by changing the definition of observably equal actions. The weak point in the current definition is the requirement on object names as parameters and new slot values. It is required that such objects names are observably equal relative to the names of the objects in the observing configuration. Particularly we have:

$$\alpha =_O \beta \wedge \alpha.dsc = o!x(\bar{q})/k \Rightarrow \beta.dsc = o!x(\bar{p})/k \wedge \bar{q} \sim_O \bar{p}$$

and the weak point is $\bar{q} \sim_O \bar{p}$ stating that parameters must be observably equal. In the above case we had:

y->o!m(a, b)/k_q and x->o!m(b, b)/k_q and this gives <a,b> $\sim_D$ <b, b> and therefore $\alpha =_D \beta$

Requiring all parameters to be equal would have the effect that the proposition could be shown to hold for the above cases. Another effect would be that C and D would have to interact with the same number of A and B objects, because when the parameters are to be equal, then we require that a refinement has the same number of visible objects as the specification.

The question is then if we need such a strong requirement. As shown below, it is possible to have a weaker requirement than the requirement for all parameters to be equal. However, the weaker requirement is strong enough to ensure that the substitution proposition holds for cases as the ones shown above.

The conclusion is that to be able to prove the proposition, a reliable refinement can not have more visible objects than the specification. However, as is shown by theorem T.6.1 and the propositions and proof leading up to this theorem, a refinement may have the same or fewer visible objects than the specification. Also, a refinement can have more, or less, objects as long as any additional objects are not visible. We could then have a situation where B has visible object names b_2 and b_3, while A has a single visible object name a.

However, we need more restrictions than this. We also need to require that whenever a given B-name appears as a parameter, then the corresponding A-name will appear. For example, whenever b_2 occurs as parameter then the A-name a_2 must occur in b_2's position. If we did not have this requirement we could have the following situation:

B = x : ([s->z, w->m, x->b_1, y-b_2, z->b_2], $s!w(x, y, z); ||
b_1 : ([m->p],) || p : ([],) ||
b_2 : ([m->r],) || r : ([],)

A = y : ([s->z, w->m, x->a_1, y->a_1, z->a_2], $s!w(x, y, z); ||
a_1 : ([m->p],) || p : ([],) ||
a_2 : ([m->r],) || r : [s->o, w->m], s!w())

D = z : ([m->q],) ||
q : ([:s->nil, :t->nil, :u->nil, w->m], s!w(); t!w())

C = z : ([m->q],) ||
q : ([:s->nil, :t->nil, :u->nil, w->m], s!w(); u!w())
o : ([m->s]) || s : ([],)

Then we would have:

$\bar{\alpha}$ = < y->z!m(a_1, a_1, a_2)/k_q, k_q>a_1!m()/k_p, k_q->a_1!m()/k'$_p$ >
$\bar{\beta}$ = < x->z!m(b_1, b_2, b_2)/k_q, k_q->b_1!m()/k_p, k_q->b_2!m()/k_r >
$\bar{\gamma}$ = < x->z!m(b_1, b_2, b_2)/k_q, k_q->b_1!m()/k_p, k_q->b_2!m()/k_r >
$\bar{\delta}$ = < y->z!m(a_1, a_1, a_2)/k_q, k_q->a_1!m()/k_p, k_q->a_2!m()/k_r, k_r->o!m() >

and we have $\bar{\delta} =_C \bar{\gamma}$, $\bar{\gamma} =_B \bar{\beta}$ and $\bar{\alpha} =_D \bar{\beta}$ but we do not have $\bar{\delta} =_A \bar{\alpha}$ since the last action in $\bar{\delta}$ is from a sentence in A and observable from C and there is no corresponding observably equal action in $\bar{\alpha}$. In this case we must require that whenever b_1 is a parameter in an action in $\bar{\beta}$ then we will always find a_1 in the

corresponding position in $\overline{\alpha}$. Also, a_1 will not be found in any other positions than the positions were b_1 is found. This relation between visible object names can be expressed by a name substitution.

We will then have a substitution associated with the definition of observably similar actions given below. This relation is an alternative to the observably equal actions relation. The name substitution associated with D will, as previously, be a function from B-object names to A-object names, eg, $\sigma \in$ (B.Dom $\rightarrow$ A.Dom). Before we define observable similarity of actions, we define what it means for a substitution to be reliable relative to a set of object names.

Definition: Reliable substitution relative to a set of object names

We say that the substitution σ is reliable relative to a set of names O, denoted RelSubst(σ, O), if the following hold:

RelSubst(σ, O) == $O \cap names(\sigma) = \emptyset$

This definition says that a reliable substitution will never substitute a name in O and will never have a value which is a name in O.

Next we observe that a reliable substitution relative to three configurations, as defined in section 5.2.2 is also reliable relative to a set of object names which are the names of the objects in the observing context. This shows that the definition of observably similar actions found below, which refer to a substitution which is reliable relative to a set of object names, will also be applicable when the set of object names are replaced with the names of objects in the observing context.

Observation O.5.4.1 : Reliable substitutions relative to configurations are also reliable relative to sets of object names

When we have configurations B, D and A, D with safe names, ie, A||D, B||D $\in \mathscr{C}_{Safe}$, and a substitution which is reliable relative to A, B and D, ie, $\sigma \in$ B $\rightarrow$ A then we also have RelSubst(σ, D.Dom) because:

RelSubst(σ, A, B, D) gives $\sigma \in$ B $\rightarrow$ A and

B||D $\in \mathscr{C}_{Safe}$ gives B.Dom $\cap$ D.Dom = $\emptyset$ and

A||D $\in \mathscr{C}_{Safe}$ gives A.Dom $\cap$ D.Dom = $\emptyset$.

Then we have D.Dom $\cap$ names(σ) = $\emptyset$ which gives RelSubst(σ, D.Dom).

5.4.2 Slot names as parameters in messages

In this section we will look at an example where the objects send messages with parameters values which are slot names. It is shown that if such parameter values may differ in observably equal action then either we have to put complex requirements on the use of names in refinements of configurations or it is impossible to prove the substitution proposition. For instance will this problem occur when we have the following configurations sending messages with slot names as parameters:

B = b_1 : ([s->o, w->m, x->b_2, y->m], $s!w(x, y); ||
b_2 : ([m->p],) || p : ([],) ||
m : ([],)

A = a_1 : ([s->o, w->m, x->a_2, y->n], $s!w(x, y); ||
a_2 : ([m->p, n->q],) || p : ([],) || q : ([],)

D = o : ([m->p],) ||
p : ([:s->nil, :t->nil, w->m], s!w();)

C = o : ([m->p],) ||
p : ([:s->nil, :w->nil,], s!w();)

We would then get the action sequences where the parameter values m and n are also slot names:

$\overline{\alpha}$ = < a_1->o!m(a_2, n)/k_p, k_p->a_2!m()/k >
$\overline{\beta}$ = < b_1->o!m(b_2,m)/k_p, k_p->b_2!m()/k >
$\overline{\gamma}$ = < b_1->o!m(b_2, m)/k_p, k_p->b_2!m()/k >
$\overline{\delta}$ = < a_1->o!m(a_2, n)/k_p, k_p->a_2!n()/k >

We then have $\bar{\delta} =_C \bar{\gamma}$, $\bar{\gamma} =_B \bar{\beta}$ and $\bar{\alpha} =_D \bar{\beta}$, but we do not have $\bar{\delta} =_A \bar{\alpha}$ since we have the last action k_p->a_2!m()/k and k_p->a_2!n()/k with different message selectors. This does not give reliable substitution.

One alternative to ensure reliability is to require that for each pair of message selectors m, n which are found as parameters in corresponding positions in corresponding messages from A in A||D and B in B||D respectively (or in any derivations of A||D and B||D) and where n ≠ m, then n and m can not be names in C. This requirement on C is expressed by the configurations A, B and D. This requires looking at all message sends from A to D and compare them with message sends from A to C. This would mean that actions from A||C have to be considered in order to establish $A \leq_C B$. This means that we would have to check the collaboration properties of A||C explicitly. As discussed in chapters 1 and 2, this is something we want to avoid so that components can be developed separately in space or time. This is one of the motivations for having reliable refinements and defining the substitution proposition. This alternative is therefore not chosen in the present work.

The alternative requirement which does not involve C is:

> any name used as parameter in message-send actions from B,
> and which are not names of objects in B or C
> must not be found as slot name or slot value in C.

If this holds and the requirement on names from the above subsection holds, all parameters which are not C-names can differ in messages from B and its reliable refinements. Differences in message parameters must comply with rules similar to those for differences in A and B object names as parameters. This means that we need to require that whenever a given name from B appears, then the corresponding name from A will appear. This can be expressed by using a substitution as is done for differences in object names. A may then send the same number or fewer different names to C than B does. It is also necessary to require that such names are never used in if-sentences if A may send fewer different names to C than B does.

To include the above requirement in the definitions and proofs leading up to the theorem showing the substitution proposition would mean introducing a new substitution or expand the object name substitution to also include other names which are found as parameters in messages from B to C. It also means making a change in the definition of reliable if-sentences. Some attempts at including this in the definitions and proofs were done in relation to this thesis. The definitions and proofs became substantially more complex with the expansion than without. Since it seemed that the proof of reliable substitution could be done by adding these requirements, and these requirements very similar to the requirements for differences in object names, while they added substantially more complexity, this requirement was left for further study. In stead the simple requirement:

> Two parameters must be equal if
> they are found in corresponding positions in corresponding observed message send actions from B and A
> and
> the parameters are *not* names of objects in B and A respectively.

This will be taken into account when we define a reliable version of observable equality of actions. The drawback of this solution is only that different message selectors can not be sent from B and A to D and C in order to get different messages back at a later stage in the execution of the system. In practise this is used in creating so called pluggable editors, originally developed and much used in Smalltalk. Later it has also been possible to make pluggable editors in Java. Pluggable editors are discussed further in chapter 8.

This is an example of the trade-off between complex reliability requirements and flexible components. This is further discussed in chapter 8.

5.4.3 Observable similarity of actions

Observable similarity of actions, denoted $\alpha \sim_{O,\sigma} \beta$, is defined as follows:

Definition: Observable similarity relative to a set of object names and a reliable substitution

Given a set of object names O and a substitution σ which is reliable relative to the set of object names O, denoted, RelSubst(σ, O), we define *observable similarity* as follows:

$o!x(\bar{q})/k \sim_{O,\sigma} p!y(\bar{p})/l$ == $o \in O \vee p \in O \Rightarrow \langle o, x, k\rangle = \langle p, y, l\rangle \wedge \bar{q} = \bar{p}\sigma$

$\overline{o.s} := i \sim_{O,\sigma} \overline{p.t} := j$ == $\# \overline{o.s} = \# \overline{p.t} \wedge$
$\forall\ i \leq \# \overline{o.s} \bullet o_i \in O \vee p_i \in O \Rightarrow o_i.s_i = p_i.t_i \wedge i = j\sigma$

$i.s:=k/o \sim_{O,\sigma} j.t:=l/p$ == $(i \in O \vee j \in O \Rightarrow i.s = j.t \wedge k = l\sigma) \wedge$
$(o \in O \vee p \in O \Rightarrow \langle o, k\rangle = \langle p, l\rangle)$

error $\sim_{O,\sigma}$ error == true

$\alpha \sim_{O,\sigma} \beta$ == $\alpha.exe \sim_O \beta.exe \wedge \alpha.dsc \sim_{O,\sigma} \beta.dsc$

Note that we only require weak equality for the .exe-parts of observably similar actions.

Compared to the definition of observably equal actions we here have $\bar{q} = \bar{p}\sigma$ instead of $\bar{q} \sim_O \bar{p}$, $i = j\sigma$ instead of $i \sim_O j$ and $k = l\sigma$ instead of $k \sim_O l$. This means that names in observably similar actions which are not slot names are equal relative to the substitution σ.

Note that this relation is not an equality relation since it is not commutative in that
we do not have $\langle\alpha\rangle \leq_{O,\sigma} \langle\beta\rangle \Rightarrow \langle\beta\rangle \leq_{O,\sigma} \langle\alpha\rangle$
This is because we will not necessarily have, eg, $\bar{q} = \bar{p}\sigma \Rightarrow \bar{p} = \bar{q}\sigma$.

Proposition P.5.4.1 : Observable similarity is transitive

Observable similarity is transitive in that

$$\alpha \sim_{O,\sigma} \beta \wedge \beta \sim_{O,\rho} \gamma \Rightarrow \alpha \sim_{O,\rho\sigma} \gamma$$

Proof:
The relation is transitive because it is defined using equivalence relations and we have for any names e, f and g:
$e = f\sigma \wedge f = g\rho \Rightarrow e = g\rho\sigma$
Then we have for all names in α and γ which are parameters, new slot values or names of new clones are equal relative to the substitutions $\rho\sigma$, and the proposition holds.
❑

Note that the relation is reflexive, ie, $\alpha \sim_{O,\sigma} \alpha$, provided the substitution is empty or the names in α are not found as keys in the substitution σ.

What follows are some observations related to observably similar actions and actions which are equal relative to a reliable substitution. These observations are used later in proofs leading up to the proof of the substitution proposition in chapter 6.

Observation O.5.4.2 : Properties of names in actions which are equal relative to a reliable substitution

Given three configurations with safe names, A||D, B||D $\in \mathscr{C}_{Safe}$, and a reliable substitution $\sigma \in B \rightarrow A$ and two actions $\alpha \in$ Traces(A||D) and $\beta \in$ Traces(B||D).

When we have $\alpha \sim_{D,\sigma} \beta$ or $\alpha \equiv \beta\sigma$, then by definition of the relations, the following holds for names in the actions:

When $<p_1,\ldots,p_n> = \alpha$.names and $<q_1,\ldots,q_n> = \beta$.names, then

(*) all names which are not A or B-names are equal $p_i \notin A \vee q_i \notin B \Rightarrow p_i = q_i$
all A-names in α are B-names in β and vice versa: $p_i \in A \Leftrightarrow q_i \in B$

Since we have (*), then particularly we have:
all D-names are equal in the actions $p_i \in D \vee q_i \in D \Rightarrow p_i = q_i$

Observation O.5.4.3 : Equal actions relative to a substitution are observably similar

Given two actions α and β, a substitution σ and a set of object names O. By looking at the definition of observably similar actions we see that when we have equal actions $\alpha \equiv \beta\sigma$ and we have RelSubst(σ, O), ie, σ does not substitute object names in the set of object names O or give values which are in the set, then the two actions are observably similar relative to the object names in O and the substitution σ. Provided *the substitution does not substitute slot names*, we then have:

$$\text{RelSubst}(\sigma, O) \wedge \alpha \equiv \beta\sigma \Rightarrow \alpha \sim_{O,\sigma} \beta.$$

When we have O = D.Dom and A||D, B||D $\in \mathscr{C}_{Safe}$, and $\sigma \in B \rightarrow A$ then by observation O.5.4.1 (reliable substitutions relative to a configuration are also reliable relative to sets of object names) we also have RelSubst(σ, D.Dom) which gives

$$\alpha \equiv \beta\sigma \Rightarrow \alpha \sim_{D,\sigma} \beta.$$

5.4.4 Reliability of configuration specialisation

Assume that we have the following configurations with observably similar actions and where the parameters are equal relative to a substitution σ = {a/b} (which gives $\sigma \in B \rightarrow A$):

B = b : ([m->q],) || q : ([],)
A = a : ([m->q],) || q : ([],)
D = o : ([x->b, w->m], $x!w();)
C = p : ([y->b, w->m], $x!w();)

Then we get the following action sequences:

Traces(B||D) = < o->b!m()/k_q >
Traces(A||Dσ) = < o->a!m()/k_q >
Traces(B||C) = < o->b!m()/k_q >
Traces(A||Cσ) = < o->a!m()/k_q >

We then have $C \leq_B D$, $A \leq_D B$, $C \leq_A D$ and $A \leq_C B$. However, if we did not use the same substitution to specialise both D and C to collaborate with A, then we would not be able to prove that $C \leq_B D \wedge A \leq_D B \Rightarrow C \leq_A D \wedge A \leq_C B$ since the relationship between the A-names in D and C would not be known. When we use the same substitution we will in the above case have:

$$D(o:x) \in B \wedge D(o:x) = C(p:y) \wedge D\sigma(o:x) \in A \Rightarrow D\sigma(o:x) = C\sigma(p:y)$$

To be able to prove this, we must show that all A-names in Cσ are related to the B-names in C as follows:

(*) $\forall$ i, s • @C(i:s) $\Rightarrow$ Cσ(i:s) = (C(i:s))σ

When we have no external inheritance in C, configurations with safe names and a reliable substitution, this holds by proposition P.5.3.1. However, there are cases when it is not enough that P.5.3.1 holds in order to prove $C \leq_B D \wedge A \leq_D B \Rightarrow C \leq_A D \wedge A \leq_C B$. An example is the following configurations where there are errors due to unknown message receivers in B||D, A||Dσ and B||D, while a receiver is found in A||Cσ:

B = b : ([m->q],) || q : ([],)
A = a : ([m->q],) || q : ([],)
D = o : ([x->r, w->m], $x!w();)
C = p : ([y->a, w->m], $x!w();)

we get the following actions:

Traces(B||D) = < o->error >
Traces(A||Dσ) = < o->error >
Traces(B||C) = < p->error >
Traces(B||D) = < p->a!m()/k_q >

We then have C $\leq_B$ D, A $\leq_D$ B. However, we do not have C $\leq_A$ D since the actions p->a!m() and o->error are not observably similar, and we do not have A $\leq_C$ B since the actions p->a!m()/k_q and p->error are not observably similar.

In order to ensure that the substitution proposition holds, we must set requirements on the A and B object names in C and D. This means that there must be some relationship between the object names in a specification D and the refinements of D, if the refinements are to be reliable refinements such that the substitution proposition holds. The object names which must be related are those names which are names of objects in other configurations. In this case these are the A- and B-names found in C and D.

This problem emerges from the fact that C knows more A-names than B-names. In this case we have Cσ(i:s) = (C(i:s))σ as above, we even have Cσ(i:s) = C(i:s), but we also have:

C(p:y) $\notin$ Bρ||C giving p->error
Cσ(p:y) $\in$ A giving p->a!m()/k_q

To get C $\leq_A$ D and A $\leq_C$ B we must ensure that
when D(o:x) $\notin$ B||D, C(p:y) $\notin$ Bρ||C and Dσ(o:x) $\notin$ A||Dσ, as in the above case,
then Cσ(p:y) $\notin$ A.

This can only be achieved by knowing the relationships between the B-names in C and D and the A-names in Cσ and Dσ. To show this, all A-names in actions from Cσ must be the result of substituting the B-names in actions from C, ie, $\delta \equiv \gamma\sigma$, and also requiring that all A-names in actions from Dσ are substitutions from B-names, ie, $\alpha \equiv \beta\sigma$, where the B-names in the actions from C and D are equal since the actions are observably equal relative to B, ie, $\gamma \sim_B \beta$. This means showing $\alpha \equiv \beta\sigma$ and $\delta \equiv \gamma\sigma$, which gives $\delta \sim_A \alpha$, ie, we must show:

$\forall\ \alpha$: Traces(A||Dσ), β : Traces(B||D), γ : Traces(B||C), δ : Traces(A||Cσ) •

α.exe = β.exe $\in$ D $\wedge$ γ.exe = δ.exe $\in$ C $\wedge$ $\gamma \sim_B \beta \wedge \gamma, \beta \in$ obs(B) $\wedge\ \alpha \in$ obs(A)

$\Rightarrow$ $\alpha \equiv \beta\sigma \wedge \delta \equiv \gamma\sigma$

To show this, the definitions of a relation which is a reliable version of observable equality of configurations has to take into account the A- and B-names which are found in C, and correspondingly, the C- and D-names found in A.

First we discuss how to show $\alpha \equiv \beta\sigma$ while further below, after the definition of observable similarity of action sequences, we discuss showing $\delta \equiv \gamma\sigma$.

Instead of redefining observable equality, requirements can be put on the use of A and B names in D. These requirements must be strong enough to ensure $\alpha \equiv \beta\sigma$ for all actions from D. As discussed before, it is preferred to strengthen the definition of the relations rather than putting requirements on configurations. Therefore, the new version of observable equality will include a case:

$\alpha \in$ Traces(A||Dσ) $\wedge\ \beta \in$ Traces(B||D) $\wedge\ \alpha$.exe = β.exe $\in$ D $\wedge\ \alpha \in$ obs(A) $\Rightarrow\ \alpha \equiv \beta\sigma$

When α.exe = β.exe $\in$ D $\wedge\ \alpha \in$ obs(A), this is a case where α and β are not observable from D. We then require for reliable observable similarity of actions that:

if α.exe $\in$ D $\wedge\ \alpha \in$ obs(O) then $\alpha \sim_{O,\sigma} \beta$ else $\alpha \equiv \beta\sigma$

When we filter out all observed actions, as is done in $\bar{\alpha}$/obs(O) in the definition of observable equality, then such actions will not be included in the result action sequence. Therefore, in order to define a reliable refinement relation we do not filter out when comparing action sequences. We avoid filtering them out by including the actions which are from execution of sentences in the observers in the compared action sequences. In the definition of the relation we therefore use the following notation:

$\bar{\alpha}$/obs&exe(O) where obs&exe(O) = obs(O) $\cup$ { α | α.exe $\in$ O}

This denotes a sequence of actions which consists of all the action in $\bar{\alpha}$ which are O-observed and/or from O and where the actions are found in the same order as in $\bar{\alpha}$.

5.4.5 Observably similar action sequences

Based in the preceding discussions we get the following definition of observably similar action sequences:

Definition: Observably similar action sequences relative to a substitution; $\bar{\alpha} \leq_{O,\sigma} \bar{\beta}$

An action sequence $\bar{\alpha}$ is said to be observably similar to an action sequence $\bar{\beta}$ relative to a set of object names O and a substitution σ, denoted $\bar{\alpha} \leq_{O,\sigma} \bar{\beta}$, if the following holds:

$\bar{\alpha} \leq_{O,\sigma} \bar{\beta}$ == $\bar{\alpha}$/obs&exe(O) $\approx_{O,\sigma}$ $\bar{\beta}$/obs&exe(O)

where $\bar{\alpha} \approx_{O,\sigma} \bar{\beta}$ == $\forall$ i $\leq$ # $\bar{\alpha}$ • *if* $\alpha_i \in$ obs(O) *then* $\alpha_i \sim_{O,\sigma} \beta_i$ *else* $\alpha_i \equiv \beta_i\sigma$

In the rest of this thesis $\alpha \leq_{O,\sigma} \bar{\beta}$ is used as short hand notation for <α> $\leq_{O,\sigma} \bar{\beta}$ while $\alpha \leq_{O,\sigma} \beta$ is used as short hand notation for <α> $\leq_{O,\sigma}$ <β>.

We let priming the observer names in the relation, eg, $\bar{\alpha} \leq_{O',\sigma} \bar{\beta}$ mean that the prime O' is the object names in O and all new names in the action sequences relative to the object names in O, eg, in the example we have O' = O $\cup$ NewNames($\bar{\alpha}$, O).

Furthermore, $\bar{\alpha} \leq_{D,\sigma} \bar{\beta}$ is used as short hand for $\bar{\alpha} \leq_{D.Dom,\sigma} \bar{\beta}$ where D is the name of some configuration and $\bar{\alpha} \leq_{D',\sigma} \bar{\beta}$ means that the observers are D' = D.Dom $\cup$ NewNames($\bar{\alpha}$, D).

Proposition P.5.4.2 The "Observably similar action sequence relation" is transitive
The Observably similar action relation is transitive since:

$\forall$ $\bar{\alpha}$, $\bar{\beta}$, $\bar{\gamma}$, O, σ, ρ • $\bar{\alpha} \leq_{O,\sigma} \bar{\beta} \wedge \bar{\beta} \leq_{O,\rho} \bar{\gamma} \Rightarrow \bar{\alpha} \leq_{O,\rho\sigma} \bar{\gamma}$

Proof:
This proposition follows directly from the definition and transitivity of the underlying relations.
❑

Note that when the substitution σ does not substitute any of the names in an action sequence $\bar{\alpha}$ then we trivially have:

$\bar{\alpha} \leq_{D,\sigma} \bar{\alpha}$

which gives reflexivity of the relation in this special case. When the observably similar action sequence relation is used to define a reliable refinement relation between configurations, then the normal case where reflexivity is expected, the substitution will not substitute names in the action sequences. See the discussion in relation to reflexivity of the reliable refinement relation defined in section 5.5.

Observation O.5.4.4 below shows relationships between two observably similar actions and their observability.

Observation O.5.4.4 : Observability of observably similar actions

Assume that we have $\sigma \in B \rightarrow A$ and $\alpha \leq_{O,\sigma} \beta$, $\alpha \sim_{O,\sigma} \beta$ or $\alpha = \beta\sigma$ and A||D, B||D $\in \mathscr{C}_{Safe}$. Then by observation O.5.4.2 (properties of names in observably similar action sequences) and by the definition of observably similar action sequences we have:

If either one of the actions are observed, then so is the other : $\alpha \in obs(O) \Leftrightarrow \beta \in obs(O)$
If one of the actions is from execution of a sentence in an observing object, then so is the other, and they are from execution of a sentence in the same object : $\alpha.exe \in O \vee \beta.exe \in O \Leftrightarrow \alpha.exe = \beta.exe$

Next we observe that when $\alpha \leq_{D,\sigma} \overline{\beta}$ then there is only one action in $\overline{\beta}$ which is observably similar to α, and the other actions are hidden.

Observation O.5.4.5 : Properties of similar observable action sequences

Assume that we have $\alpha \leq_{D,\sigma} \overline{\beta}$ where $\overline{\beta} = <\beta_1,...,\beta_n>$, $\sigma \in B \rightarrow A$ and A||D, B||D $\in \mathscr{C}_{Safe}$. Then by definition of observable similarity of action sequences we have:

$\exists\, i \bullet <\beta_1,...,\beta_{i-1}> \otimes D \wedge \alpha \leq_{D,\sigma} \beta_i \wedge <\beta_{i+1},...,\beta_n> \otimes D$

When we have $<\alpha> \in Traces(A||D\sigma)$ and $\exists\, \overline{\beta}$: $Traces(B||D) \bullet \alpha \leq_{D,\sigma} \overline{\beta}$ then we can assume that

$<\beta_1,...,\beta_i> \in Traces(B||D) \wedge \alpha \leq_{D,\sigma} <\beta_1,...,\beta_i>$

and

$<\beta_1,...,\beta_{i-1}> \otimes D \wedge \alpha \leq_{D,\sigma} \beta_i$

By observation O.5.4.4 (observability of observably similar actions) we also have:

$\alpha \in obs(D) \Leftrightarrow \beta_i \in obs(D)$
$\alpha \notin obs(D) \Leftrightarrow \beta_i \notin obs(D)$

Next we show that when we have observably similar actions, then the names of the objects in the two different derived context configuration are equal.

Proposition P.5.4.3 : Equal domains of derived configurations

$\forall\, \alpha, \overline{\beta}, A, B, D, \sigma \bullet \alpha \leq_{D,\sigma'} \overline{\beta} \wedge A||D\sigma \xrightarrow{\alpha} A'||D' \wedge B||D \xrightarrow{\overline{\beta}} B'||D'' \Rightarrow D'.Dom = D''.Dom$

Proof:
Case $\alpha \notin obs(D)$:

Observation O.5.4.5 (properties of similar observable action sequences) gives $\overline{\beta} \notin obs(D)$. Thus there is no creation of any object in D and this gives D.Dom = D'.Dom = D''.Dom.

Case $\alpha \in obs(D)$:

Observation O.5.4.5 (properties of similar observable action sequences) gives for $\overline{\beta} = <\beta_1,...,\beta_n>$:

$\exists\, i \bullet <\beta_1,...,\beta_{i-1}> \otimes D \wedge \alpha \leq_{D,\sigma} \beta_i \wedge <\beta_{i+1},...,\beta_n> \otimes D \wedge \beta_n \in obs(D)$

Any new objects in D' and D'' are created by D-observable actions. Then, the names of the new objects are found in the actions α and β_i.

If the actions are clone actions: By definition of observably similar actions, these D-names must be equal in the two actions. Then the names of the new objects in D' and D'' must be equal, and this gives D'.Dom = D''.Dom.

If the actions are message-send actions to an object in D: Then there will be α and β_i actions which meet the above requirements and where the name of the new method copy in D' and D'' is the same name. Then the names of the new objects in D' and D'' are equal, and this gives D'.Dom = D''.Dom.

❑

5.4.6 Reliable names in refinement configurations

The next step in showing $\delta \sim_A \alpha$ is to show $\delta \equiv \gamma\sigma$ when we assume the same properties of actions as in the above example:

$\alpha.exe = \beta.exe \in D \wedge \gamma.exe = \delta.exe \in C \wedge \gamma \sim_B \beta \wedge \gamma, \beta \in obs(B) \wedge \alpha \in obs(A) \wedge \alpha \equiv \beta\sigma$

Since we assume $\gamma.exe = \delta.exe \in C$ and also noExt(C), the same slot values will be found and as shown in the above case we then have $C\sigma(i:s) = C(i:s)$. However, the problem with showing $\delta \equiv \gamma\sigma$ came from cases where the action from C was an error action while the action from $C\sigma$ in $A||C\sigma$ gave a message-send action. This was because we had:

$C(p:y) \notin B\rho||C$ giving p->error
$C\sigma(p:y) \in A$ giving $p->a!m()/k_q$

Note that we could also have a corresponding case where γ from C is an error action and where the action δ from $C\sigma$ is a clone action, eg, c.s:=k/a. To avoid such problems we must ensure that all A-names from $C\sigma$ are results of substituting B-names in C or equal to B-names. Therefore we must require that all values in C which are A-names are also B-names, ie,

$C.Values \cap A.Dom \subseteq C.Values \cap B.Dom$

When combining C and A we must therefore require that they have reliable names which is defined as follows:

Definition: RelNames in A, B and C

Given three configurations A, B and C. We say that these configurations have reliable names if A and C have safe names and all A-names found as values in C are also names of objects in B. This can formally be defined:

RelNames(A, B, C) ==
$A||C \in \mathscr{C}_{Safe} \wedge C.Values \cap A.Dom \subseteq C.Values \cap B.Dom$

This requirement, which concerns two refinement configurations and a specification, will be used as a necessary assumption in the final formulation of the substitution proposition, however, it will not be used in the definition of a reliable refinement relation.

The next proposition shows that each B-names in every actions from execution of sentences in C in B||C will be equal to or replaced by an A-name when the same sentence is executed in $A||C\sigma$. We also have that every name in C which is not in B will be equal in $C\sigma$ and it will not be a name in A. We call this a *complete specialisation* of C with σ relative to A.

Proposition P.5.4.4 Reliable names in configurations and reliable substitutions preserve configuration names

$\forall A, B, C, o, s, \sigma \bullet RelNames(A, B, C) \wedge \sigma \in B \rightarrow D \wedge @C(o:s) \wedge$
$(C(o:s) \notin B \Rightarrow (C(o:s))\sigma \notin A \wedge C(o:s) = (C(o:s))\sigma)$

Proof:
RelNames(A, B, C) gives $C.Values \cap A.Dom \subseteq C.Values \cap B.Dom$. Then any value C(o:s) which is not a B-name will neither be an A-name. Then, since the substitution only maps from B-names to A-names, $(C(o:s))\sigma$ will not be an A-name. Also when C(o:s) is not in B, then it is not a key in the substitution and we then have $C(o:s) = (C(o:s))\sigma$ and the proposition holds.
❑

5.4.7 Reliability is preserved by substitutions

This section shows that when configurations have reliable names, then reliability of if-sentences is preserved when specialising with a reliable substitution.

The first proposition shows that we will get the same result of an if-test before and after the application of a reliable substitution to a reliable configuration.

Proposition P.5.4.5 Same result of if-test when applying a reliable substitution

When C(i).Body has the form S_1 \$ s_1,…, s_n := (v=w t f); S_2 then

$\forall$ A, B, C, i, v, w, σ •
B||C, A||C $\in \mathscr{C}_{Safe} \wedge$ noExt(C) $\wedge \sigma \in$ B $\rightarrow$ A $\wedge$ RelNames(A, B, C) $\wedge$ RelIfSentences(C, B) $\wedge$

@C(i:v) $\wedge$ @C(i:w) $\Rightarrow$ (C(i:v) = C(i:w) $\Leftrightarrow$ Cσ(i:v) = Cσ(i:w))

Proof:
By proposition P.5.3.1 (reliable substitutions give same slots and preserve "No external inheritance") we have
Cσ(i:v) = C(i:v)σ.
Similarly for w. To prove the proposition we show that the v=w test gives the same results in both C and Cσ. We have reliable if-sentences in A which gives C(i:v) $\notin$ B $\vee$ C(i:w) $\notin$ B. We must then show:

(*) (C(i:v) $\notin$ B $\vee$ C(i:w) $\notin$ B) $\Rightarrow$ (C(i:v) = C(i:w) $\Leftrightarrow$ C(i:v)σ = C(i:w)σ)

We then show
(1) (C(i:v) $\notin$ B $\vee$ C(i:w) $\notin$ B) $\wedge$ C(i:v) = C(i:w) $\Rightarrow$ C(i:v)σ = C(i:w)σ
and
(2) (C(i:v) $\notin$ B $\vee$ C(i:w) $\notin$ B) $\wedge$ C(i:v) $\neq$ C(i:w) $\Rightarrow$ C(i:v)$\sigma \neq$ C(i:w)σ

Proof of (1):
Since all keys in σ are B-names, and since C(i:v) = C(i:w) and at least one of the values is not in B, then neither is in B and then we have C(i:v) = C(i:v)σ and C(i:w) = C(i:w)σ which gives C(i:v)σ = C(i:w)σ. Then (1) holds.

Proof of (2):
Cases:
2a) C(i:v) $\notin$ B $\wedge$ C(i:w) $\notin$ B
2b) C(i:v) $\in$ B $\wedge$ C(i:w) $\notin$ B, by symmetry this also shows the case where C(i:v) $\notin$ B $\wedge$ C(i:w) $\in$ B
2a)
If neither of the actions are names in B, then since all keys in σ are B-names we then get C(i:v) = C(i:v)σ and C(i:w) = C(i:w)σ. Then when C(i:v) $\neq$ C(i:w) we also have C(i:v)$\sigma \neq$ C(i:w)σ and then (2) holds.

2b)
Assume that C(i:v) $\in$ B and C(i:w) $\notin$ B. We then have, as argued above C(i:w) = C(i:w)σ. Since RelNames(A, B, C) gives that all A-names in C are also B-names we then have C(i:w)$\sigma \notin$ A.
When C(i:v) $\in$ B then we have one of the following two cases:
C(i:v) is not a key in σ
then we have C(i:v) = C(i:v)σ and then C(i:v)$\sigma \neq$ C(i:w)σ. Then (2) holds.
C(i:v) is a key in σ
Then we have C(i:v) $\neq$ C(i:v)σ and C(i:v)$\sigma \in$ A. Since C(i:w) is not an A-name, then we have C(i:v)$\sigma \neq$ C(i:w)σ and then (2) holds.
❑

Next we show that when we have RelNames(C, D, A) and there is no external inheritance in A, the substitution ρ substitutes from D names to C names and also A and D have safe names, then RelIfSentences(A, D) implies RelIfSentences(Aρ, Cσ).

Proposition P.5.4.6 Reliable substitutions preserve reliable if sentences

$\forall$ A, B, C, D, σ, ρ •
RelNames(C, D, A) $\wedge$ noExt(A) $\wedge \sigma \in$ B $\rightarrow$ A $\wedge \rho \in$ D $\rightarrow$ C $\wedge$ A||D, B||C $\in \mathscr{C}_{Safe} \wedge$

RelIfSentences(A, D) $\Rightarrow$ RelIfSentences(Aρ, Cσ)
Proof:
By definition of reliable if-sentences we must show:

$\forall$ i : A, v, w : $\mathscr{N}$ •
($\exists$ s_1,..., s_n, t, f, S_1, S_2 • A(i).Body = S_1 \$ s_1,…, s_n := (v=w t f); S_2) $\wedge$ (A(i:v) $\notin$ D $\vee$ A(i:w) $\notin$ D)
$\Rightarrow$ (Aρ(i:v) $\notin$ C $\vee$ Aρ(i:w) $\notin$ C)

Proposition P.5.4.4 gives A(i:v) ∉ D ⇒ (A(i:v))ρ ∉ C and similar for A(i:w). Proposition P.5.3.1 gives Aρ(i:v) = (A(i:v))ρ and we then have A(i:v) ∉ D ⇒ Aρ(i:v) ∉ C and A(i:v) ∉ D ⇒ Aρ(i:w) ∉ C and the proposition holds.
❑

5.4.8 An equivalent definition of observable similarity

The proposition in this section shows that observable similarity of action sequences can be formulated in an alternative way. The proposition was formulated in order to get some insight into similar observable behaviour of reliable refinements. The conclusion of the proposition is also used in a later proof.

This definition is based on the assumption that any action from execution of a sentence *not* in the observing configuration, is from execution of a sentence in a configuration with no external inheritance. Assume that there is an action sequence $\overline{\alpha} \in$ Traces(C) and the set of observers is O. Then, by observation O.5.3.1, we have that any clone and assignment action $\overline{\alpha}_i$ such that $\overline{\alpha}_i$.exe ∉ O will have slot owners not in O. This means that for any slot s in $\overline{\alpha}_i$ where $\overline{\alpha}_i$.exe ∉ O we will have that owner($\overline{\alpha}_i$.exe, s, C) ∉ O.

In this alternative version the following hold:

Two action sequences are observably similar if for each action in ($\overline{\alpha}$)/obs&exe(O), where the i'th action is denoted α_i, there is an i'th action β_i in $\overline{\beta}$/obs&exe(O), such that the .exe-parts of the actions are observably equal (α_i.exe ~$_O$ β_i.exe) and

- if α_i is an action cloning an observing object and updates a slot in a non observing object, ie, α_i.dsc = l.s:=k/o ∧ l ∉ O, then β_i clones the same object and updates some slot not in an observing object, ie, β_i.dsc = j.t:=k/o ∧ j ∉ O

- otherwise the description parts of the actions are equal relative to the substitution σ, ie, α_i.dsc ≡ $\beta_i\sigma$.dsc.

In short, when two actions are observably similar and not hidden, then either they are clone actions cloning the same observer object and updates slots not in observers or the description parts of the actions are equal relative to a substitution.

The following proposition shows that this is equivalent to observable similarity as defined above.

Proposition P.5.4.7 Equivalent definition of observably similar action sequences from reliable configurations

Given
a set of object names O,
(*) an action sequence $\overline{\alpha}$ where each action $\overline{\alpha}_i$ such that $\overline{\alpha}_i$.exe ∉ O stem from execution of a sentence in an object not named in O and where the configuration where the sentence is found has no external inheritance.
a substitution σ where RelSubst(σ, O)
and where slot names are not changed by the substitution

We can then show that :

$$\forall\ \overline{\beta} \bullet \quad \overline{\alpha} \leq_{O,\sigma} \overline{\beta} \ \Leftrightarrow\ \overline{\alpha}/\text{obs\&exe(O)} \cong_{O,\sigma} \overline{\beta}/\text{obs\&exe(O)}$$

where $\quad \cong_{O,\sigma} \overline{\beta} \quad == \forall\ i \leq \#\overline{\alpha} \bullet \overline{\alpha}_i \cong_{O,\sigma} \overline{\beta}_i$

and $\alpha \cong_{O,\sigma} \beta \quad == \alpha.\text{exe} \sim_O \beta.\text{exe} \wedge$
if α.dsc ≡ l.s:=k/o ∧ l ∉ O
then β.dsc ≡ j.t:=k/o ∧ j ∉ O
else α.dsc ≡ βσ.dsc

Proof:

We let $<\alpha_1,\ldots, \alpha_n>$ = $\overline{\alpha}$/obs&exe(O) and $<\beta_1,\ldots, \beta_m>$ = $\overline{\beta}$/obs&exe(O). By observation O.5.3.1 (*) gives that any clone and assignment action $\overline{\alpha}_i$ such that $\overline{\alpha}_i$.exe $\notin$ O will have slot owners not in O. This means that if we have $\overline{\alpha} \in$ Traces(C) then for any slot s in $\overline{\alpha}_i$ where $\overline{\alpha}_i$.exe $\notin$ O we will have owner($\overline{\alpha}_i$.exe, s, C) $\notin$ O.

We divide the proof in two cases: $\alpha_i \in$ obs(O) and $\alpha_i \notin$ obs(O).

Case 1) $\alpha_i \in$ obs(O)
Then we show

$\alpha_i \sim_{O,\sigma} \beta_i$
$\Leftrightarrow$
β_i.exe $\sim_D$ α_i.exe $\wedge$ *if* α_i.dsc $\equiv$ l.s:=k/o $\wedge$ l $\notin$ O *then* β_i.dsc $\equiv$ j.t:=k/o $\wedge$ j $\notin$ O *else* α_i.dsc $\equiv \beta_i\sigma$.dsc

We prove this by two subcases 1a) α_i.dsc $\equiv$ l.s:=k/o $\wedge$ l $\notin$ O and 1b) all other subcases of case 1)

Case 1a)
Proof of $\Rightarrow$

When α_i.dsc $\equiv$ l.s:=k/o $\wedge$ l $\notin$ O then by (*) we have that when l $\notin$ O, then α_i.exe $\notin$ O. Then, since $\alpha_i \in$ $\overline{\alpha}$/obs&exe(O) we must have that o, k $\in$ O. Then $\alpha_i \sim_{O,\sigma} \beta_i$ gives β_i.dsc $\equiv$ j.t:=k/o and the proposition holds for this case.

Proof of $\Leftarrow$

When α_i.dsc $\equiv$ l.s:=k/o $\wedge$ l $\notin$ O then β_i.dsc $\equiv$ j.t:=k/o $\wedge$ j $\notin$ O. By (*) we have that when l $\notin$ O and j $\notin$ O, then α_i.exe $\notin$ O and β_i.exe $\notin$ O. Then by definition of observable similarity we have $\alpha_i \sim_{O,\sigma} \beta_i$ and the proposition holds for this case.

Case 1b)
Then we have $\alpha_i \sim_{O,\sigma} \beta_i \Leftrightarrow \beta_i$.exe $\sim_D$ α_i.exe $\wedge$ α_i.dsc $\equiv \beta_i\sigma$.dsc.
Proof of $\Rightarrow$

From $\alpha_i \sim_{O,\sigma} \beta_i$ we have β_i.exe $\sim_D$ α_i.exe. We then have the following cases for the description part of the actions when $\alpha_i \sim_{O,\sigma} \beta_i$ and $\alpha_i \in$ obs(O) and when α_i is not a clone action updating a slot in another configuration:

α.dsc $\equiv$ o!x($\overline{p}\sigma$)/k $\wedge$ β.dsc $\equiv$ o!x($\overline{p}$)/k where o $\in$ O
$\vee$
α.dsc $\equiv$ o.s:=kσ/iσ $\wedge$ β.dsc $\equiv$ o.s:=k/i where o $\in$ O
$\vee$
α.dsc $\equiv$ o.s:= jσ $\wedge$ β.dsc $\equiv$ o.s:=j where o $\in$ O

which gives α.dsc $\equiv \beta\sigma$.dsc and the proposition holds for this case.

Proof of $\Leftarrow$

We have the following cases for the description part of the actions when α_i.dsc $\equiv \beta_i\sigma$.dsc and $\alpha_i \in$ obs(O) and when α_i is not a clone action updating a slot in another configuration:

α_i.dsc $\equiv$ o!x($\overline{p}\sigma$)/k $\wedge$ β_i.dsc $\equiv$ o!x($\overline{p}$)/k where o $\in$ O
$\vee$
α_i.dsc $\equiv$ o.s:=kσ/iσ $\wedge$ β_i.dsc $\equiv$ o.s:=k/i where o $\in$ O
$\vee$
α_i.dsc $\equiv$ o.s:= jσ $\wedge$ β_i.dsc $\equiv$ o.s:=j where o $\in$ O

When we have β_i.exe $\sim_D$ α_i.exe, then this gives $\alpha_i \sim_{O,\sigma} \beta_i$ and the proposition holds for this case.

Case 2) $\alpha_i \notin$ obs(O)
Then we show:

$\alpha_i \equiv \beta_i\sigma$
$\Leftrightarrow$
β_i.exe $\sim_D$ α_i.exe $\wedge$ *if* α_i.dsc $\equiv$ l.s:=k/o $\wedge$ l $\notin$ O *then* β_i.dsc $\equiv$ j.t:=k/o $\wedge$ j $\notin$ O *else* α_i.dsc $\equiv \beta_i\sigma$.dsc

When α_i = $\overline{\alpha}$/obs&exe(O) then the only case when $\alpha_i \notin$ obs(O) is:

$\alpha \equiv$ e->o!x($\overline{p}$)/k where e $\in$ O

This is different from α_i.dsc $\equiv$ l.s:=k/o and we must therefore show:

$\alpha_i \equiv \beta_i\sigma \Leftrightarrow \beta_i$.exe $\equiv \alpha_i$.exe $\wedge$ α_i.dsc $\equiv \beta_i\sigma$.dsc

This holds trivially and the proposition holds for this case.
❑

5.5 A Reliable Refinement Relation

A refinement relation between configurations was defined in chapter 4. The definition was done based on a definition of the observably equal action sequence relation. Making similar refinement definitions based on the observably similar action sequence relation is not straight forward. The observable similarity relation definitions include a substitution and the complexity is related to getting the specification of the substitution right.

The most simple solution is to specify one reliable substitution and require that all the observable actions in the traces are observably similar relative to a reliable substitution. We would then get a definition of refinement configurations as follows (briefly sketched):

$A \leq_{D,\sigma} B == \quad \forall\ \overline{\alpha} : \text{Traces}(A||D\sigma)\ \exists\ \overline{\beta} : \text{Traces}(B||D) \bullet$

$\overline{\alpha} \leq_{D',\sigma} \overline{\beta} \wedge (\text{endColab}(A, D, \overline{\alpha}) \Rightarrow \text{endColab}(B, D, \overline{\beta}))$

where we have RelSubst(σ, A, B, D)

This definition is unnecessarily strict. This is because $A \leq_{D,\sigma} B$ holds also when pairs of sequences of actions are similar relative to different substitutions in $\overline{\alpha} \leq_{D',\sigma} \overline{\beta}$, ie, different σ. These different σ-substitutions must all be reliable and they can be defined based on the pair of action sequences they are associated with. When this is done right, the result will be a definition of a reliable refinement relation sufficiently strict, but not too strict, as far as the substitution is concerned. In order to define such a relation we first define "the prime of a substitution". This is done in the next subsection.

5.5.1 The prime of a substitution

The prime of a substitution is used in a definition of a reliable refinement relation sufficiently strict, but not too strict, as far as the substitution is concerned. The prime of a substitution is used in relation to defining a sufficiently restricting, but not too restricting, substitution to be used in the definition of configuration refinements. The prime of a substitution is created when there are two action sequences, eg, $\overline{\alpha}$ from A||Dσ and $\overline{\beta}$ from B||D, which are observably similar, ie, $\overline{\alpha} \leq_{D',\sigma} \overline{\beta}$. Assume that we have a function prime(σ, $\overline{\alpha}$, $\overline{\beta}$, A, B, D) which define the necessary and sufficient requirements on substitutions. For $A \leq_{D,\sigma} B$ to hold we then require that action sequences are observably similar relative to a prime substitution as follows:

$\overline{\alpha} \leq_{D',\sigma'} \overline{\beta}$ where σ' = prime(σ, $\overline{\alpha}$, $\overline{\beta}$, A, B, D)

This will define $A \leq_{D,\sigma} B$ in such a way that it holds also when pairs of sequences of actions are similar relative to different substitutions in $\overline{\alpha} \leq_{D',\sigma} \overline{\beta}$, ie, different σ'.

The prime of the substitution σ is created from the old substitution σ and the object names found in the action sequences $\overline{\alpha}$ and $\overline{\beta}$. The prime of a substitution is formally defined as follows (note that when the two actions are observably similar they are, by definition, of the same kind):

Definition: The prime of a substitution: prime(σ, α, β, A, B, D)

The function is defined using the multiple case-notation presented in appendix A, which allows casing on more than one item:

prime(σ, α, β, A, B, D) ==
case α, β *of*
e->o.s := k/i, f->o.s := l/j : *if* $o \in D \wedge i,j \notin D$ *then* σ + { k / l } *else* σ
e->o!m($p_1 \ldots p_n$)/k, f->o!m($q_1 \ldots q_n$)/k : *if* $o \in D$ *then* σ_n *else* σ
otherwise σ

where σ_n is defined as follows:
$\sigma_0 = \sigma$
and for $i \in \{1..n\}$
σ_i = *if* $q_i \notin \text{keys}(\sigma_{i-1}) \wedge q_i \in B \wedge p_i \in A$ *then* $\sigma_{i-1} + \{p_i / q_i\}$ *else* σ_{i-1}

σ_i is defined by adding zero or more key/value pairs to σ_{i-1}. Key/value pairs $\{p_i / q_i\}$ are only added to σ_{i-1} when the key q_i is not a key in σ_{i-1}, ie, $q_i \notin keys(\sigma_{i-1})$ and the key q_i is an object name in B, ie, $q_i \in B$. In addition, a new pair is only added when the value p_i is a name of an object in A, ie, $p_i \in A$.

The new substitution σ' = prime(σ, α, β, A, B, D) is created by adding substitutions for all B-object names which for the first time will appear as slot values in D after the action β. Such B-names stem from message send to objects in D and cloning of objects in B where the owner of the updated slot is in D.

Examples:
If the two actions are message-send actions, eg, α is e->o!m($p_1 \ldots p_n$)/k and β is f->(o!m($q_1 \ldots q_n$)/k, then new key/value pairs $\{p_i / q_i\}$ are added if q_i is not found as a key in σ and q_i is an object name in B and p_i is an object name in A. Note that when the two actions are observably similar, it must hold that

$q_j \neq q_i$ or ($q_j = q_i$ and $p_j = p_i$) where j < i.

If this does not hold, there will be $\{p_i / q_i\}$ substitutions which have the same B-name but different A-names. This will not give $\alpha \leq_{D,\sigma} \beta$ and it will neither give reliability.

If the two actions are clone actions, eg, α is e->o.s := k/i and β is f->o.s := l/j, then a new key/value pair {k/l} is added to σ if i and j are names of objects in A and B, respectively.

The prime function is extended to sequences of actions as follows:

prime(σ, α, <$\beta_1 \ldots \beta_n$>, A, B, D) == *if* n = 0 *then* σ *else*
prime(prime(σ, α, <$\beta_1 \ldots \beta_{n-1}$>, A, B, D), α, β_n, A, B, D)

prime(σ, <$\alpha_1 \ldots \alpha_n$>, $\overline{\beta}$, A, B, D) == *if* n = 0 *then* σ *else*
prime(prime(σ, <$\alpha_1 \ldots \alpha_{n-1}$>, $\overline{\beta}$, A, B, D), α_n, $\overline{\beta}$, A, B, D)

Next we observe that σ and prime(σ, α, $\overline{\beta}$, A, B, D) are equal when all the actions are not observed.

Observation O.5.5.1 : Non-observed actions give equal substitutions and primed substitutions

When we have an action α, then we have that

$\alpha \notin obs(D) \Rightarrow \sigma$ = prime(σ, α, $\overline{\beta}$, A, B, D)

for any substitution σ, any action sequence $\overline{\beta}$ = <β_1,...,β_n> and any configurations A, B, D.

This holds because if the substitution and the substitution's prime were different, then we must have had one of the two cases:

α = e->i.s := k/o $\wedge$ o $\in$ D or
α = e->o!m($\overline{p}$)/k $\wedge$ o $\in$ D.

When o $\in$ D then by definition of observable actions, we have $\alpha \in obs(D)$, and then the initial assumption $\alpha \notin obs(D)$ does not hold.

5.5.2 Observably similar actions and prime substitutions

Primed substitutions is used in conjunction with defining a reliable refinement relation. Typically, when

$\overline{\alpha} \in$ Traces(A||Dσ) and $\overline{\beta} \in$ Traces(B||D)

then we require

$\overline{\alpha} \leq_{D',\sigma'} \overline{\beta}$ where σ' = prime(σ, $\overline{\alpha}$, $\overline{\beta}$, A, B, D)

where $\leq_{D',\sigma'}$ denote observable similarity defined above.

The following propositions show that by the way primed substitutions, observably similar actions and primed configurations are defined, observably similar actions ensure reliable primed substitutions if the original substitution was reliable. This property is important since the primed substitution must be reliable if it is to be used in the definition of a reliable refinement relation.

Proposition P.5.5.1 Observably similar actions ensure reliable primed substitution for derived configurations

$\forall$ A, B, D, σ, α, β •
$\sigma \in B \rightarrow A \wedge A||D, B||D \in \mathscr{C}_{Safe} \wedge$
$\alpha \in Traces(A||D\sigma) \wedge \beta \in Traces(B||D) \wedge \alpha \leq_{D',\sigma'} \beta$
$\Rightarrow \sigma' \in B' \rightarrow A' \wedge A'||D', B'||D'' \in \mathscr{C}_{Safe}$

where σ' = prime(σ, α, β, A, B, D)
and A' = prime(A, Dσ, α), D' = prime(Dσ, A, α), B' = prime(B, D, β) and D'' = prime(D, B, β)

Proof:
For any new objects created by cloning B-objects or D-objects in B||D, the rules of actions give that the names of the new objects are not found as names in B or D. This means neither as object names nor slot names. We then have B'.Dom $\cap$ D''. Dom = Ø, and all new object names are different from slot names in the configurations, ie, ON(B'||D'') $\cap$ SN(B'||D'') = Ø. This gives B'||D'' $\in \mathscr{C}_{Safe}$. For corresponding reasons we have A'||D' $\in \mathscr{C}_{Safe}$.

By definition of the prime function, σ differs from σ' when either:
$\alpha \in$ obs(D) and the action is a message-send action from a sentence in A or
α is a clone action cloning an object in A and the action is from a sentence in D
The proof of the lemma is done by these two cases:

Case 1) $\alpha \in$ obs(D) and the action is a message-send action from a sentence in A
Because the requirement $q_i \notin keys(\sigma) \wedge q_i \notin B$ in the definition of prime(σ, α, β, A, B, D) only B-names will be added as keys to σ. This ensures keys(σ') $\subseteq$ B when keys(σ) $\subseteq$ B. The requirement that $p_i \in A$ in the definition of prime(σ, α, β, A, B, D) ensures values(σ') $\subseteq$ A when values(σ) $\subseteq$ A and we then have $\sigma' \in B' \rightarrow A'$.

Case 2) α is a clone action cloning an object in A and the action is from a sentence in D
In the definition of the prime function the name denoted k will be an element in keys(σ'). This name is the name of a new object in B' as compared to B, since it is required that k $\notin$ D. Then we have keys(σ') $\subseteq$ B when keys(σ) $\subseteq$ B. Similarly, the name denoted l will be an element in values(σ') and this is the name of an object in A'. Then we have values(σ') $\subseteq$ A' when values(σ) $\subseteq$ A and we then have $\sigma' \in B' \rightarrow A'$.
❑

Next we show an important property of observably similar action sequences, namely that pair wise concatenation of two observably similar action sequences gives observably similar sequences. This means that if we have the action sequences $\overline{\alpha}_1$, $\overline{\beta}_1$, $\overline{\alpha}_2$ and $\overline{\beta}_2$ and a set of object names O and a substitution σ, and where the sequences are pairwise observably similar as follows:

$$\overline{\alpha}_1 \leq_{O,\sigma'} \overline{\beta}_1 \wedge \overline{\alpha}_2 \leq_{O \cup Q,\sigma''} \overline{\beta}_2$$

where σ' includes the new names in the first pair of action sequences and σ'' is a prime of σ' and σ'' includes the new names in the second pair of action sequences. In this case we will then have that if we concatenate $\overline{\alpha}_1$ and $\overline{\alpha}_2$ then this sequences will be observably equal to the action sequences which is the concatenation of $\overline{\beta}_1$ and $\overline{\beta}_2$. We also have that when the concatenated sequences are observably similar, then there exists pairs of sub-sequences which are observably similar.

Proposition P.5.5.2 Pair wise concatenation of two observably similar action sequences gives observably similar sequences

(*) $\forall$ A, B, D, $\overline{\alpha}_1$, $\overline{\alpha}_2$, $\overline{\beta}_1$, $\overline{\beta}_2$, O, $\sigma \bullet \sigma \in B \to A \wedge$

$$\overline{\alpha}_1 \leq_{O,\sigma'} \overline{\beta}_1 \wedge \overline{\alpha}_2 \leq_{O \cup Q,\sigma''} \overline{\beta}_2 \Rightarrow (\overline{\alpha}_1 \,\&\, \overline{\alpha}_2) \leq_{O \cup Q,\sigma''} (\overline{\beta}_1 \,\&\, \overline{\beta}_2)$$

$\wedge$

(**) $\forall$ $\overline{\alpha}$, $\overline{\beta}$, O, $\sigma \bullet$

$$\overline{\alpha} \leq_{O \cup Q,\sigma''} \overline{\beta} \quad \Rightarrow \quad \exists\, \overline{\alpha}_1, \overline{\alpha}_2, \overline{\beta}_1, \overline{\beta}_2 \bullet \; \overline{\alpha}_1 \leq_{O,\sigma'} \overline{\beta}_1 \wedge \overline{\alpha}_2 \leq_{O \cup Q,\sigma''} \overline{\beta}_2 \wedge \overline{\alpha} \equiv \overline{\alpha}_1 \,\&\, \overline{\alpha}_2 \wedge \overline{\beta} \equiv \overline{\beta}_1 \,\&\, \overline{\beta}_2$$

where Q = NewNames($\overline{\alpha}_1$) and

σ' = prime(σ, $\overline{\alpha}_1$, $\overline{\beta}_1$, A, B, D) $\wedge$

σ'' = prime(σ', $\overline{\alpha}_2$, $\overline{\beta}_2$, A', B', D') where

A' = prime(A, Dσ, $\overline{\alpha}_1$), D' = prime(Dσ, A, $\overline{\alpha}_1$), B' = prime(B, D, $\overline{\beta}_1$)

Proof:
Proof of (*):
When we have Q = NewNames($\overline{\alpha}_1$) then Q only holds names not found in $\overline{\alpha}_1$ and $\overline{\beta}_1$.
When σ' = prime(σ, $\overline{\alpha}_1$, $\overline{\beta}_1$, A, B, D), then the difference between the two substitutions is that σ' will substitute some names which are not found in $\overline{\beta}_1$. Then we have $\overline{\alpha}_1 \leq_{O \cup Q,\sigma''} \overline{\beta}_1$.

By definition of obs&exe(O) we have

($\overline{\alpha}_1$)/obs&exe(O$\cup$Q) & ($\overline{\alpha}_2$)/obs&exe(O$\cup$Q) = ($\overline{\alpha}_1$ & $\overline{\alpha}_2$)/obs&exe(O$\cup$Q)

and similar for the β-actions. Then, by definition of the observable similarity for action sequences we have

$$\overline{\alpha}_1 \leq_{O,\sigma''} \overline{\beta}_1 \wedge \overline{\alpha}_2 \leq_{O \cup Q,\sigma''} \overline{\beta}_2 \Rightarrow \overline{\alpha}_1 \,\&\, \overline{\alpha}_2 \leq_{O \cup Q,\sigma''} \overline{\beta}_1 \,\&\, \overline{\beta}_2$$

and the proposition holds for this case.

Proof of (**):
When we have $\overline{\alpha} \leq_{O \cup Q,\sigma'} \overline{\beta}$ then by definition of observable similarity for action sequences this gives that there exists some sequences $\overline{\alpha}_1$, $\overline{\alpha}_2$, $\overline{\beta}_1$, $\overline{\beta}_2$, such that $\overline{\alpha}_1 \leq_{O,\sigma''} \overline{\beta}_1 \wedge \overline{\alpha}_2 \leq_{O \cup Q,\sigma''} \overline{\beta}_2$. Since we have Q = NewNames($\overline{\alpha}_1$) then Q only holds names not found in $\overline{\alpha}_1$ and $\overline{\beta}_1$.
When σ' = prime(σ, $\overline{\alpha}_1$, $\overline{\beta}_1$, A, B, D), then the difference between the two substitutions is that σ' will substitute some names which are not found in $\overline{\beta}_1$. Then we have $\overline{\alpha}_1 \leq_{O,\sigma'} \overline{\beta}_1$ and the proposition holds for this case.
❑

5.5.3 A reliable refinement relation

Based on the prime of substitutions, observable similarity of action sequences and the reliability requirements of previous sections, we can define a reliable refinement relation between configurations as follows:

Definition: Refinement relation with specialisation; $A \leq_{D,\sigma} B$

Given configurations A, B, D $\in \mathscr{C}$ and a substitution σ. We define a binary relation called *a refinement relation with specialisation*, denoted $A \leq_{D,\sigma} B$, as follows:

$$A \leq_{D,\sigma} B == \quad \forall\ \bar{\alpha} : Traces(A||D\sigma)\ \exists\ \bar{\beta} : Traces(B||D) \bullet$$

$$A||D, B||D \in \mathscr{C}_{Safe} \wedge \sigma \in B \to A \wedge Reliable(A, D\sigma, \bar{\alpha}) \wedge$$

$$\bar{\alpha} \leq_{D',\sigma'} \bar{\beta} \wedge (endColab(A, D\sigma, \bar{\alpha}) \Rightarrow endColab(B, D, \bar{\beta}))$$

$$\text{where } D' = prime(D, A, \bar{\alpha}) \text{ and } \sigma' = prime(\sigma, \bar{\alpha}, \bar{\beta}, A, B, D)$$

The following proposition shows that the refinement relation with specialisation is transitive. In the proposition it is assumed that D is the observing configuration. D is defined for collaboration with E. D is specialised for collaboration with B by substitution ρ and Dρ is specialised for collaboration with A by substitution σ.

Proposition P.5.5.3 : The refinement relation with specialisation is transitive

The refinement relation with specialisation is transitive, ie, we have:

$$\forall\ A, B, D, E, \sigma, \rho \bullet \quad A||E \in \mathscr{C}_{Safe} \wedge A \leq_{D\rho,\sigma} B \wedge B \leq_{D,\rho} E \Rightarrow A \leq_{D,\rho\sigma} E$$

Proof:
$A \leq_{D\rho,\sigma} B$ gives that we for each $\bar{\alpha} \in Traces(A||D\rho\sigma)$ have:

$$\exists\ \bar{\beta} : Traces(B||D\rho) \bullet$$

$$A||D\rho\sigma, B||D\rho \in \mathscr{C}_{Safe} \wedge \sigma \in B \to A \wedge Reliable(A, D\rho\sigma, \bar{\alpha}) \wedge$$

$$\bar{\alpha} \leq_{(D\rho)',\sigma'} \bar{\beta} \wedge (endColab(A, D\rho\sigma, \bar{\alpha}) \Rightarrow endColab(B, D\rho, \bar{\beta}))$$

$$\text{where } \sigma' = prime(\sigma, \bar{\alpha}, \bar{\beta}, A, B, D\rho)$$

and for this $\bar{\beta}$ we have some $\bar{\gamma} \in Traces(E||D)$ where

$$B||D, E||D \in \mathscr{C}_{Safe} \wedge \rho \in E \to B \wedge Reliable(B, D\rho, \bar{\beta}) \wedge$$

$$\bar{\beta} \leq_{D',\rho'} \bar{\gamma} \wedge (endColab(B, D\rho, \bar{\beta}) \Rightarrow endColab(E, D, \bar{\gamma}))$$

$$\text{where } \rho' = prime(\rho, \bar{\beta}, \bar{\gamma}, B, E, D)$$

To prove the proposition we must then show:

$$\forall\ \bar{\alpha} : Traces(A||D\sigma)\ \exists\ \bar{\gamma} : Traces(E||D) \bullet$$

$$A||D\rho\sigma, E||D \in \mathscr{C}_{Safe} \wedge \rho\sigma \in E \to A \wedge Reliable(A, D\rho\sigma, \bar{\alpha}) \wedge$$

$$\bar{\alpha} \leq_{(D\rho)',\rho\sigma'} \bar{\gamma} \wedge (endColab(A, D\rho\sigma, \bar{\alpha}) \Rightarrow endColab(E, D, \bar{\gamma}))$$

$$\text{where } \rho\sigma' = prime(\rho\sigma, \bar{\alpha}, \bar{\beta}, A, B, D)$$

Trivially we have $(endColab(A, D\rho\sigma, \bar{\alpha}) \Rightarrow endColab(E, D, \bar{\gamma}))$. From $A \leq_{D\rho,\sigma} B$ we have $A||D\rho\sigma \in \mathscr{C}_{Safe}$, $\sigma \in B \to A$ and $Reliable(A, D\rho\sigma, \bar{\alpha})$. From $B \leq_{D,\rho} E$ we have $E||D \in \mathscr{C}_{Safe}$ and $\rho \in E \to B$.

From safe names-requirements we have that all configurations have non-overlapping names. This gives $\rho\sigma \in E \to A$ and also by observation O.5.2.1 (observing objects' names are never keys in the substitution) the substitutions do not change D-object names. We can therefore use proposition P.5.4.2 and conclude that $\bar{\alpha} \leq_{(D\rho)',\rho\sigma'} \bar{\gamma}$. This shows that the refinement relation with specialisation is transitive.
❑

By proposition P.5.5.3 the refinement relation with specialisation is a monotonous partial order. Note that in the special case where the substitution σ does not substitute any object names in A, ie, A.Dom $\cap$ σ.keys, then we have:

$$A \leq_{D,\sigma} A$$

and the refinement relation with specialisation is reflexive and then a complete partial order for this special case. Such a situation might be viewed as a normal case since the same context should collaborate with A both when A is viewed as a specification and as a refinement. If there were keys in σ which substituted an A-name, then this name would be substituted with another A-name. In such a case, the objects in A would change roles, and then the A with changed object roles might or might not be a refinement of the A without such role changes. This seems intuitively correct and should therefore be reflected in the fact that the refinement relation should not be reflexive in the general case with some specialisation of the context.

5.5.4 Equal actions from different context sentences

By definition of observable equality and similarity, two actions stemming from different sentences in the context are seen as observably *unequal*. To see the consequence of this decision incorporating the reliability requirements, consider a situation where there are two actions, α_1 from A||D and β_1 from B||D both legal at a given point in an execution. The actions are observably equal or similar as defined above, except that they stem from the execution of different sentences in D, ie, we have α_1.exe $\neq$ β_1.exe. Then there are at least two sentences which may be executed in D, ie, have a $-sign to their left.

Assume that α_1 is from execution of a sentence in D in A||D and β_1 is from execution of a sentence in D in B||D. Also assume that the two executed sentences are found in objects e and f where e = α_1.exe and f = β_1.exe. By the way Omicron is defined, and also in most practical implementations of object systems, the two sentences can then be executed in any order. The alternative actions from A||D and B||D when there are executable sentences in the objects named e and f are then:

$<\alpha_1, \alpha_2> \in$ Traces(A||D) where e = α_1.exe and f = α_2.exe
$<\beta_1, \beta_2> \in$ Traces(B||D) where f = β_1.exe and e = β_2.exe
$<\gamma_1, \gamma_2> \in$ Traces(A||D) where f = γ_1.exe and e = γ_2.exe
$<\delta_1, \delta_2> \in$ Traces(B||D) where e = δ_1.exe and f = δ_2.exe

Let O denote the object names in D. When A is a refinement of B relative to D and we ignore differences in the .exe-parts of the actions we have:

$\alpha_1 =_O \beta_1 \wedge \alpha_2 =_O \beta_2$ and
$\gamma_1 =_O \delta_1 \wedge \gamma_2 =_O \delta_2$

If $\alpha_1 = \gamma_1$ except for the .exe-parts, then we also have $\gamma_1 = \beta_1$ where the .exe parts are equal. We then have $\gamma_1 =_O \beta_1$. Since β_1 is from the same sentence in f as γ_1, the only way there may be differences in β_1 and γ_1 are if objects in D inherit slots from objects in A and B. When we have the reliability requirement "no external inheritance", it is still reliable to let D inherit from A and B. However, such inheritance is "dirty" or it is difficult to find a good reason for it, since a reliable refinement of D will not do the same. A "clean and tidy" specification would therefore not be expected to have such inheritance. Therefore, if D was a clean and tidy specification we would always have $\gamma_1 = \beta_1$. The consequence of this is that if we have the above situations, then we would also have:

$\gamma_1 =_O \beta_1 \wedge \gamma_2 =_O \beta_2$ and
$\alpha_1 =_O \delta_1 \wedge \alpha_2 =_O \delta_2$

where the .exe-parts of the actions are pairwise equal as defined for observably equal actions. Therefore, requiring the .exe-parts to be equal for actions from the observing configuration has no influence on similarity relations for "nice and tidy" specifications with no external inheritance since this requirement will always be met. And since observable similarity implies observable equality, this also holds for observably similar action sequences.

5.5.5 Limitations on visible objects in refinements

When it is required that:

$$\alpha \leq_{D,\sigma'} \beta \qquad \text{where } \sigma' = \text{prime}(\sigma, \alpha, \beta, A, B, D)$$

for $A \leq_{D,\sigma} B$ to hold, then this allows variations in the use of A-names and B-names in the actions.
However, the use of A-names in message-send actions to D-objects from sentences in A are limited. The limitation applies to D-observable message-send actions from sentences in A. The limits are set by the corresponding observably similar message-send actions from sentences in B. Denote the observably similar D-observable actions from sentences in A and B by α and β respectively. The variation in A-names in α is then limited as follows for each observably similar β:

- the number of different A-names in α is limited by the number of different B-names in β and
- the placement of the A-names in α is restricted to the positions where B-names are found in β and
- new A-names can only be introduced to C and D when new B-names are introduced
- the same A-name is always found in corresponding positions for a given B-name, eg, whenever the B-name p is found in a β-action then the A-name q will be found in the corresponding position in the corresponding α-action. Also, in any corresponding α' and β' actions occurring after α and β respectively, q will always be found in α' in the position where p is found in β'.

This sets a limit on the number of visible object names of the refinement A to the number of visible object names of the specification B. Formally stated this says:

$$\#\text{Visible}(A, D\sigma) \leq \#\text{Visible}(B, D)$$

The consequence of this is that *specification configurations such as B and D specify a maximum number of visible object names of refinements of B and D*. Note that this does not restrict the total number of objects in a refinement of a specification, it only limits the number of *visible* objects. How limits on the number of visible objects relates to design practices is discussed in chapter 8.

5.5.6 Relationships between the refinement relations

In chapter 4 a refinement relation between configurations was defined based on component developers intuitive notion of similarity as described in chapter 2. As the examples in this chapter have shown, the component developers notion of similarity does not ensure reliable substitution. Therefore the definition of the refinement relation found in chapter 4 was modified to give a reliable refinement relation as defined in chapter 1. The reliable refinement relation was called "refinement with specialisation". Below we show that when a configuration is a refinement with specialisation of some other configuration, then it is also a refinement of the configuration as defined in chapter 4.

We first show that when two actions are observably similar relative to a reliable substitution, then they are also observably equal.

Proposition P.5.5.4 Observably similar actions are also observably equal

Given a substitution σ and a set of object names O such that RelSubst(σ, O). We then have for any two actions α and β:

$$\alpha \sim_{O,\sigma} \beta \Rightarrow \alpha \sim_{O} \beta$$

Proof:
When we have $\alpha \sim_{O,\sigma} \beta$, then by observation O.5.4.2 we have that all O-names in the actions are equal and the slot names are equal. By definition of observable equality, this gives $\alpha \sim_{O} \beta$.
❑

Next we show $A \leq_{D,\sigma} B \Rightarrow A \leq_{D} B$ where we let σ be empty. This is because if the substitution is non-empty, then the actions from A||Dσ and A||D may be substantially different and it is therefore not possible to show the proposition. To make it possible to prove the proposition with a non-empty substitution, the refinement relation of chapter 4 has to be redefined to allow different contexts to be combined with the refinement and the specification so that the behaviour of A||Dσ is compared with the behaviour of B||D. This is left for further study, as it does not seem to give any important insights into the problem area of reliable substitution.

Proposition P.5.5.5 Refinements with specialisation are also refinements

Then refinement with specialisation implies refinement, ie,

$\forall$ A, B, D • $A \leq_{D,\{\}} B \Rightarrow A \leq_D B$

where {} denote an empty substitution.

Proof:

$A \leq_{D,\{\}} B$ gives that we for each $\overline{\alpha} \in$ Traces(A||D) have:

$\exists\ \overline{\beta}$: Traces(B||D) •

A||Dσ, B||D $\in \mathscr{C}_{Safe} \wedge \{\} \in B \rightarrow A \wedge$ Reliable(A, D, $\overline{\alpha}$) $\wedge$

$\overline{\alpha} \leq_{D',\sigma'} \overline{\beta} \wedge$ (endColab(A, D, $\overline{\alpha}$) $\Rightarrow$ endColab(B, D, $\overline{\beta}$))

where σ' = prime({}, $\overline{\alpha}$, $\overline{\beta}$, A, B, D)

To prove the proposition we must then show:

$\forall\ \overline{\alpha}$: Traces(A||D) $\exists\ \overline{\beta}$: Traces(B||D) •

$\overline{\alpha} =_{D'} \overline{\beta} \wedge$ (endColab(A, D, $\overline{\alpha}$) $\Rightarrow$ endColab(B, D, $\overline{\beta}$))

Since this by definition of observable equality gives $A \leq_D B$.

When we have $\overline{\alpha} \leq_{D',\sigma'} \overline{\beta}$, then by definition of observably similar action sequences we have:

When $<\alpha_1, \alpha_2,...> = \overline{\alpha}$/obs&exe(D') $\approx_{D',\sigma'} \overline{\beta}$/obs&exe(D')

and by definition of $\approx_{D',\sigma'}$ this gives for $<\alpha_1, \alpha_2,...> = \overline{\alpha}$/obs&exe(D') and $<\beta_1, \beta_2,...> = \overline{\beta}$/obs&exe(D') that:

$\forall\ i \leq \#\overline{\alpha}$ • *if* $\alpha_i \in$ obs(D') *then* $\alpha_i \sim_{D',\sigma'} \beta_i$ *else* $\alpha_i = \beta_i\sigma$

Since this gives $\alpha_i \sim_{D',\sigma'} \beta_i$ for all actions which are in obs(D'), then we have by proposition P.5.5.4:

When $<\alpha_1, \alpha_2,...> = \overline{\alpha}$/obs(D') and $<\beta_1, \beta_2,...> = \overline{\beta}$/obs(D') then $\forall\ i \leq \#\overline{\alpha}$ • $\alpha_i \sim_{D'} \beta_i$

Then, by definition of observably equal action sequences this gives $\overline{\alpha} =_{D'} \overline{\beta}$.

Since we also have (endColab(A, D, $\overline{\alpha}$) $\Rightarrow$ endColab(B, D, $\overline{\beta}$)) from $A \leq_{D,\{\}} B$, then we have $A \leq_D B$ and the proposition holds.

❑

CHAPTER 6

Proving Reliable Substitution

This chapter shows properties of combinations of configurations which meet the reliability requirements presented in chapter 5. The final theorem, T.6.4, shows that the refinement relation with specialisation as defined in chapter 5 is reliable as defined in chapter 1. This shows that the reliability requirements and the definition of refinement with specialisation give reliable substitution of components.

Section 2.1 shows important properties of reliable configurations and specialisations of such configurations. Proposition P.6.4 shows relations between actions from corresponding sentences in a reliable configuration and a specialised version of the same configuration. Proposition P.6.6 shows that a specialised versions of a reliable configuration is also reliable while proposition P.6.7 shows properties of configurations derived from reliable configurations and specialisations of such configurations. These are central properties in showing the substitution proposition.

Section 6.2 shows the simple substitution theorem T.6.1. This theorem states the substitution proposition for systems having two parts: one component and one context.

Section 6.3 shows another substitution property related to combining new and old versions of components in a system. It is assumed that we have a set of existing components. For some of these components there are reliable refinements. It is then shown that independently of how many components are replaced by their reliable refinements, the existing components will observe no difference in behaviours.

In section 6.4 the general substitution proposition for any number of components is shown to hold for the reliable refinement relation defined in chapter 5. This is theorem T.6.3.

Section 6.5 shows that the reliability requirements and the refinement relation with specialisation give reliable substitution. This means that if some arbitrary number of components are substituted with their reliable refinement, then the other components will not observe any difference. The other components may be either "old" components or reliable refinements of old components. This is shown in theorem T.6.4.

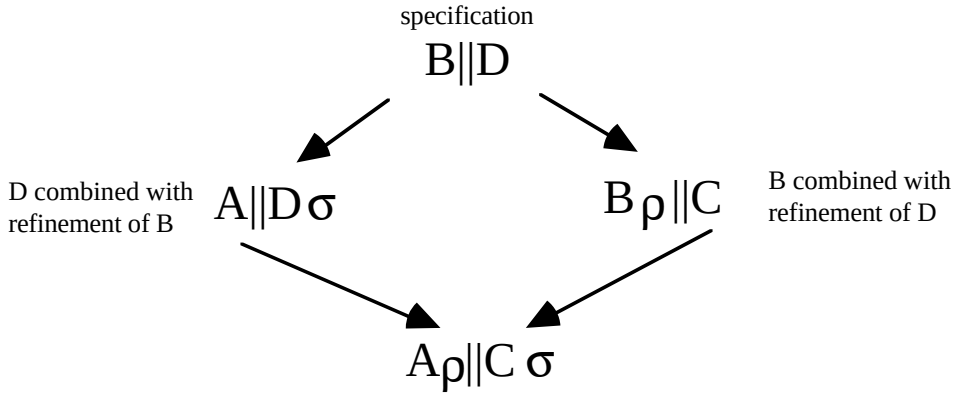
The last section, section 6.6, discusses various alternative reliability requirements in relation to a library of objects, ie, objects which are assumed to always be present in the system and therefore never be substituted.

6.1 Reliability Properties

The substitution proposition for a system consisting of two parts is formulated as follows when using the refinement relation with specialisation defined in chapter 5:

RelNames(A, B, C) $\wedge$ A $\leq_{D,\,\sigma}$ B $\wedge$ C $\leq_{B,\,\rho}$ D $\Rightarrow$ Aρ $\leq_{C,\,\sigma}$ Bρ $\wedge$ Cσ $\leq_{A,\,\rho}$ Dσ

The following figure shows the configurations and substitutions found in the simple substitution proposition and illustrates the relations between these configurations and substitutions.



The two refinements combined

Because of the symmetric form of the conclusion of the proposition, it is necessary and sufficient to show

Aρ $\leq_{C,\,\sigma}$ Bρ.

in order to prove the proposition.

By definition of the refinement relation with specialisation defined in chapter 5, we must then, among other things, show:

Aρ||Cσ $\in$ $\mathscr{C}_{Safe}$ $\wedge$ σ $\in$ Bρ $\rightarrow$ Aρ $\wedge$ Reliable(Aρ, Cσ).

Proposition P.6.1 shows Aρ||Cσ $\in$ $\mathscr{C}_{Safe}$ $\wedge$ σ $\in$ Bρ $\rightarrow$ Aρ while P.6.6 shows that Reliable(Aρ, Cσ) holds.

When we have A $\leq_{D,\,\sigma}$ B $\wedge$ C $\leq_{B,\,\rho}$ D then we also have σ $\in$ B $\rightarrow$ A $\wedge$ ρ $\in$ D $\rightarrow$ C $\wedge$ B||C $\in$ $\mathscr{C}_{Safe}$. This is used in the premise of P.6.1.

Proposition P.6.1 Reliable refinements ensure safe names and reliable substitutions for specialised refinements

$\forall$ A, B, C, D, ρ, σ $\bullet$ RelNames(A, B, C) $\wedge$ σ $\in$ B $\rightarrow$ A $\wedge$ ρ $\in$ D $\rightarrow$ C $\wedge$

B||C $\in$ $\mathscr{C}_{Safe}$ $\Rightarrow$ Aρ||Cσ $\in$ $\mathscr{C}_{Safe}$ $\wedge$ σ $\in$ Bρ $\rightarrow$ Aρ

Proof:
RelNames(A, B, C) gives A||C $\in$ $\mathscr{C}_{Safe}$. Since we have A||C, B||C $\in$ $\mathscr{C}_{Safe}$ giving A.Dom $\cap$ C.Dom = Ø and B.Dom $\cap$ C-Dom = Ø, and when we have ρ $\in$ D $\rightarrow$ C then O.5.2.1 (observing objects' names are never keys in the substitution) then we have A.Dom = Aρ.Dom and B.Dom = Bρ.Dom. Then we also have σ $\in$ Bρ $\rightarrow$ Aρ. Since σ $\in$ B $\rightarrow$ A then we also get C.Dom = Cσ.Dom. Since the substitutions are reliable, then by observation O.5.2.2 (reliable substitution do not change slot names in configurations with safe names) slot names in the configurations are not changed by the substitutions. We then have Aρ||Cσ $\in$ $\mathscr{C}_{Safe}$.
❑

6.1.1 Reliability of specialised configurations

The next proposition shows that each D-name in every action from execution of sentences in A in A||D will be equal to or replaced by a C-name when the same sentence is executed in Aρ||C. We call this a complete specialisation of A relative to C.

Proposition P.6.2 : Specialisation is complete for reliable refinements

$\forall$ A, B, C, D, o, s, ρ, σ, α •
$A \leq_{D,\sigma} B \wedge C \leq_{B,\rho} D \wedge \alpha \in Traces(A||D\sigma) \wedge \alpha.exe \in A \wedge \alpha \in obs(D) \wedge @A(o:s) \wedge$

$A(o:s) \in \alpha.names \wedge A(o:s) \in D \Rightarrow (A(o:s))\rho \in C$

Proof:
Since $\alpha \in obs(D)$ and $A \leq_{D,\sigma} B$ then there is an action $\beta \in Traces(B||D)$ where $\alpha \sim_{D,\sigma'} \beta$. Then, by observation O.5.4.2 (properties of names in actions which are equal relative to a reliable substitution) all D-names in the actions are equal.
By observation O.5.4.4 (observability of observably similar actions) we have $\beta \in obs(D) \wedge \beta.exe \in B$. Since $C \leq_{B,\rho} D$, then, by definition of observably similar action sequences, there is some action γ such that $\gamma \equiv \beta\rho'$ and then O.5.4.4 gives $\gamma \in obs(C) \wedge \gamma.exe \in B$. If $\rho \neq \rho'$ then a pair is added to ρ to give ρ'. The pair is added if the actions are clone actions creating new D and C-objects and the new pair is then from the name of a new object in D to the name of a new object in C.

Since $\gamma \equiv \beta\rho'$, then by observation O.5.4.2 (properties of names in actions which are equal relative to a reliable substitution) each D-name in β will have a corresponding C-name in γ according to the mappings in ρ'. Therefore, when all D-names in β are mapped to C-names in γ and we also have that all D-names in α and β are equal, then all D-names in α are mapped to C-names in δ by the substitution ρ'. Ie, we have:

if $j \in \alpha \wedge j \in D$ then $j \in \beta$ and then $j\rho' \in \gamma \wedge j\rho' \in C$

This gives $(A(o:s))\rho \in C$, since $\rho \neq \rho'$ only when a pair is added to ρ to give ρ' where the pair defines a substitution of a D-name with a C-name. This gives $A\rho(o:s) \in C$ and the proposition holds.
❑

For the substitution proposition to hold we must have RelMessageSend(Aρ, Cσ) and RelMethodLookup(Aρ, Cσ). Proposition P.6.3 and P.6.5 show that observably similar actions from A||Dσ and Aρ||Cσ ensure reliable message sending and reliable method lookup from Aρ when combined with C provided A has reliable message sending when combined with D. If these were not true then RelMessageSend(Aρ, Cσ) and RelMethodLookup(Aρ, Cσ) have to be checked explicitly and that would ruin the basic intention of the substitutability proposition: that A and C can be developed separately both in space and time while retaining reliable substitution.

Proposition P.6.3 Reliable refinements ensures reliable message sending in specialised refinements

$\forall$ A, B, C, D, ρ, σ •
$RelNames(A, B, C) \wedge RelNames(C, D, A) \wedge A \leq_{D,\sigma} B \wedge C \leq_{B,\rho} D \Rightarrow RelMessageSend(A\rho, C\sigma)$

Proof:
To show RelMessageSend(Aρ, Cσ) we must show that no send sentence in A will give an error action due to no appropriate method found in Cσ, ie, we must show:

$\forall$ i, t, w • $i \in A \wedge$
(($\exists$ S_1, S_2, $p_1,\ldots,p_n$ •
$A\rho(i).Body \equiv S_1 \$ t!w(p_1,\ldots,p_n); S_2) \wedge A(i:t)\rho \in C\sigma \Rightarrow C(A(i:t)\rho:A(i:w)\rho)\sigma \in C\sigma$

$A \leq_{D,\sigma} B$ gives RelMessageSend(A, Dσ) which gives $A(i:t) \in D\sigma \Rightarrow D\sigma(A(i:t):A(i:w)) \in D$. The action from the message-send sentence will therefore be a message-send action, never an error action in A||Dσ.

Proposition P.6.2 gives $A(i:t) \in D\sigma \Rightarrow A(i:t)\rho \in C\sigma$ for all i, t where $A(i:t) \in D\sigma$.

Assume that the message-send action from the object named i in A in A||Dσ is $\alpha \equiv i\text{->}o!m(\bar{p})$. Since this action is observable from D, then by definition of $A \leq_{D,\sigma} B$ there will be some β from B||D such that $\alpha \sim_{D,\sigma} \beta$. Then we have

$\beta \equiv j$ -> o!m($\bar{p}\sigma$) where $j \in B$

By definition of $C \leq_{B, \rho} D$ there is some action γ from $B\rho||C$ such that

$\gamma \equiv j$ -> oρ!m($\bar{p}\sigma\rho$) where o$\rho \in C$

By proposition P.5.3.1 we have $A\rho(i:t) = (A(i:t))\rho$, and then since the receiver in α is o which was the value of A(i:t) and $o \notin D\sigma$, then the receiver in the action from execution of the corresponding sentence in $A\rho$ will be oρ. Also, by O.5.2.2 (reliable substitutions do not change slot names) the method selector will be m since $A\rho(i:w) = A(i:w)$. Also, the same number of parameters will be found in all actions.

By proposition P.5.3.2 we have $C(o\rho, m) = C\sigma(o\rho, m)$, ie, the same method object will be found in C and $C\sigma$. Then we have $C(A(i:t)\rho:A(i:w)\rho)\sigma \in C\sigma$ and the proposition hods.
❑

The next proposition shows an important property of complete specialisation which is a result of the reliability requirements. The proposition shows that we can deduce the action α which will come from execution of a sentence in a reliable configuration, eg, A in $A||D\sigma$, from the action δ from execution of the corresponding sentence in $A\rho||C$. The proposition shows that when the actions come from execution of a sentence in A, then the actions are observably similar relative to C.

Proposition P.6.4 Reliability gives equal actions relative to a reliable substitution

$\forall$ A, B, C, D, ρ, σ, δ •
RelNames(A, B, C) $\wedge$ RelNames(C, D, A) $\wedge$ $A \leq_{D,\sigma} B \wedge C \leq_{B,\rho} D \wedge$

$\delta \in$ Traces($A\rho||C\sigma$) $\wedge$ δ.exe $\in A \Rightarrow \exists \alpha$: Traces($A||D\sigma$) • $\delta \equiv \alpha\rho'$

where $\rho' = \rho + \{k/l\}$ when δ has the form a->i.s := k/oρ,
α has the form a->i.s := l/o where $o \in D$ and
$\rho' = \rho$ in all other cases

Proof:
Since there is some action δ.exe $\in A\rho$, there is an executable sentence in $A\rho$. Since the substitution does not change execution marks, there is also an executable sentence in A. Then we have $\exists \alpha$: Traces(A||D). By observation O.5.2.1 (observing objects' names are never keys in the substitution) we have

$\exists \alpha$: Traces(A||D) • δ.exe = α.exe

If both actions are error actions, we have $\delta \equiv \alpha\rho$.

Since the symbols in the sentences such as ! and := are not changed by the substitution, and by observation O.5.2.2 (reliable substitutions do not change slot names in configurations) we have that all sentences will be equal in the two configurations A and $A\rho$. Since δ.exe = α.exe. The two actions are from execution of the same sentence since there is only one executable sentence in each object. Then we have

either both actions are error actions and then we have $\delta \equiv \alpha\rho$,
or none are error actions and therefore both are the same kind of action, and then we must show $\delta \equiv \alpha\rho'$,
or one is an error action and the other is not, in which case the proposition does not hold

We first show that if none of the actions are error actions then we have $\delta \equiv \alpha\rho'$. Then we show that we will never have a case where only one of the actions are error actions.

When equal sentences are executed in A and $A\rho$, then by proposition P.5.3.1 (reliable substitutions give same slots and preserve "No external inheritance") and P.5.4.5 (same result of if-test when applying a reliable substitution) the same slots will be found in both cases and any differences in slot values is adjusted for by ρ. We then have $\delta \equiv \alpha\rho$ for all cases except one: The only case when ρ does not adjust for differences in names in the actions, is when the actions are clone actions cloning objects not in A. We then have a case where there is some i, s, k, l and o such that

$\delta \equiv$ e->i.s := k/oρ, $\alpha \equiv$ e->i.s := l/o and $o \in D$.

The difference in names of new objects is then adjusted for by adding {k/l} to the substitution ρ. We then have

$\delta \equiv \alpha\rho'$
where $\rho' = \rho + \{k/l\}$

In all other cases the names in the actions are slot values in A and $A\rho$ and proposition P.5.3.1 gives $\delta \equiv \alpha\rho$ and the proposition holds.

We next show that we will never have a case where one action is an error action and the other is not. The actions can come from different kinds of sentences which give different causes for errors as follows:

assignment	no slot owner
clone	no slot owner and unknown clone original
message-send	no slot owner , unknown message receiver and no appropriate method for a message receiver in A

The proof for this case is done by the following subcases:

Case 1) Error due to no slot owner
Case 2) Error due to unknown clone original or unknown message receiver
Case 3) Error due to no appropriate method for a message receiver

Case 1) Error due to no slot owner
Since there is no external inheritance in A and the action is from execution of a sentence in A we have for all objects i and slots s in A : that owner(A, i, s) $\in$ A holds.
Proposition P.5.3.1 gives owner(A, i, s) = owner(Aρ, iρ, sρ). We then have owner(Aρ, iρ, sρ) $\in$ A. Then neither of the actions will be error actions due to slot not found.

Case 2) Error due to unknown clone original or unknown message receiver
Assume that the clone original or receiver is given by A(i:s) and we have A(i:s) $\notin$ A||Dσ. By P.5.3.1 (reliable substitutions give same slots and preserve "No external inheritance") we have Aρ(i:s) = A(i:s)ρ.

If we have A(i:s) $\notin$ A||D then P.5.4.4 gives A(i:s)ρ $\notin$ C. Since ρ $\in$ D $\rightarrow$ C and A||D, A||C $\in$ $\mathscr{C}_{\text{Safe}}$, we have A(i:s)ρ $\notin$ Aρ. We then have A(i:s)ρ $\notin$ Aρ||C. Thus both actions will be error actions due to unknown clone original or unknown message receiver.

If A(i:s) $\in$ D, then by proposition P.6.2 we have A(i:s)ρ $\in$ C. Thus none of them are error actions due to unknown clone original or unknown message receiver.

If A(i:s) $\in$ A, then by observation O.5.2.1 (observing objects' names are never keys in the substitution) we have A(i:s)ρ $\in$ A. Then A-names are not changed and this gives A(i:s) = A(i:s)ρ. Thus none of the actions are error actions due to unknown clone original or unknown message receiver.

Case 3) Error due to no appropriate method for a message receiver
As shown above under Case 2) we have that if A(i:s) gives the receiver of the message then we have:
if A(i:s) $\in$ D then Aρ(i:s) $\in$ C and
if A(i:s) $\in$ A then Aρ(i:s) $\in$ A and A(i:s) = Aρ(i:s), ie, the same A-object is the receiver

If the receiver of the two actions are objects in D and C, then there might be errors due to no appropriate method for the message. However, we have RelMessageSend(A, D) and by proposition P.6.3 we have RelMessageSend(Aρ, Cσ). Therefore, there can be no errors due to no appropriate method for a message receiver in D and C. Therefore both actions can not be error actions due to no method found.

By proposition P.5.3.2 the same method object will be found in A and Aρ. Then, if the correct number of input-slots is found in the method in A, then the correct number of input-slots will also be found in Aρ. Therefore both actions will be error actions or both will be message-send actions.

We have then showed that either the two actions must be error actions or none of them are and then the proposition holds.
❏

Note that when we have δ $\in$ action(Aρ||C) $\wedge$ δ.exe $\in$ A $\wedge$ α $\in$ Traces(A||D) $\wedge$ δ $\equiv$ αρ' and the substitution is reliable, then by observation O.5.4.3 (equal actions relative to a substitution are observably similar) we have δ $\sim_{C,\rho'}$ α and then also δ $\leq_{C,\rho'}$ α.

Proposition P.6.5 Reliable method lookup is preserved by reliable substitutions

$\forall$ A, B, C, D, σ, ρ •
RelNames(A, B, C) $\wedge$ RelNames(C, D, A) $\wedge$ A $\leq_{D,\sigma}$ B $\wedge$ C $\leq_{B,\rho}$ D $\wedge$
RelMethodLookup(A, Dσ) $\Rightarrow$ RelMethodLookup(Aρ, Cσ)

Proof:
To show RelMethodLookup(Aρ, Cσ) we must show that for every A-observable message-send action from Aρ||C there will be a method which is found in Aρ. The implication is therefore only necessary to show when we have actions such that $\delta \in Traces(A\rho||C\sigma) \wedge \delta \in obs(A)$.

We consider all cases of where the executed sentence is found.
Case $\delta.exe \in A$:

Proposition P.6.4 (reliability gives equal actions relative to a reliable substitution) gives that there is some α such that $\alpha \in Traces(A||D\sigma) \wedge \delta \equiv \alpha\rho$.

By observation O.5.4.2 (properties of names in actions which are equal relative to a reliable substitution) the actions describe the same message to the same A-object. By proposition P.5.3.2 the same method object will be found in A and Aρ giving RelMethodLookup(Aρ, Cσ) for this case.

Case $\delta.exe \in C$

Proposition P.6.4 (reliability gives equal actions relative to a reliable substitution) gives that there is some γ such that $\gamma \in Traces(B\rho||C) \wedge \delta \equiv \gamma\sigma$.

Assume that the receiver in δ is an object in A named o. Since we have RelNames(C, D, A) giving that all A-names in C are also B-names and since the substitution σ substitutes from B-names to A-names, then the receiver in γ is either an object in B named o or an object named p where $o = p\sigma$, ie, $\gamma \in obs(B)$. Then the two actions have receivers not in C. By observation O.5.2.2 (reliable substitutions do not change slot names in configurations) the substitution does not substitute slot names, the actions have the same selector.

Since $C \leq_{B,\rho} D$ there is some $\beta \in Traces(B||D)$ where $\gamma \leq_{B,\rho'} \beta$. Since $\gamma.exe \in C$ and $\gamma \in obs(B)$ and the action is a message-send action, then $\beta.exe \in D$ and by definition of observable similarity we have $\gamma.dsc \equiv \beta\rho'.dsc$. Since the substitution ρ' is reliable relative to B, then observation O.5.4.2 (properties of names in actions which are equal relative to a reliable substitution) gives that the B-names are not changed and we then have $\beta \in obs(B)$ and the message selector and the receiver are the same in both actions.

Since $\beta.exe \in D$ and since we have $A \leq_{D,\sigma} B$, then there is some $\alpha \in Traces(A||D\sigma)$ where $\alpha \equiv \beta\sigma'$. By observation O.5.4.2 then $\alpha \in obs(A)$ and the message selector is the same in both actions. Also, since the receiver is an object in A, we will have the same receiver in both δ and α since we have the same B-receiver and the same substitution is applied to both actions. Then, by proposition P.5.3.2 the same method object will be found in both A and Aρ, giving RelMethodLookup(Aρ, Cσ) for this case.

❑

The next proposition shows that refinement configurations are reliable configurations when specialised with reliable substitutions.

Proposition P.6.6 Specialised reliable refinements are reliable configurations

$\forall$ A, B, C, D, σ, ρ •
RelNames(A, B, C) $\wedge$ RelNames(C, D, A) $\wedge$ $A \leq_{D,\sigma} B \wedge C \leq_{B,\rho} D \Rightarrow$ Reliable(Aρ, Cσ)

Proof:
To show Reliable(Aρ, Cσ) we must show noExt(Aρ), RelIfSentence(Aρ, Cσ), RelMessageSend(Aρ, Cσ) and RelMethodLookup(Aρ, Cσ).

Proposition P.5.3.1 (reliable substitutions give same slots and preserve "No external inheritance") gives noExt(Aρ).
Proposition P.5.4.6 (reliable if-sentences is preserved by reliable substitutions) gives RelIfSentence(Aρ, Cσ),
Proposition P.6.3 (reliable refinements ensures reliable message sending in specialised refinements) gives RelMessageSend(Aρ, Cσ) and
Proposition P.6.5 (reliable method lookup is preserved by reliable substitutions) gives RelMethodLookup(Aρ, Cσ).
This shows that the proposition holds.
❑

6.1.2 Derived substitutions and configurations

The next proposition shows that when we have two actions from execution of the corresponding sentences in A and Aρ and where the actions are observably similar relative to a substitution, then there are common derived configurations. The transitions for actions $\delta \in \text{Traces}(A\rho||C) \wedge \alpha \in \text{Traces}(A||D) \wedge \delta \leq_{A,\rho'} \alpha$ can then be written

$$A||D\sigma \xrightarrow{\alpha} A'||D' \wedge A\rho||C \xrightarrow{\delta} A'\rho'||C'$$

This property is important when proving the substitution proposition by induction, as is done in theorem T.6.1. Intuitively, this property reflects the necessity of having reliable substitutions, reliable method lookup and no external inheritance in order to be able to show properties of refinement configurations.

Proposition P.6.7 Observably similar actions give a common derived configuration

$\forall$ A, C, D, δ, ρ, α, β •
noExt(A) $\wedge$ $\rho \in D \rightarrow C \wedge A||D, A\rho||C \in \mathscr{C}_{\text{Safe}} \wedge$
RelMethodLookup(Aρ, C) $\wedge$

$\delta \in \text{Traces}(A\rho||C) \wedge \alpha \in \text{Traces}(A||D) \wedge \delta \leq_{A,\rho'} \alpha$
$\Rightarrow$
$\exists A' \bullet A||D\sigma \xrightarrow{\alpha} A'||D' \wedge A\rho||C \xrightarrow{\delta} A'\rho'||C'$

where $\rho' = \text{prime}(\rho, \delta, \alpha, C, D, A)$

Proof:
Cases:
Case 1) $\delta \notin$ obs(A) and then by definition of observable equality we have $\delta \equiv \alpha\rho'$
Case 2) $\delta \in$ obs(A) and then by definition of observable equality we have $\delta \sim_{A,\rho'} \alpha$

For all cases we have that when $\delta \sim_{A,\rho'} \alpha$ or $\delta \equiv \alpha\rho'$ then $\alpha.\text{exe} \sim_A \delta.\text{exe}$. Then
If $\delta.\text{exe} \notin$ A then also $\alpha.\text{exe} \notin$ A. Then none of the actions are from execution of sentences in A and there will therefore be no movement of execution marks in any of the A-configurations.
If $\delta.\text{exe} \in$ A then $\alpha.\text{exe} = \delta.\text{exe} \in A\rho$. Then, as argued in observation O.3.2 (equal actions from the same configuration give equal derived configurations) the corresponding sentences in A and Aρ will be executed and the execution mark moved correspondingly in the two configurations. Therefore the execution mark are found in corresponding places in A' and A'ρ'.

Case 1) $\delta \notin$ obs(A) and $\delta \equiv \alpha\rho'$

Case 1a) $\delta.\text{exe} \in$ C
Then $\delta \otimes$ A and then by proposition P.4.1.1 (silent actions are hidden actions and therefore do not change the observing configuration) we have A' ≡ A. Also, when $\delta \notin$ obs(A), then by observation O.5.5.1 (non-observed actions give equal substitutions and primed substitutions) we have ρ' = ρ and the proposition holds for this case.

Case 1b) $\delta.\text{exe} \in$ A
Since there is no external inheritance in A, then by observation O.5.3.1 (in configurations with no external inheritance, an action can only update slots within the configuration where the executed sentence is found) we only have the following case:
$\delta \equiv$ e->o!m($\overline{q}$)/k where o $\notin$ A
Then $\delta \equiv \alpha\rho'$ gives $\alpha \equiv$ e->oρ!m($\overline{p}\rho$)/k. Then no new objects will be created in A and no slots updated.
The only difference between A and derivations of A is the movement of the execution mark. As shown above, the execution mark is moved correspondingly in the two configurations. Therefore the execution marks are found in corresponding places in A' and A'ρ'. Thus the proposition holds for this case.

Case 2) $\delta \in$ obs(A)
Cases of δ:
message send to an object in A
update of a slot in A
assignment
clone of an object in A
clone of an object not in A
clone of an object in A and update of a slot not in A
error action in a sentence in A

Case 2A) message send to an object in A. Let $\delta \equiv$ f->o!m($\overline{q}$)/k

By definition of observably similar actions we then have:

$\alpha \equiv$ e->o!m($\overline{p}$)/k where $\overline{q} = \overline{p}\rho'$.

By proposition P.5.3.2 the same method object will be found in both A and Aρ. This object will be copied and placed in A'. By definition of observably similar actions, the new objects will get equal names in A' and A'ρ. The substitution ρ will adjust for any differences in parameter names in the two actions and then the proposition holds for this case.

Case 2B) update slot in A

Since there is no external inheritance in A, then by observation O.5.3.1 (in configurations with no external inheritance, an action can only update slots within the configuration where the executed sentence is found) we have one of the following cases of δ:

case 2B i) δ has the form e->l.s := j where $l \in A$

case 2B ii) δ has the form e->l.s := k/o where $l \in A$

Case 2B i) Assume $\delta \equiv$ e->l.s :$\equiv$ j where $l \in A$

We then have $\alpha \equiv$ e->(l.s := j) $\wedge$ i = jρ

Then the same slot will be updated by both actions and ρ will adjust for the difference in slot values, and the proposition holds for this case.

Case 2B ii) Assume $\delta \equiv$ e->l.s := k/o where $l \in A$

The action will either clone an objet in A or an object in C. In both cases the same slot will be updated by both actions and ρ' will adjust for the difference in names of new objects and slot values.

If an object in A is cloned, ie, $o \in A$, then both actions clone the same A-object and we have:

$\alpha \equiv$ e->(l.s:=k/o)

The same object will be cloned and the same slot will be updated by both actions, and the name of the new object is the same in both actions. This will give the same slot value for both actions, and the proposition holds for this case.

If δ clones an object in C, then we have:

$\alpha \equiv$ e->(l.s:=r/p) $\wedge$ $p \notin A \wedge o = p\rho \wedge k = r\rho'$

Then both actions are clones of objects not in A as ρ' only substitute from D-names to C-names. Then there will be no new object in derivations of A, and ρ' will adjust for the difference in slot values, and the proposition holds for this case.

Case 2C) δ clones an object in A and update a slot not in A, thus assuming $\delta \equiv$ e->l.s := k/o where $o \in A$ and $l \notin A$

By definition of observably similar actions we then have:

α.dsc $\equiv$ j.t:=k/o $\wedge$ $j \notin O$

Then the same A-object (the object named o) is copied as result of both actions. Also, the new objects get equal names. No slot is updated in A and $\rho' = \rho$. Therefore, a common A' can be found, and the proposition holds for this case.

Case 2D) error in a sentence in A, assuming $\delta \equiv$ e->error where $e \in A$

By definition of observably similar actions we then have $\alpha \equiv$ e->error. Then it will be the same object in A which terminated, and the proposition holds for this case.

❑

6.2 The Simple Substitution Theorem

6.2.1 The simple substitution theorem

The substitution proposition expressed for systems consisting of two configurations is stated as follows:

Theorem T.6.1: The simple substitution theorem

$$\forall\ A, B, C, D, \sigma, \rho \bullet RelNames(A, B, C) \wedge RelNames(C, D, A) \wedge$$
$$A \leq_{D,\sigma} B \wedge C \leq_{B,\rho} D \Rightarrow A\rho \leq_{C,\sigma} B\rho \wedge C\sigma \leq_{A,\rho} D\sigma$$

Because of symmetry it is only necessary to prove:

$$\forall\ A, B, C, D, \sigma, \rho \bullet RelNames(A, B, C) \wedge RelNames(C, D, A) \wedge$$
$$A \leq_{D,\sigma} B \wedge C \leq_{B,\rho} D \Rightarrow A\rho \leq_{C,\sigma} B\rho$$

By definition of $A\rho \leq_{C,\sigma} B\rho$, we can reformulate the simple substitution theorem as follows:

$$\forall\ A, B, C, D, \sigma, \rho \bullet$$
$$RelNames(A, B, C) \wedge RelNames(C, D, A) \wedge A \leq_{D,\sigma} B \wedge C \leq_{B,\rho} D$$
$$\Rightarrow$$
$$(\forall\ \bar{\delta} : Traces(A\rho||C\sigma)\ \exists\ \bar{\gamma} : Traces(B\rho||C) \bullet$$

$$A\rho||C\sigma, B\rho||C \in \mathscr{C}_{Safe} \wedge \sigma \in B\rho \rightarrow A\rho \wedge Reliable(A\rho, C\sigma, \bar{\delta})$$
$$\wedge\ \bar{\delta} \leq_{C',\sigma'} \bar{\gamma} \wedge (endColab(A\rho, C\sigma, \bar{\delta}) \Rightarrow endColab(B\rho, C, \bar{\gamma})))$$

where $\sigma' = prime(\sigma, \bar{\delta}, \bar{\gamma}, A\rho, B\rho, C)$

For the conclusion to hold we must show that for all A, B, C, D, σ and ρ such that the premise of the theorem holds, then for every action sequences $\bar{\delta}$ in $Traces(A\rho||C\sigma)$ there is an action sequence $\bar{\gamma}$ in $Traces(B\rho||C)$ such that $Reliable(A\rho, C\sigma, \bar{\delta}) \wedge \bar{\delta} \leq_{O,\sigma} \bar{\gamma} \wedge (endColab(A\rho, C\sigma, \bar{\delta}) \Rightarrow endColab(B\rho, C, \bar{\gamma}))$.

We prove this by induction over the length of $\bar{\delta}$, ie, induction over the length of $\bar{\delta}$ (written $\#\bar{\delta}$). Since the induction is done on the length of $\bar{\delta}$, $\bar{\delta}$ is moved to the front and the proposition is restated as follows:

$$\forall\ \bar{\delta}, A, B, C, D, \sigma, \rho \bullet$$
$$\bar{\delta} \in Traces(A\rho||C\sigma) \wedge RelNames(A, B, C) \wedge A \leq_{D,\sigma} B \wedge C \leq_{B,\rho} D$$
$$\Rightarrow$$
$$(A\rho||C\sigma, B\rho||C \in \mathscr{C}_{Safe} \wedge \sigma \in B\rho \rightarrow A\rho \wedge Reliable(A\rho, C\sigma, \bar{\delta}) \wedge$$
$$\exists\ \bar{\gamma} : Traces(B\rho||C) \bullet \bar{\delta} \leq_{C',\sigma'} \bar{\gamma} \wedge (endColab(A\rho, C\sigma, \bar{\delta}) \Rightarrow endColab(B\rho, C, \bar{\gamma})))$$

where $\sigma' = prime(\sigma, \bar{\delta}, \bar{\gamma}, A\rho, B\rho, C)$

The induction base is $\#\bar{\delta} = 0$ and the induction hypothesis says that the proposition holds for all action sequences in traces which are shorter than $\#\bar{\delta}$ where $\#\bar{\delta} = n + 1$. The induction base is shown in section 6.2.2 while the induction step is shown in section 6.2.3.

6.2.2 The induction base of the theorem

When $\#\bar{\delta} = 0$, then since the empty action sequence is found in all traces, the definition of the refinement relation gives that we must show the following for the induction base:

$$(A\rho||C\sigma, B\rho||C \in \mathscr{C}_{Safe} \wedge \sigma \in B\rho \rightarrow A\rho \wedge Reliable(A\rho, C\sigma) \wedge (endColab(A\rho, C\sigma) \Rightarrow endColab(B\rho, C))) \Rightarrow A\rho \leq_{C,\sigma} B\rho$$

By this we prove the induction base of the theorem as follows:

Induction base of theorem T.6.1: $\#\bar{\delta} = 0$

To show $A\rho \leq_{C,\sigma} B\rho$ when $\#\bar{\delta} = 0$ then by definition of $\leq_{C,\sigma}$ it suffices to show:

$$\forall\ A, B, C, D, \sigma, \rho \bullet$$
$$A\rho||C\sigma \in \mathscr{C}_{Safe} \wedge RelNames(A, B, C) \wedge RelNames(C, D, A) \wedge A \leq_{D,\sigma} B \wedge C \leq_{B,\rho} D$$
$$\Rightarrow$$
$$(A\rho||C\sigma, B\rho||C \in \mathscr{C}_{Safe} \wedge \sigma \in B\rho \rightarrow A\rho \wedge Reliable(A\rho, C\sigma) \wedge (endColab(A\rho, C\sigma) \Rightarrow endColab(B\rho, C)))$$

Proof:
Proposition P.6.1 (reliable refinements ensure safe names and reliable substitutions for specialised refinements) gives $\sigma \in B\rho \rightarrow A\rho \wedge A\rho||C\sigma \in \mathscr{C}_{Safe}$. From $C \leq_{B,\rho} D$ we have $B\rho||C \in \mathscr{C}_{Safe}$.

$A \leq_{D,\sigma} B \wedge C \leq_{B,\rho} D$ and proposition P.6.6 (specialised reliable refinements are reliable configurations) gives $Reliable(A\rho, C\sigma)$.

When $endColab(A\rho, C\sigma)$ we have by definition:

$$\forall\ \bar{\delta}' : Traces(A\sigma||C\rho) \bullet \bar{\delta}'.exe \in A \Rightarrow \bar{\delta}' \otimes C$$

Next we show $(endColab(A\rho, C\sigma) \Rightarrow endColab(B\rho, C))$ when $A \leq_{D,\sigma} B \wedge C \leq_{B,\rho} D$. First we show that we have $endColab(A, D\sigma)$ by induction on the length of the longest $\bar{\delta}'$.

Base : $\#\bar{\delta}' = 0$
We the longest action sequence from $A\rho||C\sigma$ is the empty action sequence, then the longest action sequence from $A\rho$ will also be empty. We then have $endColab(A, D\sigma)$.

Step: $\#\bar{\delta}' = m$

Induction hypothesis: we have $endColab(A, D\sigma)$ when the length of the longest $\bar{\delta}'$ is shorter than m.

Assume $\bar{\delta}' = \delta\ \&\ \bar{\delta}''$. We have $\delta.exe \in A$, $\delta \otimes C$ and $endColab(A'\rho, C\sigma)$, where A' is A after δ, from definition of endColab(). The induction hypothesis then gives $endColab(A', D\sigma)$. By proposition P.6.4 we have some α from A in $A||D$ where $\delta \equiv \alpha\rho$. By observation O.5.4.4 (observability of observably similar actions) we then have $\alpha \otimes D$. This gives $endColab(A, D\sigma)$.

We have then shown $endColab(A, D\sigma)$ for any length of the longest $\bar{\delta}'$. From $A \leq_{D,\sigma} B$ we then have endColab(B, D) which gives:

$$\forall\ \bar{\beta} : Traces(B||D) \bullet \bar{\beta}.exe \in B \Rightarrow \bar{\beta} \otimes D$$

By definition of the refinement relation with specialisation and since we have $C \leq_{B,\rho} D$ and by proposition P.5.4.7 we have that each action from a sentence in $B\rho$ in $B\rho||C$ will be equal to an action from the corresponding sentence in B in B||D relative to the substitution ρ. We then have:

$$\forall\ \bar{\gamma} : Traces(B\rho||C) \bullet \bar{\gamma}.exe \in B \Rightarrow \bar{\gamma} \equiv \bar{\beta}\rho$$

Then by observation O.5.4.4 (observability of observably similar actions) we have $\bar{\gamma} \otimes C$. We then have $endColab(B\rho, C)$ which shows the induction base of the theorem.
❑

6.2.3 The induction step of the theorem

For the induction step assume $\#\bar{\delta} > 0$. We next show two lemmas related to proving properties of the first action in an action sequence in Traces(Bρ||C) based on assumptions about the first action in $\bar{\delta}$ from Traces(Aρ||Cσ) where $\#\bar{\delta} > 0$.

The next lemma, L.6.2.1 is a special case of lemma L.6.2.2. By this lemma we avoid a very long proof of lemma L.6.2.2. The part shown in L.6.2.1 is an important part of showing the substitution proposition. The lemma shows that when a sentence in A gives an action which is similar to an action from a sentence in B as observed from D (ie, we have $\alpha \sim_{D,\sigma'} \beta$), then the same sentences executed in Aρ and Bρ will give observably similar actions as observable from C (ie, we have $\delta \leq_{C,\sigma'} \gamma$). To show this, actions from B which are observable from D and C have to be taken into account. The proposition and the proof are therefore quite complicated in that actions from all four configurations are involved. Two and two of the actions are from execution of the same sentence, two are observably similar and then each of these actions are equal to one of the other actions relative to the appropriate substitutions, ie, we have a situation as follows:

$$\alpha \in \text{Traces}(A||D\sigma) \wedge \beta \in \text{Traces}(B||D) \wedge \gamma \in \text{Traces}(B\rho||C) \wedge \delta \in \text{Traces}(A\rho||C\sigma) \wedge$$

$$\alpha.\text{exe} = \delta.\text{exe} \in A \wedge \alpha \in \text{obs}(D) \wedge \beta.\text{exe} = \gamma.\text{exe} \in D \wedge$$

$$\alpha \sim_{D,\sigma'} \beta \wedge \delta \equiv \alpha\rho' \wedge \gamma \equiv \beta\rho''$$

where $\rho' = \text{prime}(\rho, \delta, \alpha, C\sigma, D\sigma, A)$, $\rho'' = \text{prime}(\rho, \gamma, \beta, C, D, B)$ and $\sigma' = \text{prime}(\sigma, \alpha, \beta, A, B, D)$

Lemma L.6.2.1 then concludes $\delta \leq_{C,\sigma'} \gamma$. This lemma also shows that the prime of the substitution ρ as found when creating it from ρ and the actions from B, is equal to the prime of the substitution ρ as found when creating it from ρ and the actions from A, ie,

$$\text{prime}(\rho, \delta, \alpha, C\sigma, D\sigma, A) = \text{prime}(\rho, \gamma, \beta, C, D, B).$$

The assumptions in the proposition will hold when the reliability requirements hold and when the premise of the substitution proposition holds. We have that:

$\alpha \sim_{D,\sigma'} \beta$ where $\sigma' = \text{prime}(\sigma, \alpha, \beta, A, B, D)$,
when A is a reliable refinement to B relative to D with σ and $\alpha \in \text{obs}(D)$, ie, $A \leq_{D,\sigma} B$

$\delta \equiv \alpha\rho'$ where $\rho' = \text{prime}(\rho, \delta, \alpha, C\sigma, D\sigma, A)$,
when the two actions stem from execution of the same sentence in A and Aρ respectively

$\gamma \equiv \beta\rho''$ where $\rho'' = \text{prime}(\rho, \gamma, \beta, C, D, B)$,
when the two actions stem from execution of the same sentence in B and Bρ respectively, and C is a reliable refinement of D relative to B with the substitution ρ, ie, $C \leq_{B,\rho} D$

The conclusion of the lemma is necessary in order to show that Aρ is a reliable refinement of Bρ relative to Cσ, ie, $A\rho \leq_{C,\sigma} B\rho$, which is part of the conclusion of the substitution proposition.

In the proof of the next lemma, the substitutions are combined and the equality $\beta\sigma\rho \equiv \beta\rho\sigma$ is used. Then it must be possible to distinguish which names come from which parts of the configurations so that the proper substitution is used, and that just one substitution is used for each free name. The following observation explains why $\beta\sigma\rho \equiv \beta\rho\sigma$ can be used in the proof of the next lemma.

Observation O.6.1 : About combining reliable substitutions

When we have

$$A||D, B||D, B||C, A||C \in \mathscr{C}_{\text{Safe}} \wedge \sigma \in B \rightarrow A \wedge \rho \in D \rightarrow C$$

the following restrictions on the substitutions σ and ρ hold:

$$\text{keys}(\sigma) \cap \text{keys}(\rho) = \emptyset \wedge \text{values}(\sigma) \cap \text{keys}(\rho) = \emptyset \wedge \text{values}(\rho) \cap \text{keys}(\sigma) = \emptyset$$

Because we then have that keys in one substitution are never found as values in the other substitution, and the substitutions have separate keys, then for any action β from B||D we have

$$\beta\sigma\rho \equiv \beta\rho\sigma$$

Lemma L.6.2.1 Property of equal and observably similar actions

$\forall$ A, B, C, D, α, δ, β, δ, γ, σ, ρ •

A||D, B||D, B||C, A||C $\in \mathscr{C}_{Safe} \wedge$

$\sigma \in B \rightarrow A \wedge \rho \in D \rightarrow C \wedge$ Reliable(A, Dσ) $\wedge$ Reliable(C, Bρ) $\wedge$ RelMessageSend(Aρ, Cσ) $\wedge$

$\delta \in$ Traces(Aρ||Cσ) $\wedge$ $\beta \in$ Traces(B||D) $\wedge$ $\alpha \in$ Traces(A||Dσ) $\wedge$ $\gamma \in$ Traces(Bρ||C) $\wedge$

δ.exe $\in$ A $\wedge$ $\delta \in$ obs(C) $\wedge$ α.exe = δ.exe $\in$ A $\wedge$ $\alpha \in$ obs(D) $\wedge$ $\alpha \sim_{D,\sigma'} \beta \wedge \delta \equiv \alpha\rho' \wedge \gamma \equiv \beta\rho''$

$\Rightarrow \rho' = \rho'' \wedge \delta \leq_{C,\sigma'} \gamma$

where ρ' = prime(ρ, δ, α, C, D, A), ρ'' = prime(ρ, γ, β, C, D, B) and σ' = prime(σ, α, β, A, B, D)

Proof:

First we show that δ.exe $\in$ A and γ.exe $\in$ B gives δ.exe $\sim_C$ γ.exe:

We have α.exe $\in$ A. By O.5.4.2 (properties of names in actions which are equal relative to a reliable substitution) we then have β.exe $\in$ B, δ.exe $\in$ A and γ.exe $\in$ B. This gives δ.exe $\sim_C$ γ.exe since σ' substitute from B names to A names.

Next we show $\rho' = \rho''$ and $\delta \leq_{C,\sigma'} \gamma$ for the description parts of the actions.

Since we have $\gamma \equiv \beta\rho''$, then by observation O.5.4.3 (equal actions relative to a substitution are observably similar) we have $\gamma \sim_{B,\rho''} \beta$ which gives $\gamma \leq_{B,\rho''} \beta$.

Since we have no external inheritance in A and $\alpha \in$ obs(D) and α.exe $\in$ A then by observation O.5.3.1 (in configurations with no external inheritance, an action can only update slots within the configuration where the executed sentence is found) one of the following two cases must occur:

Case 1) α.dsc $\equiv$ o!x($\bar{q}$)/k $\wedge$ o $\in$ D

Case 2) α.dsc $\equiv$ (i.s:=k/o) $\wedge$ o $\in$ D

Case 1) α.dsc $\equiv$ o!x($\bar{q}$)/k

By definition of $\alpha \sim_{D,\sigma'} \beta$ we then have β.dsc $\equiv$ o!x($\bar{p}$)/k $\wedge$ $\bar{q} = \bar{p}\sigma'$ and $\beta \notin$ obs(B). We then have α.dsc $\equiv \beta\sigma'$.dsc.

When $\beta \notin$ obs(B) and γ.dsc $\equiv \beta\rho''$.dsc then O.5.4.2 (properties of names in actions which are equal relative to a reliable substitution) gives $\gamma \notin$ obs(B). When $\gamma \notin$ obs(B) then O.5.5.1 (non-observed actions give equal substitutions and primed substitutions) gives ρ = prime(ρ, γ, β, C, D, B).

Similarly we have ρ = prime(ρ, δ, α, C, D, A). We then have $\rho' = \rho'' = \rho$ for this case. We then have:

α.dsc $\equiv \beta\sigma'$.dsc $\wedge$ δ.dsc $\equiv \alpha\rho$.dsc $\wedge$ γ.dsc $\equiv \beta\rho$.dsc

Since there are safe names and reliable substitutions, we can replace α with $\beta\sigma'$ in δ.dsc $\equiv \alpha\rho$.dsc and get

δ.dsc $\equiv \beta\sigma'\rho$.dsc.

Observation O.6.1 (about combining reliable substitutions) gives $\beta\sigma'\rho \equiv \beta\rho\sigma'$ and then δ.dsc $\equiv \beta\rho\sigma'$.dsc. When δ.dsc $\equiv \beta\rho\sigma'$.dsc and γ.dsc $\equiv \beta\rho$.dsc, then $\beta\rho'$.dsc can be replaced with γ.dsc which gives δ.dsc $\equiv \gamma\sigma'$.dsc. When we then have $\delta \in$ obs(C), δ.exe $\sim_C$ $\gamma\sigma'$.exe and δ.dsc $\equiv \gamma\sigma'$.dsc, this gives $\delta \leq_{C,\sigma'} \gamma$ for this case.

Case 2) α.dsc $\equiv$ i.s:=k/o $\wedge$ o $\in$ D

By definition of observably similar actions we then have $\beta\sigma'$.dsc $\equiv$ j.t:=k/o $\wedge$ i,j $\notin$ D $\wedge$ o $\in$ D

From $\delta \equiv \alpha\rho'$, reliable substitution and the definition of prime substitutions we have δ.dsc $\equiv$ i.s:=kρ'/oρ'.

Similarly we have γ.dsc $\equiv$ j.t:=kρ''/oρ''.

By observation O.4.3.2 (simplifying assumption about names of new objects), the name k is not found in any of the configurations A, B, C or D. This gives

δ.dsc $\equiv$ i.s:=l/p is a legal transition from Aρ||Cσ and

γ.dsc $\equiv$ j.t:=l/p is a legal transition from Bρ||C,

where {p/o} is an element in ρ and {l/k} is the new element in the prime substitutions.

By definition of prime substitutions we have $\rho' = \rho''$.

This also gives p $\in$ C and then δ, $\gamma \in$ obs(C). Since δ.exe $\sim_C$ γ.exe was shown above and by definition of observably similar actions, this gives $\delta \sim_{C,\sigma'} \gamma$. This gives $\delta \leq_{C,\sigma'} \gamma$ and the lemma holds for this case.

❑

The next lemma is used to show the induction step of the substitution theorem T.6.1. This lemma shows properties of the actions in Bρ||C when properties of actions from sentences in A in Aρ||Cσ and A||Dσ are known. Particularly it is shown that there is an action sequence $\bar{\gamma}$ from Bρ||C which is observably similar to an action δ from Aρ||Cσ, ie, $\delta \leq_C \bar{\gamma}\sigma'$. In addition various properties of the primed substitutions and derived configurations are shown. It would be possible to split this lemma into smaller lemmas, where each lemma

concluded part of the conclusions of this lemma. This is not done since to prove one part of the conclusion, it would be necessary to prove many of the same properties as stated in the other parts. Therefore the parts are all shown at the same time in the same lemma:

Lemma L.6.2.2 Reliable refinements give observably similar actions and common derived configurations

$\forall$ A, B, C, D, σ, ρ, δ • RelNames(A, B, C) $\wedge$ RelNames(C, D, A) $\wedge$
$A \leq_{D,\sigma} B \wedge C \leq_{B,\rho} D \wedge \delta \in Traces(A\rho||C\sigma) \wedge \delta.exe \in A$
$\Rightarrow$
$\exists\ \alpha : Traces(A||D\sigma),\ \bar{\gamma} : Traces(B\rho||C), \bar{\beta} : Traces(B||D)$ •

(i) $\delta \leq_{C,\sigma'} \bar{\gamma} \wedge \delta \leq_{A,\rho'} \alpha \wedge \alpha \leq_{D,\sigma'} \bar{\beta} \wedge \bar{\gamma} \leq_{B,\rho'} \bar{\beta} \wedge$

(ii) $prime(\rho, \bar{\gamma}, \bar{\beta}, C, D, B) = prime(\rho, \delta, \alpha, C, D, A) \wedge$
$prime(\sigma, \alpha, \bar{\beta}, A, B, D) = prime(\sigma, \delta, \bar{\gamma}, A, B, C) \wedge$

(iii) $\exists$ A', B', C' D', B'', D'' •
$A\rho||C\sigma \xrightarrow{\delta} A'\rho||C'\sigma' \wedge A||D\sigma \xrightarrow{\alpha} A'||D' \wedge B\rho||C \xrightarrow{\bar{\gamma}} B'||C' \wedge B||D \xrightarrow{\bar{\beta}} B''||D'' \wedge$

(iv) $D'.Dom = D''.Dom \wedge B'.Dom = B''.Dom$

where $\sigma' = prime(\sigma, \delta, \bar{\gamma}, A, B, C)$ and $\rho' = prime(\rho, \delta, \alpha, C, D, A)$

Point (iii) states that there are common derived configurations A' and C' resulting from actions α, $\bar{\beta}$, $\bar{\gamma}$ and δ.

Proof:
Small roman numbers in parenthesis, ie, (ii), in the beginning of a line indicate that the line states the corresponding part of the conclusion of the lemma.

Proposition P.6.4 (reliability gives equal actions relative to a reliable substitution) gives:

$\exists\ \alpha : Traces(A||D\sigma) \bullet \delta \equiv \alpha\rho'$ where $\rho' = prime(\rho, \delta, \alpha, C, D, A)$

By observation O.5.4.1 (reliable substitutions relative to configurations are also reliable relative to sets of object names) we have RelSubst(ρ, A.Dom). Then, by observation O.5.4.3 (equal actions relative to a substitution are observably similar) we have $\delta \sim_{A,\rho} \alpha$. Then by definition of observably similar action sequences, we have:
(i) $\delta \leq_{A,\rho'} \alpha$

Proposition P.6.7 (observably similar actions give a common derived configuration) gives
$\exists$ A', C', D' • $A||D\sigma \xrightarrow{\alpha} A'||D' \wedge A\rho||C\sigma \xrightarrow{\delta} A'\rho'||C'$

The rest of the proof is divided in two cases:
Case 1) $\delta \notin obs(C)$, in which case we have $\delta \in obs(A)$
Case 2) $\delta \in obs(C)$

Case 1) $\delta \notin obs(C) \wedge \delta \in obs(A)$
Let $\bar{\beta}$ = <> and let $\bar{\gamma}$ = <>. Then we have
(i) $\delta \leq_{C,\sigma'} \bar{\gamma} \wedge \alpha \leq_{D,\sigma'} \bar{\beta} \wedge \bar{\gamma} \leq_{B,\rho'} \bar{\beta}$
When the action sequences are empty, they are not observable, and when $\delta \notin obs(C)$, then observation O.5.5.1 (non-observed actions give equal substitutions and primed substitutions) gives:
(ii) $prime(\sigma, \alpha, \bar{\beta}, A, B, D) = prime(\sigma, \delta, \bar{\gamma}, A, B, C) = \sigma$ and
(ii) $prime(\rho, \bar{\gamma}, \bar{\beta}, C, D, B) = prime(\rho, \delta, \alpha, C, D, A) = \rho$

When $\delta.exe = \alpha.exe \in A$, $\alpha \notin obs(D)$, $\delta \notin obs(C)$, $\bar{\beta}$ = <> and $\bar{\gamma}$ = <> then proposition P.4.1.1 (silent actions are hidden actions and therefore do not change the observing configuration) gives:

(iii) $A\rho||C\sigma \xrightarrow{\delta} A'\rho||C\sigma \wedge A||D\sigma \xrightarrow{\alpha} A'||D\sigma \wedge B\rho||C \xrightarrow{\bar{\gamma}} B||C \wedge B||D \xrightarrow{\bar{\beta}} B||D$

Obviously part (iv) of the lemma holds since we have D' = D, D'' = D, B' = B and B'' = B.

We have then shown all parts, and the lemma holds for this case.

Case 2) $\delta \in$ obs(C)
When $\delta \leq_{A,\rho'} \alpha$ and $\delta \in$ obs(C) then observation O.5.4.4 (observability of observably similar actions) gives
$\alpha \in$ obs(D).

When $\alpha \in$ Traces(A||Dσ) and since A $\leq_{D,\sigma}$ B holds then we also have

(i) $\exists\ \overline{\beta}$: Traces(B||D) • $\alpha \leq_{D,\sigma'} \overline{\beta}$

where $\overline{\beta} = <\beta_1,..., \beta_n>$ where n ≥ 1 and since $\alpha \in$ obs(D) and α.exe $\in$ A then by observation O.5.4.5 (properties of similar observable action sequences) we can assume:

$<\beta_1,..., \beta_{n-1}> \otimes D \wedge \alpha \leq_{D,\sigma'} \beta_n \wedge \beta_n \in$ obs(D) $\wedge\ \beta_n$.exe $\in$ B for i $\in$ 1..n

Since C $\leq_{B,\rho}$ D holds then

(i) $\exists\ \bar{\gamma}$: Traces(Bρ||C) • $\bar{\gamma} \leq_{B,\rho''} \overline{\beta}$ where ρ" = prime(ρ, $\bar{\gamma}$, $\overline{\beta}$, C, D, B)

Let $\bar{\gamma} = <\gamma_1, .., \gamma_m>$. Since $\overline{\beta}$ from execution of sentences in B, then m = n. Then observation O.5.4.4 (observability of observably similar actions) gives γ_j.exe $\in$ B, $<\gamma_1,...,\gamma_{n-1}> \otimes$ C and $\gamma_n \in$ obs(C). Then observation O.5.5.1 (non-observed actions give equal substitutions and primed substitutions) gives:

prime(ρ, $<\gamma_1,..,\gamma_{n-1}>$, $<\beta_1,..,\beta_{n-1}>$, C, D, B) = ρ

Since the n-1 first actions do not affect the substitutions, then
ρ" = prime(ρ, $\bar{\gamma}$, $\overline{\beta}$, C, D, B) = prime(ρ, γ_n, β_n, C, D, A)
By observation O.5.4.5 (properties of similar observable action sequences) we have $\gamma_n \leq_{B,\rho''} \beta_n$.
Since $\gamma_n \in$ obs(C) and γ_n.exe $\in$ B, then the definition of observably similar actions gives $\gamma_n \equiv \beta_n\rho''$.

We now have α.exe $\in$ A $\wedge\ \alpha \in$ obs(D) $\wedge\ \alpha \leq_{D,\sigma'} \beta_n \wedge \delta \equiv \alpha\rho' \wedge \gamma_n \equiv \beta_n\rho''$. Since $\alpha \in$ obs(D) then by definition of observably similar action sequences we have $\alpha \sim_{D,\sigma'} \beta_n$. Proposition L.6.2.1 (property of equal and observably similar actions) gives:

$\delta \leq_{C,\sigma'} \gamma_n \wedge$ prime(ρ, γ_n, β_n, C, D, B) = prime(ρ, δ, α, C, D, A)

Since $<\gamma_1,...,\gamma_{n-1}> \otimes$ C and $\delta \leq_{C,\sigma'} \gamma_n$, then $\bar{\gamma}$ is such that:
(i) $\delta \leq_{C,\sigma'} \bar{\gamma}$

Above prime(ρ, γ_n, β_n, C, D, B) = prime(ρ, δ, α, C, D, A) and
prime(σ, $\bar{\gamma}$, $\overline{\beta}$, A, B, D) = prime(σ, γ_n, β_n, A, B, D) were shown. Then we can conclude:

(ii) prime(ρ, $\bar{\gamma}$, $\overline{\beta}$, C, D, B) = prime(ρ, δ, α, C, D, A)

Observation O.5.5.1 (non-observed actions give equal substitutions and primed substitutions) gives
σ = prime(σ, α, $<\beta_1,.. ,\beta_{n-1}>$, A, B, D) and σ = prime(σ, δ, $<\gamma_1,.. ,\gamma_{n-1}>$, A, B, D).
Any differences in prime(σ, α, $\overline{\beta}$, A, B, D) and prime(σ, δ, $\bar{\gamma}$, A, B, D) would then be caused by different B-names in β_n and γ_n or different A-names in α and β_n. Since $\gamma_n \equiv \beta_n\rho''$ as shown above, then by observation O.5.4.2 (properties of names in actions which are equal relative to a reliable substitution) the B-names must be equal in this case. Since $\delta \equiv \alpha\rho'$ then by observation O.5.4.2 (properties of names in actions which are equal relative to a reliable substitution) the A-names in these two actions are equal. This gives:
(ii) prime(σ, α, $\overline{\beta}$, A, B, D) = prime(σ, δ, $\bar{\gamma}$, A, B, C)

We will now show part iii: there are common derived configurations:
Initially we showed that
(iii) $\exists$ A', C', D' • Aρ||Cσ $\xrightarrow{\delta}$ A'ρ'||C'σ' $\wedge$ A||Dσ $\xrightarrow{\alpha}$ A'||D'.

We also have:
Bρ||C $\xrightarrow{\bar{\gamma}}$ B'||C" $\wedge$ B||D $\xrightarrow{\bar{\beta}}$ B"||D"

We must then show that we have a common derived configuration from δ and $\bar{\gamma}$. Since $\gamma_1,...,\gamma_{n-1}$ are silent actions, then by proposition P.4.1.1 (silent actions are hidden actions and therefore do not change the observing configuration) the actions do not affect C. Since we have $\delta \leq_{C,\sigma'} \gamma_n$, then by P.6.7 (observably similar actions give a common derived configuration) we have common derived configurations from δ and γ_n and we have C'' ≡ C' and

(iii) $\exists$ C' • Aρ||Cσ $\xrightarrow{\delta}$ A'ρ'||C'σ' ∧ Bρ||C $\xrightarrow{\bar{\gamma}}$ B'||C'.

We have then found common derived configurations (part iii).

Next we show part (iv):
Since we have $\alpha \leq_{D,\sigma'} \bar{\beta}$, then proposition P.5.4.3 gives

(iv) D'.Dom = D''.Dom.

Since we have $\gamma_i \equiv \beta_i\rho''$ for all actions from sentences in B. By proposition P.5.5.1 we have that the substitution is reliable for all derived configurations and then observation O.5.4.3 (equal actions relative to a substitution are observably similar) gives $\gamma_i \leq_{B',\rho''} \beta_i$. Then proposition P.5.4.3 gives:

(iv) B'.Dom = B''.Dom

We have then shown all parts of the proposition for all cases, and the lemma holds.
❑

Note that there are no common derived configurations B' and D' such that:

$$A||D\sigma \xrightarrow{\alpha} A'||D'\sigma' \wedge B||D \xrightarrow{\bar{\beta}} B'||D' \wedge A\rho||C\sigma \xrightarrow{\delta} A'\rho'||C'\sigma' \wedge B\rho||C \xrightarrow{\bar{\gamma}_1} B'\rho'||C'$$

The reason for this is that, eg, the method found for a message to a D-object might be different in A||Dσ and B||D since we do not require reliable method lookup in D. Then the methods can be found in A and B, and the methods can therefore have, eg, different slot names and slot values. Also, if the execution of sentences give error actions, the kind of sentences in the body of the methods can be quite different, while still giving observably equal actions.

Since there are no common derived configurations B' and D' we can not make use of an induction hypothesis with:

$$A' \leq_{D',\sigma'} B' \wedge C' \leq_{B',\rho'} D'$$

as, eg, A'||D'σ' might not be the result of the α action. Then we do not know if $A' \leq_{D',\sigma'} B'$ holds even if $A \leq_{D,\sigma} B$ holds. In order to be able to state a suitable induction hypothesis when we do induction on the length of $\bar{\delta}$ where $\bar{\delta} \in$ Traces(Aρ||Cσ), we use a more explicit version of the definition of the refinement relation. In this version, the configuration which collaborate with A is explicitly found in the relation expression, so that we, eg, can state $A' \leq_{D'', D',\sigma'} B'$ where D' and D'' stem from actions as follows:

$$A||D\sigma \xrightarrow{\alpha} A'||D'' \text{ and } B||D \xrightarrow{\bar{\beta}} B'||D'.$$

Definition: Alternative refinement relation; $A \leq_{D,E,\sigma} B$

Given two configuration D,E ∈ $\mathscr{C}$ where D.Dom = E.Dom and a substitution σ. We define a binary relation called *an alternative refinement relation*, denoted $A \leq_{D,E,\sigma} B$, as follows:

$A \leq_{D,E,\sigma} B$ ==

A||D, B||D ∈ $\mathscr{C}_{Safe}$ ∧ σ ∈ B → A ∧ Reliable(A, Dσ) ∧

$\forall\ \bar{\alpha}$: Traces(A||D) $\exists\ \bar{\beta}$: Traces(B||E) •
Reliable(A, D, $\bar{\alpha}$) ∧ $\bar{\alpha} \leq_{D',\sigma'} \bar{\beta}$ ∧ (endColab(A, D, $\bar{\alpha}$) ⇒ endColab(B, E, $\bar{\beta}$))

where σ' = prime(σ, $\bar{\alpha}$, $\bar{\beta}$, A, B, D)

Observation O.6.2: The definitions of reliable refinements are equivalent

We have the following relations between the two versions of the refinement relation:

$A \leq_{D,\sigma} B \Leftrightarrow A \leq_{D\sigma,D,\sigma} B$

This is evident from the definitions of the two relations.

Lemmas L.6.2.3 shows an other important relationships between the two definitions of the reliable refinement relation. This relationship is used in the proof of the induction step of the theorem.

Lemma L.6.2.3: Observable similarity of actions and refinement configurations

$\forall$ A, B, D, $\sigma \bullet A \leq_{D,\sigma} B$

$\Leftrightarrow$

$\exists$ A', B', D', D'', $\alpha, \overline{\beta} \bullet \alpha \in$ Traces(A||Dσ) $\wedge$ $\overline{\beta} \in$ Traces(B||D) $\wedge$ Reliable(A, Dσ) $\wedge$

$A||D\sigma \xrightarrow{\alpha} A'||D' \wedge B||D \xrightarrow{\overline{\beta}} B'||D'' \wedge \alpha \leq_{D,\sigma'} \overline{\beta} \wedge A' \leq_{D',D'',\sigma'} B'$

where σ' = prime(σ, α, $\overline{\beta}$, A, B, D)

Proof:
Proof of $\Rightarrow$:
By definition of $A \leq_{D,\sigma} B$ we have Reliable(A, Dσ) and we also have:

$\forall\ \overline{\alpha}$: Traces(A||Dσ) $\exists\ \overline{\beta}$: Traces(B||D) $\bullet\ \overline{\alpha} \leq_{O,\sigma''} \overline{\beta}$

where O = D.Dom $\cup$ NewNames($\overline{\alpha}$, D) and σ'' = prime(σ, $\overline{\alpha}$, $\overline{\beta}$, A, B, D)

We will then have:

$\forall\ \overline{\alpha}_1, \overline{\alpha}_2 \bullet \overline{\alpha}_1$ & $\overline{\alpha}_2 \in$ Traces(A||Dσ)

$\exists\ \overline{\beta}_1, \overline{\beta}_2 \bullet \overline{\beta}_1$ & $\overline{\beta}_2 \in$ Traces(B||D) $\wedge\ \overline{\alpha}_1$ & $\overline{\alpha}_2 \leq_{O,\sigma''} \overline{\beta}_1$ & $\overline{\beta}_2$

By proposition P.5.5.2 this will also hold when $\langle\alpha\rangle = \overline{\alpha}_1$ and for all $\overline{\alpha}_2 \in$ Traces(A'||D''). Then there is some $\overline{\beta}_1$ where $\alpha \leq_{D,\sigma'} \overline{\beta}_1$ and we let $\overline{\beta}_1 = \overline{\beta}$. By definition of prime substitutions we have σ'' = prime(σ', $\overline{\alpha}_2$, $\overline{\beta}_2$, A', B', D'') and we have $\overline{\alpha}_2 \leq_{O,\sigma''} \overline{\beta}_2$.

By proposition P.5.4.3 we have D'.Dom = D''.Dom. By proposition P.5.5.1 we have $\sigma' \in B' \rightarrow A' \wedge A'||D', B'||D'' \in \mathscr{C}_{Safe}$. Trivially we have

(endColab(A, Dσ, $\overline{\alpha}$) $\Rightarrow$ endColab(B, D, $\overline{\beta}$)) $\Rightarrow$ (endColab(A, D', $\overline{\alpha}_2$) $\Rightarrow$ endColab(B, D'', $\overline{\beta}_2$))

Then we have $A' \leq_{D',D'',\sigma'} B'$ and the proposition holds for this case.

Proof of $\Leftarrow$:
$A' \leq_{D',D'',\sigma'} B'$ gives Reliable(A', D'') which gives Reliable(A, D, α). $A' \leq_{D',D'',\sigma'} B'$ gives $A'||D', B'||D'' \in \mathscr{C}_{Safe} \wedge \sigma' \in B' \rightarrow A'$. Then, by definition of safe configurations and prime substitutions we also have $A||D, B||D \in \mathscr{C}_{Safe} \wedge \sigma \in B \rightarrow A$.

By definition of $A' \leq_{D',D'',\sigma'} B'$ we have:

$\forall\ \overline{\alpha}$: Traces(A'||D') $\exists\ \overline{\beta}_2$: Traces(B'||D'') $\bullet$

$\overline{\alpha} \leq_{O,\sigma''} \overline{\beta}_2 \wedge$ (endColab(A', D', $\overline{\alpha}$) $\Rightarrow$ endColab(B, D, $\overline{\beta}_2$))

where O = D'.Dom $\cup$ NewNames($\overline{\alpha}$, D') and σ'' = prime(σ, $\overline{\alpha}$, $\overline{\beta}$, A', B', D')

Then by proposition P.5.5.2 we have α & $\overline{\alpha} \leq_{O,\sigma''} \overline{\beta}$ & $\overline{\beta}_2$ for all $\alpha \in$ Traces(A||Dσ).
We trivially have

(endColab(A, D', $\overline{\alpha}$) $\Rightarrow$ endColab(B, D'', $\overline{\beta}_2$)) $\wedge\ \alpha \in$ Traces(A||Dσ) $\wedge\ \overline{\beta} \in$ Traces(A||D)

$\Rightarrow$ (endColab(A, Dσ, α & $\overline{\alpha}$) $\Rightarrow$ endColab(B, D, $\overline{\beta}$ & $\overline{\beta}_2$))

Since we have Reliable(A, Dσ) and A||D, B||D ∈ $\mathscr{C}_{\text{Safe}}$ ∧ σ ∈ B → A, then, by the way observably similar action sequences and the refinement relation are defined, we have A $\leq_{D,\sigma}$ B when the premise of the proposition holds.
❑

We can then prove the induction step as follows:

Induction step of theorem T.6.1: At least one executable sentence in Aρ||Cσ

By observation O.6.2 the simple substitution theorem can be restated for the induction step as follows:

∀ $\bar{\delta}$, A, B, C, D, σ, ρ • RelNames(A, B, C) ∧ RelNames(C, D, A) ∧

$\bar{\delta}$ ∈ Traces(Aρ||Cσ) ∧ A $\leq_{D\sigma,D,\sigma}$ B ∧ C $\leq_{B\rho,B,\rho}$ D
⇒

(Aρ||Cσ, Bρ||C ∈ $\mathscr{C}_{\text{Safe}}$ ∧ σ ∈ Bρ → Aρ ∧ Reliable(Aρ, Cσ, $\bar{\delta}$) ∧
∃ $\bar{\gamma}$: Traces(Bρ||C) • $\bar{\delta}$ $\leq_{C',\sigma'}$ $\bar{\gamma}$ ∧ (endColab(Aρ, Cσ, $\bar{\delta}$) ⇒ endColab(Bρ, C, $\bar{\gamma}$)))

where σ' = prime(σ, $\bar{\delta}$, $\bar{\gamma}$, Aρ, Bρ, C)

Assuming that $\bar{\delta}$ is one of the longest action sequences in Traces(Aρ||Cσ) and the first action in $\bar{\delta}$ is δ we have:

$A||D\sigma \xrightarrow{\bar{\alpha}} A'||D' \wedge B||D \xrightarrow{\bar{\beta}} B''||D'' \wedge A\rho||C\sigma \xrightarrow{\delta} A''||C'' \wedge B\rho||C \xrightarrow{\bar{\gamma}} B'||C'$

for some A', B', B'', C', D', D''

Let σ' = prime(σ, $\bar{\alpha}$, $\bar{\beta}$, A, B, D) and σ'' = prime(σ, δ, $\bar{\gamma}$, Aρ, Bρ, C) and ρ' = prime(ρ, $\bar{\gamma}$, $\bar{\beta}$, C, D, B)

If we can show:

(a) D'.Dom = D''.Dom ∧ B'.Dom = B''.Dom and
(b) σ' = σ'' and C'' ≡ C'σ' and A'' ≡ A'ρ' for all δ $\leq_{C,\sigma'}$ $\bar{\gamma}$ and $\bar{\alpha}$ $\leq_{D,\sigma'}$ $\bar{\beta}$ and $\bar{\gamma}$ $\leq_{B,\rho'}$ $\bar{\beta}$

then we can instatiate the instance hypothesis for some $\bar{\delta}$'' where $\#\bar{\delta}$ > $\#\bar{\delta}$'' as follows:

RelNames(A', B'', C') ∧ RelNames(C', D', B') ∧

$\bar{\delta}$'' ∈ Traces(A'ρ'||C'σ') ∧ A' $\leq_{D',D'',\sigma'}$ B'' ∧ C' $\leq_{B',B'',\rho'}$ D'
⇒

(A'ρ'||C'σ', B'ρ'||C' ∈ $\mathscr{C}_{\text{Safe}}$ ∧ σ' ∈ B'ρ' → A'ρ' ∧ Reliable(A'ρ', C'σ', $\bar{\delta}$'') ∧
∃ $\bar{\gamma}$' : Traces(B'ρ'||C') • $\bar{\delta}$'' $\leq_{C'',\sigma''}$ $\bar{\gamma}$' ∧ (endColab(A'ρ', C'σ', $\bar{\delta}$'') ⇒ endColab(B'ρ', C', $\bar{\gamma}$')))

where σ'' = prime(σ', $\bar{\delta}$'', $\bar{\gamma}$', A'ρ', B'ρ', C')

Since $\bar{\delta}$ is one of the longest action sequences in Traces(A||C) and $\bar{\delta}$ = δ & $\bar{\delta}$' then the length of all possible $\bar{\delta}$' is less than $\#\bar{\delta}$. Thus, this instance of the induction hypothesis shows A'ρ' $\leq_{C'\sigma',C',\sigma'}$ B'ρ'.

If we can show:

(c) $\exists\ \bar{\gamma}$: Traces(Bρ||C) • $\delta \leq_{C,\sigma'} \bar{\gamma}$ and

(d) $A\rho||C\sigma \in \mathscr{C}_{Safe} \wedge \sigma \in B\rho \rightarrow A\rho \wedge$ Reliable(Aρ, Cσ) and

(e) that the premise of out instance of the induction hypothesis holds

then, by the conclusion of induction hypothesis, $A'\rho' \leq_{C'\sigma',C',\sigma'} B'\rho'$, and by lemma L.6.2.3 (observable similarity of actions and refinement configurations) we can conclude:

$A\rho \leq_{C,\sigma} B\rho$

Below some properties are shown in two cases[10]:

Case 1) the action δ is from a sentence in A and

Case 2) the action δ is from a sentence in C.

The shown properties are later used to prove (a) to (e).

1) δ.exe ∈ A

Lemma L.6.2.2 (observably similar actions and common derived configurations) gives:

$\exists\ \alpha$: Traces(A||Dσ), $\bar{\gamma}$: Traces(Bρ||C), $\overline{\beta} \in$ Traces(B||D) •

(1.1) $\delta \leq_{C,\sigma'} \bar{\gamma} \wedge \delta \leq_{A,\rho'} \alpha \wedge \overline{\alpha} \leq_{D,\sigma'} \overline{\beta} \wedge \bar{\gamma} \leq_{B,\rho'} \overline{\beta} \wedge$

(1.2) prime(σ, α, $\overline{\beta}$, A, B, D) = prime(σ, δ, $\bar{\gamma}$, A, B, C) ∧

(1.3) prime(ρ, $\bar{\gamma}$, $\overline{\beta}$, C, D, B) = prime(ρ, δ, α, C, D, A) ∧

(1.4) $\exists$ A', B', C', D', B'', D'' •

$A\rho||C\sigma \xrightarrow{\delta} A'\rho'||C'\sigma' \wedge A||D\sigma \xrightarrow{\alpha} A'||D' \wedge B\rho||C \xrightarrow{\bar{\gamma}} B'||C' \wedge B||D \xrightarrow{\overline{\beta}} B''||D'' \wedge$

(1.5) D'.Dom = D''.Dom ∧ B'.Dom = B''.Dom

2) δ.exe ∈ C

Lemma L.6.2.2 (observably similar actions and common derived configurations) where A takes the role of C etc. gives:

$\exists\ \gamma$: Traces(Bρ||C), $\overline{\alpha}$: Traces(A||Dσ), $\overline{\beta}$: Traces(B||D) •

(2.1) $\delta \leq_{A,\rho'} \overline{\alpha} \wedge \delta \leq_{C,\sigma'} \gamma \wedge \gamma \leq_{B,\rho'} \overline{\beta} \wedge \overline{\alpha} \leq_{D,\sigma'} \overline{\beta} \wedge \bar{\gamma} \leq_{B,\rho'} \overline{\beta} \wedge$

(2.3) prime(ρ, γ, $\overline{\beta}$, C, D, B) = prime(ρ, δ, $\overline{\alpha}$, C, D, A) ∧

(2.2) prime(σ, $\overline{\alpha}$, $\overline{\beta}$, A, B, D) = prime(σ, δ, γ, A, B, C) ∧

(2.4) $\exists$ C', D', A', B', D'', B'' •

$A\rho||C\sigma \xrightarrow{\delta} A'\rho'||C'\sigma' \wedge B\rho||C \xrightarrow{\gamma} B'||C' \wedge A||D\sigma \xrightarrow{\overline{\alpha}} A'||D' \wedge B||D \xrightarrow{\overline{\beta}} B''||D'' \wedge$

(2.5) B'.Dom = B''.Dom ∧ D'.Dom = D''.Dom

Proof of (a):

Given by (1.5) and (2.5).

Proof of (b).

Given by (1.2), (1.4), (2.3) and (2.4)

---

[10] It may seem like the proofs of the two cases are similar, and that there is some symmetry which could make proving one of the cases unnecessary. However, by looking at the proof of lemma L.6.2.2 and by noting that part 1.1 and 2.1 in lemma L.6.2.2 have two parts, and one part is used in case 1) and the other in case 2), it should be clear that there is no such symmetry.

Two different lemmas could be created, one for each case. However, to prove one case it is necessary to also establish facts which must be proven for the other case. In other words, two lemmas, one for each case would have to prove the same things, even if the lemmas' conclusions may differ. Therefore, a single lemma is used to prove both cases, namely lemma L.6.2.2. The "unnecessary" part of the conclusion for case 1) is the last part of part 1.1, namely $\delta \leq_{A,\rho'} \alpha$ and the "unnecessary" part of the conclusion for case 2) is the first part of part 2.1, namely $\delta \leq_{A,\rho'} \overline{\alpha}$. This difference is due to the fact that A and C and their associated actions change roles in the two cases.

Proof of (c):

Let $\bar{\gamma} = \langle\gamma\rangle$ in (2.1). Then (1.1) and (2.1) shows $\delta \leq_{C,\sigma'} \bar{\gamma}$.

Proof of (d):

The premise of the theorem gives

RelNames(A, B, C) which gives $A.Dom \cap C.Dom = \emptyset$ and

$A \leq_{D\sigma,\ D,\sigma} B$ which gives $\sigma \in B \rightarrow A$ and

$C \leq_{B\rho,\ B,\rho} D$ which gives $\rho \in D \rightarrow C$ and $B.Dom \cap C.Dom = \emptyset$

Then by proposition P.6.1 (reliable refinements ensure sage names and reliable substitutions for specialised refinements) we have $A\rho \| C\sigma \in \mathscr{C}_{Safe} \wedge \sigma \in B\rho \rightarrow A\rho$.

The premise of the theorem gives $A \leq_{D\sigma,\ D,\sigma} B \wedge C \leq_{B\rho,\ B,\rho} D$ which by observation 6.2 gives $A \leq_{D,\sigma} B \wedge C \leq_{B,\rho} D$. By this and since we also have RelNames(A, B, C) and RelNames(C, D, A) the proposition P.6.6 (specialised reliable refinements are reliable configurations) gives Reliable($A\rho$, $C\sigma$).

Proof of (e):

RelNames(A, B, C) gives $A \| C \in \mathscr{C}_{Safe} \wedge C.Values \cap A.Dom \subseteq C.Values \cap B.Dom$.

The rules of action give $A' \| C' \in \mathscr{C}_{Safe}$. By the rules of action, and since the actions are observably similar, any new values in C' compared with C will be B'-names. We then have:

$C'.Values \cap A'.Dom \subseteq C'.Values \cap B'.Dom$

and then we have RelNames(A', B', C'). Similarly we have RelNames(C', D', B').

By lemma L.6.2.3 (observable similarity of actions and refinement configurations) we have $A' \leq_{D',\ D'',\sigma'} B' \wedge C' \leq_{B',\ B'',\rho'} D'$.

We have then shown the induction base and the induction step for all cases and the theorem holds.
❑

6.3 Replacing Configurations

The refinement relation is defined so that a refinement and its specification have similar observable behaviour relative to a context of observing objects. It was defined for a system consisting of two components. In the general case we may have more than two components. A nice property would then be that an observing context of objects would observe similar behaviour when an arbitrary number of its collaborators were substituted with reliable refinements.

For example there might be a system with three components, eg, B||D||F. Consider two configurations A and C which are reliable refinements of B and D, respectively, ie, we have

$$A \leq_{D||F,\sigma} B \wedge C \leq_{B||F,\rho} D$$

where the substitutions are reliable substitutions. It would then be nice to be able to conclude that the combination of the refinements, ie, $A\rho||C\sigma$ is a refinement of the combination of specifications B||D relative to the last specification component F, ie, $A\rho||C\sigma \leq_{F,\sigma\rho} B||D$. This property is shown in the next lemma. A general version of this lemma, where a system may have any number of components, is shown in theorem T.6.2.

Lemma L.6.3.1 Refinements are observably similar to parts of the observing configuration

$$\forall\ A, B, C, D, F, \sigma, \rho \bullet$$

$$\text{RelNames}(A, B, C) \wedge \text{RelNames}(C, D, A) \wedge$$
$$A \leq_{D||F,\sigma} B \wedge C \leq_{B||F,\rho} D \Rightarrow A\rho||C\sigma \leq_{F,\sigma\rho} B||D$$

Proof:
By definition of the refinement relation the conclusion can be restated

$$A\rho||C\sigma||F, B||D||F \in \mathscr{C}_{\text{Safe}} \wedge \sigma\rho \in B||D \rightarrow A||C \wedge$$
$$\text{Reliable}(A\rho||C\sigma, F) \wedge \text{Reliable}(A\rho||C\sigma, F\sigma\rho, \bar{\delta}) \wedge$$
$$\forall\ \bar{\delta} : \text{Traces}(A\rho||C\sigma||F\sigma\rho)\ \exists\ \bar{\beta} : \text{Traces}(B||D||F) \bullet$$
$$\bar{\delta} \leq_{F,\sigma\rho} \bar{\beta} \wedge (\text{endColab}(A\rho||C\sigma, F\sigma\rho, \bar{\delta}) \Rightarrow \text{endColab}(B||D, F, \bar{\beta}))$$

By the premise of the lemma and the definition of the refinement relation we have:
$A||D||F \in \mathscr{C}_{\text{Safe}}$ which gives $A.\text{Dom} \cap D.\text{Dom} \cap F.\text{Dom} = \emptyset$ and $SN(A||D||F) \cap ON(A||D||F) = \emptyset$
$C||B||F \in \mathscr{C}_{\text{Safe}}$ which gives $C.\text{Dom} \cap B.\text{Dom} \cap F.\text{Dom} = \emptyset$ and $SN(C||B||F) \cap ON(C||B||F) = \emptyset$
RelNames(A, B, C) gives $A||C \in \mathscr{C}_{\text{Safe}}$ which gives:
$A.\text{Dom} \cap C.\text{Dom} = \emptyset$ and $SN(A||C) \cap ON(A||C) = \emptyset$
We then have
$A.\text{Dom} \cap C.\text{Dom} \cap F.\text{Dom} = \emptyset$ and $SN(A||C||F) \cap ON(A||C||F) = \emptyset$
which gives $A||C||F \in \mathscr{C}_{\text{Safe}}$. By the premise of the lemma and the definition of the refinement relation we also have:
$\sigma \in B \rightarrow A$ and $\rho \in D \rightarrow C$
Since the configurations have safe names, ie, they have non-overlapping domains, then we have
$\sigma\rho \in B||D \rightarrow A||C$ and $A\rho||C\sigma||F \in \mathscr{C}_{\text{Safe}}$

$B||D||F \in \mathscr{C}_{\text{Safe}}$ is given by $A \leq_{D||F,\sigma} B$.

The premise of the lemma and the definition of the refinement relation also give:
Reliable(A, $D\sigma||F\sigma$) and Reliable(C, $B\rho||F\rho$).
This gives:
$\text{noExt}(A) \wedge \text{RelMethodLookup}(A, D\sigma||F\sigma) \wedge \text{RelIfSentence}(A, D\sigma||F\sigma) \wedge \text{RelMessageSend}(A, D\sigma||F\sigma)$
$\text{noExt}(C) \wedge \text{RelMethodLookup}(C, B\rho||F\rho) \wedge \text{RelIfSentence}(C, B\rho||F\rho) \wedge \text{RelMessageSend}(C, B\rho||F\rho)$

We then have:
noExt(A||C) and by proposition P.5.3.1 (reliable substitutions give same slots and preserve "No external inheritance") we have noExt($A\rho||C\sigma$).
RelIfSentence($A\rho||C\sigma, F\sigma\rho$) by definition of RelIfSentence and reliable configurations and substitutions.
RelMessageSend($A\rho||C\sigma, F\sigma\rho$) by definition of RelMessageSend and reliable configurations and substitutions.

RelMethodLookup(Aρ||Cσ, Fσρ) by definition of RelMessageSend and reliable configurations and substitutions.

Since the configurations have non-overlapping domains and $\sigma\rho \in B||D \rightarrow A||C$ and $A\rho||C\sigma||F \in \mathscr{C}_{Safe}$, this gives Reliable(Aρ||Cσ, F).
The same arguments can be given for all derivations of Aρ||Cσ||Fσρ and we then have
Reliable(Aρ||Cσ, Fσρ, $\bar{\delta}$).

The premise of the lemma gives:

$\forall\ \bar{\alpha}$: Traces(A||Dσ||Fσ) $\exists\ \bar{\beta}$: Traces(B||D||F) •
$\bar{\alpha} \leq_{D||F,\sigma} \bar{\beta} \wedge$ (endColab(A, Dσ||Fσ, $\bar{\alpha}$) ⇒ endColab(B, D||F, $\bar{\beta}$)) and
$\forall\ \bar{\gamma}$: Traces(C||Bρ||Fρ) $\exists\ \bar{\beta}$: Traces(B||D||F) •
$\bar{\gamma} \leq_{B||F,\rho} \bar{\beta} \wedge$ (endColab(C, Bρ||Fρ, $\bar{\alpha}$) ⇒ endColab(D, B||F, $\bar{\beta}$))

For $A\rho||C\sigma \leq_{F,\sigma\rho} B||D$ to hold we must show:

$\forall\ \bar{\delta}$: Traces(Aρ||Cσ||Fσρ) $\exists\ \bar{\beta}$: Traces(B||D||F) •
$\bar{\delta} \leq_{F,\sigma\rho} \bar{\beta} \wedge$ (endColab(Aρ||Cσ, Fσρ, $\bar{\delta}$) ⇒ endColab(B||D, F, $\bar{\beta}$))

Next we prove this by induction on the length of $\bar{\delta}$.

Induction base: $\#\bar{\delta} = 0$
The lemma holds for this case if we can show:
endColab(Aρ||Cσ, Fσρ) ⇒ endColab(B||D, F) when $A \leq_{D||F,\sigma} B \wedge C \leq_{B||F,\rho} D$.

When endColab(Aρ||Cσ, Fσρ) we have by definition:

$\forall\ \bar{\delta}'$: Traces(Aρ||Cσ||Fσρ) • $\bar{\delta}'$.exe ∈ Aσ||Cσ ⇒ $\bar{\delta}' \otimes$ F

First we show that we have endColab(A, Dσ||Fσ) by induction on the length of the longest $\bar{\delta}'$.

Base : $\#\bar{\delta}' = 0$
We the longest action sequence from Aρ||Cσ is the empty action sequence, then the longest action sequence from A will also be empty. We then have endColab(A, Dσ||Fσ).

Step: $\#\bar{\delta}' = m$
Induction hypothesis: we have endColab(A, Dσ||Fσ) when the length of the longest $\bar{\delta}'$ is shorter than m.
Assume $\bar{\delta}' = \delta$ & $\bar{\delta}''$. We have δ.exe ∈ A, $\delta \otimes C\sigma||F\sigma\rho$ and endColab(A'ρ, Cσ||Fσρ), where A' is A after δ, from definition of endColab(). The induction hypothesis then gives endColab(A', Dσ||Fσ). By proposition P.6.4 we have some α from A in A||D||F where $\delta \equiv \alpha\rho$. By observation O.5.4.4 (observability of observably similar actions) we then have $\alpha \otimes D||F$. This gives

endColab(A, Dσ||Fσ).

Similarly, because of symmetry, we can show that endColab(Aρ||Cσ, Fσρ) also gives

endColab(C, Bρ||Fρ).

We have then shown endColab(A, Dσ||Fσ) and endColab(C, Bρ||Fρ) for any length of the longest $\bar{\delta}'$. From $A \leq_{D||F,\sigma} B$ we then have endColab(B, D||F) which gives:

$\forall\ \bar{\beta}$: Traces(B||D||F) • $\bar{\beta}$.exe ∈ B ⇒ $\bar{\beta} \otimes$ D||F

and from $C \leq_{B||F,\rho} D$ we have endColab(D, B||F) which gives:

$\forall\ \bar{\beta}$: Traces(B||D||F) • $\bar{\beta}$.exe ∈ D ⇒ $\bar{\beta} \otimes$ B||F

We then have $\forall\ \bar{\beta}$: Traces(B||D||F) • $\bar{\beta}$.exe ∈ B||D ⇒ $\bar{\beta} \otimes$ F

By definition we then have endColab(B||D, F) which shows that the induction base holds.

Induction step: $\#\bar{\delta}$ = n+1:

We consider the different cases for where the first action in $\bar{\delta}$ comes from. Let δ be the first action in $\bar{\delta}$. Because of symmetry between A and C it is only necessary to make the proof for the two cases δ.exe ∈ A and δ.exe ∈ F. If we can show

$\exists <\beta_1,..,\beta_n> : Traces(B||D||F) \bullet \delta \leq_{F,\sigma'\rho'} <\beta_1,..,\beta_n>$

where σ' = prime(σ, δ, $<\beta_1,..,\beta_n>$, A, B, C||F) and ρ' = prime(ρ, δ, $<\beta_1,..,\beta_n>$, C, D, A||F)

then the induction hypothesis gives that we have $\bar{\delta}' \leq_{F',\sigma\rho} \bar{\beta}'$ where $\bar{\delta}'$ is the rest of the action sequence $\bar{\delta}$ after the first action and $\bar{\beta}'$ is the rest of $\bar{\beta}$ after the initial actions $<\beta_1,..,\beta_n>$ and F' is the derived configuration after δ.

Case 1) δ.exe ∈ A

Since δ.exe ∈ A, then proposition P.6.4 (reliability gives equal actions relative to a reliable substitution) gives ∃ α : action(Aρ||Cσ||Fσρ) • δ ≡ αρ'' where ρ'' = prime(ρ, δ, α, C, D, A||F). Then

$\exists <\beta_1,..,\beta_n> : Traces(B||D||F) \bullet \alpha \leq_{D||F,\sigma'} <\beta_1,..,\beta_n>$ where β_i.exe ∈ B.

This gives $<\beta_1,..,\beta_{n-1}> \otimes D||F$ and $\alpha \leq_{D||F,\sigma'} \beta_n$. Cases by observability of α gives three cases: not observable from D||F, observable from D, and observable from F:

Case α ⊗ F.

Then $\beta_n \otimes F$ and also δ ⊗ F. This gives $\delta \leq_{F,\sigma'\rho'} <\beta_1,..,\beta_n>$ and $\bar{\delta} \leq_{F,\sigma\rho} \bar{\beta}$ holds for this case.

Case α ∈ obs(F)

Then β_n ∈ obs(F) and also δ ∈ obs(F). We then have the following cases for the different kinds of actions for observably similar α and β_n actions and possible F-observable actions from execution of a sentence in A:

$(\alpha \equiv e->o!x(q_1,\ldots,q_m)/k \wedge \beta_{n-1} \equiv f->o!x(p_1,\ldots,p_m)/k \wedge \forall\ 1 \leq i \leq m \bullet q_i = p_i\sigma')$ or

$(\alpha \equiv e->(i.s:=k/o) \wedge \beta_{n-1} \equiv f->(j.t:=k/o) \wedge i,j \notin F \wedge o \in F)$

Since we have δ ≡ αρ'' the action δ for these cases will be:

$\delta \equiv e->o!x(r_1,\ldots,r_m)/k \wedge \forall\ 1 \leq i \leq m \bullet r_i = q_i\rho''$

$\delta \equiv e->(i.s:=k/o\rho')$

This gives $\delta \leq_{F,\sigma'\rho'} \beta_n$ which gives $\delta \leq_{F,\sigma'\rho'} <\beta_1,..,\beta_n>$ and $\bar{\delta} \leq_{F,\sigma\rho} \bar{\beta}$ holds for this case.

Case 2) δ.exe ∈ F

We must then show:

$\exists <\beta> : Traces(B||D||F) \bullet \delta \equiv \beta\sigma'\rho'$ where σ' = prime(σ, δ, β, A, B, F) and ρ' = prime(ρ, δ, β, C, D, F)

By definition of observably similar action sequences and observably similar actions we have that the sentence which gave α in A||Dσ||Fσ will allow an action β in B||D||F such that

$\alpha \equiv \beta\sigma'$ where σ' = prime(σ, α, β, A, B, D||F)

Also, the sentence will give an action γ in C||Bρ||Fρ such that

$\gamma \equiv \beta\rho'$ where ρ' = prime(σ, α, β, C, D, B||F)

For each slot name in the executed sentence in F which gave δ we show that we have:

* (slot not found in Aρ||Cσ||Fσρ ⇒ slot not found in B||D||F) and

** (slot found in Aρ||Cσ||Fσρ and assume the value of the slot is i ⇒
a slot is found in B||D||F and the value is iσ'ρ')

Then we have that in the *-case both actions are error actions and in case ** we have δ ≡ βσ'ρ'.

Case slot not found:

Cases by where the slot in the executed sentence in F in B||D||F giving β is found. Because of symmetry we have the following cases:

slot found in F or

slot found in B

Also because of symmetry we only consider cases with inheritance through B when the slot is found in F in the cases below.

Case slot found in F without external inheritance:

Can not hold since we then would also find the slot in Aρ||Cσ||Fσρ.

Case slot found in F with external inheritance through B:

Can not have inheritance through A since noExt(A). Then no slot will be found in A||Dσ||Fσ and we will not have α ≡ βσ'. This case will therefore never occur.

Case slot found in B:

Can not find the slot in A since no slot was found in Aρ||Cσ||Fσρ. Then no slot will be found in A||Dσ||Fσ and we will not have α ≡ βσ'. This case will therefore never occur.

We then have $\delta \equiv$ f->error and $\beta \equiv$ f->error which gives $\delta \equiv \beta\sigma'\rho'$ for this case.

Case slot found:
We denote the value from the different actions by i and a subscript naming the action where the value is found. For example i_δ denote the i-value in the action δ.
Cases by where the slot in the executed sentence in F in $A\rho||C\sigma||F\sigma\rho$ giving δ is found. Because of symmetry we have the following cases:
slot found in F or
slot found in $A\rho$
Also because of symmetry we only consider cases with inheritance through $A\rho$ when the slot is found in F in the cases below.
Case slot is found in F :
- no external inheritance: then
$i_\delta \in A \Rightarrow i_\delta = i_\beta\sigma'$
$i_\delta \in C \Rightarrow i_\delta = i_\beta\rho'$
$i_\delta \notin A||C \Rightarrow i_\delta = i_\beta$
this gives $\delta \equiv \beta\sigma'\rho'$
- inheritance through A :
Can not have inheritance through A since noExt(A). Thus this case will never occur.

Case slot found in A :
- directly found in A: we then have $i_\delta = i_\alpha\rho'$ and :
$i_\delta \in A \Rightarrow i_\alpha = i_\beta\sigma'$ which gives $i_\delta = i_\beta\sigma'$
$i_\delta \in C \Rightarrow i_\alpha = i_\beta \in D$ and $i_\gamma = i_\beta\rho' \in C$. This gives $i_\delta = i_\beta\rho'$
$i_\delta \notin A||C \Rightarrow i_\delta = i_\alpha = i_\gamma = i_\beta$. This gives $i_\delta = i_\beta\rho'$
We then have $\delta \equiv \beta\sigma'\rho'$ for this case.

- inheritance through C:
Can not have inheritance through C since noExt(C). Thus this case will never occur.

We then have $\delta \equiv \beta\sigma'\rho'$ for all possible cases which gives $\delta \leq_{F,\sigma'\rho'} \beta$ and $\bar{\delta} \leq_{F,\sigma\rho} \bar{\beta}$ holds for this case.

If we have endColab($A\rho||C\sigma$, $F\sigma\rho$, $\bar{\delta}$) and $\bar{\delta} \leq_{F,\sigma\rho} \bar{\beta}$ then because of safe names, reliable names and reliable substitutions and all actions in $\bar{\delta}$ which are hidden to $F\sigma\rho$ are from sentences in reliable configurations, we also have endColab(B||D, F, $\bar{\beta}$) and the lemma holds.
❑

In the above case the system had three components. In the general case there may be more than three components, but we still want the same property. This is shown in the next theorem. The theorem is formulated for a system $D \equiv D_1||..||D_n$, which plays the role of a specification and n configurations $C_1..C_n$ where configuration C_i is a refinement of the specification component D_i. For each specification component there is a corresponding substitution, denoted σ_i, which substitute from D_i names to C_i names.

Theorem T.6.2 The component combination theorem

$\forall\ D_1,..,D_n,C_1,..,\ C_n : \mathscr{C},\ \sigma_1..\sigma_n : \mathscr{Su} \bullet$

$(\forall\ i \in \{1..n\} \bullet \text{RelNames}(C_i, D_i, C\text{-}i) \wedge C_i \leq_{D-i,\sigma_i} D_i)$

$\Rightarrow \forall\ i \in \{1..n\} \bullet C\text{-}i\ \sigma_i \leq_{D_i,\sigma-i} D\text{-}i$

where σ is the substitutions $\sigma_1..\sigma_n$
$D \equiv D_1||..||D_n$ and $C \equiv C_1||..||C_n$
D-i denote all D_j except D_i and
σ-i denote all σ except σ_i and
$C\sigma$-i denote $C_1\sigma_1 \parallel ... \parallel C_{i-1}\sigma_{i-1} \parallel C_{i+1}\sigma_{i+1} \parallel \ldots \parallel C_n\sigma_n$

Proof:
To show the theorem we show that C-i σi $\leq_{D_i,\sigma-i}$ D-i holds for any i.

By applying L.6.3.1 (refinements are observably similar to parts of the observing configuration) letting A ≡ C_1 and C ≡ C_2 and F ≡ D_3,..,D_n, the lemma concludes (to simplify the expressions below, the substitutions are not included):

$$C_1||C_2 \leq_{D-\{1,2\}} D_1||D_2$$

From the premise of the proposition we have $\forall$ i ∈ {3..n} • $C_i \leq_{D-i,\sigma_i} D_i$. We can then apply L.6.3.1 once more and let A ≡ C_1C_2, C ≡ C_3 and F ≡ D_4,..,D_n and get:

$$C_1||C_2||C_3 \leq_{D-\{1,2,3\}} D_1||D_2||D_3$$

In this way we can re-apply L.6.3.1 until we have:

$$C_1||..||C_{n-1} \leq_{D_n} D_1||...||D_{n-1}$$

Similarly we can show this for any configuration D_j where j ∈ {1…n} since || by definition is associative, and commutative and therefore the theorem holds.
❑

Observation O.6.3 : Components can be combined arbitrarily and refinement is preserved

By the proof of theorem T.6.2 it is obvious that it can be shown that if we have:

$$\forall i \in \{1..n\} \bullet \text{RelNames}(C_i, D_i, C\text{-}i) \wedge C_i \leq_{D-i,\sigma_i} D_i$$

then for any combinations of D-configurations, here denoted D_X, we have

$$C\text{-}x \leq_{D_x,\sigma-x} D\text{-}x$$

where D_X is some combination $D_i||D_j||..||D_k||D_l$ for i ≠ j ≠ k ≠ l etc. and D-x is all D except i,j,..,k,l and similar for C-x.

From this we can conclude that when we replace any number of components with their reliable refinements, the objects in the other configurations will not observe any difference in behaviours.

6.4 The General Substitutability Theorem

The following theorem shows the general substitution proposition of chapter 1 for the reliable refinement relation.

In theorem T.6.2 the observers in the conclusion were specification components, ie,

D_i is the observer in $\forall\, i \in \{1..n\} \bullet C\text{-}i\ \sigma i \leq_{D_i,\sigma-i} D\text{-}i$.

Observation O.6.3 concluded from T.6.2 that any number of specification components may be substituted by their reliable refinements while the remaining *specification components do not observe any difference*. In the substitution proposition of chapter 1, the conclusion said that *reliable refinements will not observe any difference* when any number of specification components are substituted with their reliable refinements. In this last case it is the refinements which are the observers in the conclusion of the proposition. Theorem T.6.3 is a reformulation of the substitution proposition using the reliable refinement relation and the difference from T.6.2 is that it has refinement components as observers in the conclusion, ie,

C_{-i} are the observers in $\forall\, i \in \{1..n\} \bullet C_i\ \sigma\text{-}i \leq_{C-i,\sigma_i} D_i\ \sigma\text{-}i$

Theorem T.6.3 The general substitutability theorem

$$\forall\ D_1,..,D_n,C_1,..,C_n : \mathscr{C},\ \sigma_1..\sigma_n : \mathscr{Su} \bullet$$

$$\forall\, i \in \{1..n\} \bullet \text{RelNames}(C_i, D_i, C\text{-}i) \wedge C_i \leq_{D-i,\sigma_i} D_i$$

$$\Rightarrow \forall\, i \in \{1..n\} \bullet C_i\ \sigma\text{-}i \leq_{C-i,\sigma_i} D_i\ \sigma\text{-}i$$

where σ is the combined substitution $\sigma_1..\sigma_n$
$D \equiv D_1||..||D_n$ and $C \equiv C_1||..||C_n$
D-i denote all D_j except D_i and
σ-i denote all σ except σ_i and
$C\sigma$-i denote $C_1\sigma_1 \parallel ... \parallel C_{i-1}\sigma_{i-1} \parallel C_{i+1}\sigma_{i+1} \parallel \ldots \parallel C_n\sigma_n$

Proof:
First we show that the substitutions can be applied in any order without changing the result. From RelNames(C_i, D_i, C-i) we have $\forall\, i \in \{1..n\} \bullet C_i.Dom \cap C\text{-}i.Dom = \emptyset$. This gives:

$\forall\, i \in \{1..n\} \bullet C\text{-}i.Dom \cap D\text{-}i.Dom = \emptyset$

From $\forall\, i \in \{1..n\} \bullet C_i \leq_{D-i,\sigma_i} D_i$ we have $\forall\, i \in \{1..n\} \bullet C_i.Dom \cap D\text{-}i.Dom = \emptyset \wedge D_i.Dom \cap D\text{-}i.Dom = \emptyset$.
Since $\sigma_i \in D_i \rightarrow C_i$ and the object names in the configurations do not overlap, the substitutions can be combined and the order of application can be changed without changing the result of applying a given substitution to a name in the configuration.

The rest of the proof is done by induction on the number of components in the system, that is induction over n.

Induction Basis: There are two components in the system, ie, n=2. This is shown in theorem T.6.1.

Induction step: The system has n components where n > 2.
The induction hypothesis gives that the theorem holds for n-1 components.

Let the n-1'th and n'th component of a system with n components together form a component. We will then have a system with only n-1 component. An example situation is shown in figure F.6.1. Here we have a system which originally had four components (n = 4), but where the 3rd and 4th components are combined. This is illustrated by the thick outline around component 3 and 4. The thickest arrows in the figure show which collaborations are proven to be observably similar by the induction hypothesis when the third and fourth components are combined into one component.

The general substitution theorem for n components where the configurations n-1 and n are combined is stated:

$\forall$ i $\in$ {1..n-2} • RelNames(C_i, D_i, C-i) $\wedge$ RelNames(C_i, D_i, $C_{n-1} \parallel C_n$,) $\wedge$
$C_i \leq_{D-i,\sigma_i} D_i \wedge C_{n-1} \parallel C_n \leq_{D-\{n-1,n\},\sigma_{n-1}\sigma_n} D_{n-1} \parallel D_n$
$\Rightarrow$
$\forall$ i $\in$ {1..n-2} • C_i σ–i $\leq_{C-i,\sigma_i} D_i \wedge$
$C_{n-1} \parallel C_n$ σ–{n-1,n} $\leq_{C-\{n-1,n\},\sigma_{n-1}\sigma_n} D_{n-1} \parallel D_n$ σ–{n-1,n}

When two components are combined, as expressed above, the induction hypothesis gives that the conclusions hold, provided we can show the premise.

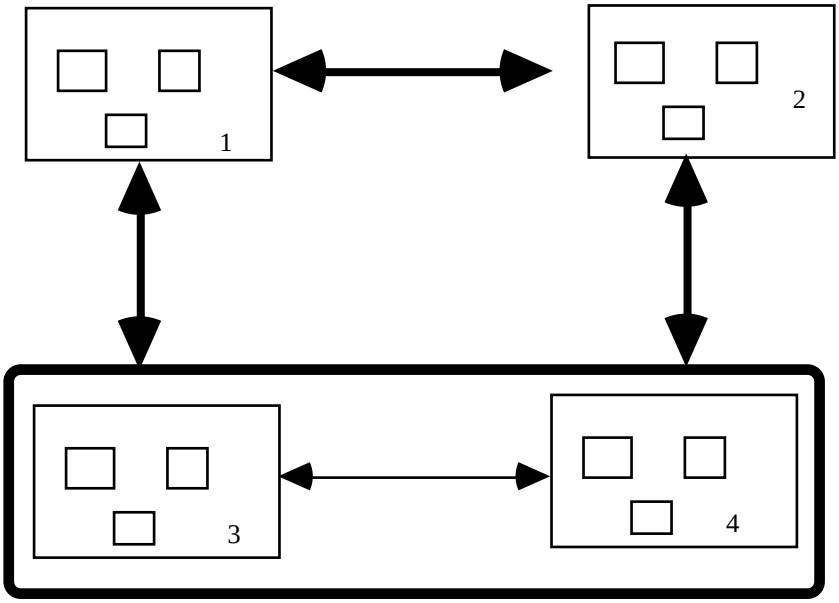


Figure F.6.1: Illustration of what is shown by the induction hypothesis when n-1=3 and n = 4. The 3'rd and 4th components form together a component. This is illustrated by the box with the thick border. The thickest arrows show which collaborations are assumed to be observably similar by the induction hypothesis.

The premise holds because:

The definition of RelNames() ensures that all objects in the two configurations have unique names.
Then $\forall$ i $\in$ {1..n} • RelNames(C_i, D_i, C-i) gives

$\forall$ i $\in$ {1..n} • $C_i \| C\text{-}i \in \mathscr{C}_{Safe} \wedge C_i.Values \cap C\text{-}i.Dom \subseteq D_i.Values \cap C\text{-}i.Values$

This gives in particular:
$\forall$ i $\in$ {1..n-2} •
$C_i \| C_{n-1}$, $C_i \| C_n \in \mathscr{C}_{Safe} \wedge$
$C_i.Values \cap C_{n-1}.Dom \subseteq D_i.Values \cap C_{n-1}.Values \wedge$
$C_i.Values \cap C_n.Dom \subseteq D_i.Values \cap C_n.Values$
which gives
$\forall$ i $\in$ {1..n-2} • RelNames(C_i, D_i, C-i) $\wedge$ RelNames(C_i, D_i, $C_{n-1} \parallel C_n$)

Next we have that $\forall$ i $\in$ {1..n} • $C_i \leq_{D-i,\sigma_i} D_i$ gives

$\forall$ i $\in$ {1..n-2} • $C_i \leq_{D-i,\sigma_i} D_i$.

We then need to show $C_{n-1} \parallel C_n \leq_{D-\{n-1,n\},\sigma_{n-1}\sigma_n} D_{n-1} \parallel D_n$ from $\forall$ i $\in$ {1..n} • $C_i \leq_{D-i,\sigma_i} D_i$. The premise of the theorem and lemma L.6.3.1 (refinements are observably similar to parts of the observing configuration) where A ≡ C_{n-1}, C ≡ C_n and F ≡ D-{n-1,n}, gives:

$C_{n-1} \leq_{D-(n-1),\sigma_{n-1}} D_{n-1} \wedge C_n \leq_{D-n,\sigma_n} D_n \Rightarrow C_{n-1} \parallel C_n \leq_{D-\{n-1,n\},\sigma_{n-1}\sigma_n} D_{n-1} \parallel D_n$

and then the premise holds. The conclusion then gives $\forall i \in \{1..n\text{-}2\} \bullet C_i\, \sigma\text{–}i \leq_{C-i,\sigma_i} D_i$.

For the general substitution proposition to hold for n components, it must be shown that the n-1'st and n'th configurations are refinements too. This can be illustrated as in figure F.6.2 where the thickest arrows show which collaborations have to be proved observably similar for configuration 4. Similarly, the corresponding collaborations have to be proved observably similar for configuration 3. This is expressed as

$$C_3 \leq_{C\text{-}3} D_3 \wedge C_4 \leq_{C\text{-}4} D_4$$

To prove the theorem we must then also show:

$$C_{n-1}\sigma\text{-}(n\text{-}1) \leq_{C-(n-1),\sigma_{n-1}} D_{n-1}\sigma\text{-}(n\text{-}1) \wedge C_n\sigma\text{-}n \leq_{C_n,\sigma-n} D_n\sigma\text{-}n$$

Because of symmetry between components numbered n-1 and n it is only necessary to prove observably similarity for one of the configurations. The premise of the theorem gives $C_n \leq_{D-n,\sigma_n} D_n$. By the premise of the theorem and theorem T.6.2 we have $C\text{-}n \leq_{D_n,\sigma-n} D\text{-}n$. We can then use theorem T.6.1 letting

$$B \equiv D_n \qquad A \equiv C_n \qquad D \equiv D\text{-}n \qquad C \equiv C\text{-}n$$

and conclude

$$C_n \leq_{D-n,\sigma_n} D_n \wedge C\text{-}n \leq_{D_n,\sigma-n} D\text{-}n$$
$$\Rightarrow C_n\sigma\text{-}n \leq_{C-n,\sigma_n} D_n\sigma\text{-}n \wedge C\text{-}n\ \sigma_n \leq_{C_n,\sigma-n} D\text{-}n\ \sigma_n$$

This gives $C_n\sigma\text{-}n \leq_{C_n,\sigma-n} D_n\sigma\text{-}n$ and then theorem holds.

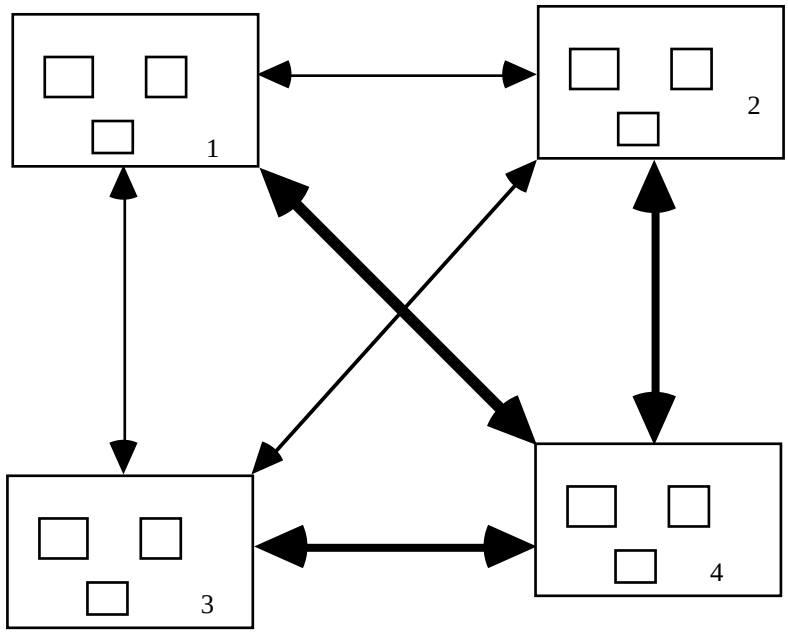


Figure F.6.2: The thickest arrows show which collaborations have to be proven observably similar for configuration 3 in order to prove the theorem for four components, given the induction hypothesis illustrated in F.6.1.

❑

We have now shown the general substitution proposition and thereby shown that the reliability requirements from chapter 5 are sufficient to ensure reliable substitution. Chapter 7 shows that similar reliability requirements are sufficient to show the substitution proposition for sequential object component systems.

6.5 Reliable Substitution

When we have reliable substitution, then any number of components can be substituted with their reliable refinements while all other components will observe the same behaviour of the system. Let D be divided into three sets of components, denoted Dx, Dy and Dz. Similarly we have Cx, Cy and Cz where the indexes of, eg, Dx and Cx are equal so that $C_i \in Cx \Leftrightarrow D_i \in Dx$. We can then express reliable substitution in an even more general form then the general substitution proposition as follows:

Theorem T.6.4 The reliable substitution theorem

$$\forall\ D_1,..,D_n,C_1,..,C_n : \mathscr{C},\ \sigma_1..\sigma_n : \mathscr{Su} \bullet$$

$$\forall\ i \in \{1..n\} \bullet \text{RelNames}(C_i, D_i, C\text{-}i) \wedge C_i \leq_{D-i,\sigma_i} D_i \Rightarrow Cx\ \sigma y \leq_{(CyDz)\sigma y,\sigma x} Dx\ \sigma y$$

Where
$D_1,..,D_n$ is divided into three sets, denoted Dx, Dy and Dz and
similarly $C_1,..,C_n$ is divided into Cx, Cy and Cz where the indexes of, eg, Dx and Cx are equal so that
$C_i \in Cx \Leftrightarrow D_i \in Dx$.
σy is a set of substitutions, one σ_i for each C_i in CyDz
A substitution σ_i in the set σy is applied to all components C_j and D_k where $j \neq i$ and $k \neq i$.
σy is a set of substitutions, one σ_j for each C_j not in Cy.

Proof:
From observation O.6.3 (components can be combined arbitrarily and refinement is preserved) and the proof of theorem T.6.3 we see that the theorem holds.
❑

The theorem says that if Dx, ie, some arbitrary number of components, are substituted with their reliable refinements Cx, then the other components, CyDz, will not observe any difference. The other components may be both "old" components, here denoted Dz, or reliable refinements of old components, here denoted Cy.

This shows that the reliability requirements and the refinement relation with specialisation gives reliable substitution. Chapter 8 shows some examples of how the properties which give reliable substitution can benefit system development practices. Chapter 9 shows how reliable substitution is linked to composition and decomposition properties of assumption/guarantee specifications.

6.6 A Library of Objects

The idea behind the substitution proposition was that each component could be substituted with a reliable refinement while all other components will observe the same behaviour of the rest of the system. The price to pay for this was that no object in one component could inherit from an object in another component or find a method in another component. This means that a component can not be what is traditionally seen as a program library. Typically an object will inherit shared data from a library and find methods there. For example, a common library L is used both by a refinement of B and a refinement of D, ie, we would like to be able to show:

The library proposition:

$$A||L \leq_{D,\sigma} B \wedge C||L \leq_{B,\rho} D \Rightarrow A\rho||L \leq_{C,\sigma} B\rho \wedge C\sigma||L \leq_{A,\rho} D\sigma$$

when A and C can inherit and find methods in L, ie, the reliability requirements NoExt(A), NoExt(C), RelMethodLookup(A, L) and RelMethodLookup(C, L) do not hold.

The central proposition in showing the refinement relation, which is based in the reliability requirements, is proposition P.6.4 "Reliability gives equal actions relative to a reliable substitution". This proposition shows:

$$\forall\ A, B, C, D, L, \rho, \sigma, \delta \bullet$$
$$\text{RelNames}(A||L, B, C) \wedge \text{RelNames}(C||L, D, A) \wedge A||L \leq_{D,\sigma} B \wedge C||L \leq_{B,\rho} D \wedge$$

$$\delta \in \text{Traces}(A\rho||L||C\sigma) \wedge \delta.\text{exe} \in A||L \Rightarrow \exists\ \alpha : \text{Traces}(A||L||D\sigma) \bullet \delta \equiv \alpha\rho'$$

where $\rho' = \rho + \{k/l\}$ when δ has the form a->i.s := k/oρ,
α has the form a->i.s := l/o where $o \in D$ and
$\rho' = \rho$ in all other cases

In this case all the reliability requirements have to hold for L, ie, we require:

NoExt(L, B||D), RelMethodLookup(L, B||D), RelMessageSend(L, B||D) and RelIfSentences(L, B||D)

Another central proposition is P.6.7 "Observably similar actions give a common derived configuration". This shows an important property in order to be able to do the inductive proofs of the above theorems. However, from the premise of the library proposition we do not know if L gets observably similar actions from, eg, C and D. We would need to know that L gets observably similar actions to show that we have a common L' after actions from Aρ||L||C and Aρ||L||D, ie:

$$\delta \in \text{Traces}(A\rho||L||C) \wedge \alpha \in \text{Traces}(A||L||D) \wedge \delta \leq_{A,\rho'} \alpha$$
$$\Rightarrow$$
$$\exists\ A' \bullet A||L||D\sigma \xrightarrow{\alpha} A'||L'||D' \wedge A\rho||L||C \xrightarrow{\delta} A'\rho'||L'||C'$$

There are two main alternatives which allow us to show that we have a common L'. Either we restate the library proposition as follows, ie, require all L observed actions to be observably similar in the premise of the proposition:

$$A \leq_{D||L,\sigma} B \wedge C \leq_{B||L,\rho} D \Rightarrow A\rho||L \leq_{C,\sigma} B\rho \wedge C\sigma||L \leq_{A,\rho} D\sigma$$

or

we require that there are no L-observable actions in any of the systems. Then L ≡ L' for all actions.

This latter alternative is perhaps the most common way to treat program libraries, typically sets of classes. This is further discussed in chapter 8.

If it is assumed that no actions are observed by L, it is not necessary to require RelMethodLookup(L, B||D). However, when A and C can inherit from L, it is necessary to require RelMethodLookup(A||L, D) and RelMethodLookup(C||L, B) in order to be able to show properties as expressed in proposition P.6.4. Similarly for no external inheritance and reliable if-sentences.

Reliable message sending from L is required if L itself executes sentences. It is more typical that a library only keeps inactive methods, and the methods are only executed when they are copied into one of the components. When there are no methods which are executed while they are part of the library, ie, there are no objects with execution marks in the library, then there will be no message sending from the library. Then the reliable

message sending requirement will always be met. In practice it is then only necessary to require reliable message sending from the components and not the library.

From the above discussions it can be concluded that traditional class libraries are reliable since they are not changed or substituted during runtime.

CHAPTER 7

A Sequential Version of Omicron

As mentioned in chapter 3 there are basically two different ways to view the execution of a sentence in a program. A sentence may be seen as an atomic operation to be performed, or as an expression which is to be evaluated. Chapter 3 presented a version of the Omicron language with atomic operation semantics. This chapter will present a version with expression evaluation semantics.

The Omicron version with atomic operational semantics is called *parallel Omicron* since several objects can execute in parallel. The Omicron version with expression evaluation semantics is called sequential Omicron since only one object can execute at the time. This is modelled by only having one execution mark in any configuration. The main difference between parallel and sequential Omicron is that in sequential Omicron a sender of a message waits for a return before it continues to execute. The return include a value: an object name. In contrast, in parallel Omicron a sender of a message does not wait for a return, but continues to execute right after the message is sent.

When the two alternatives are compared, it is clear that in many ways the atomic operation semantics gives the simplest calculus. The simplest version is therefore chosen as the main version and was presented in chapter 3, used in the discussions in chapter 5 on reliability requirements and used when showing reliable substitution in chapter 6. The motivation behind also presenting the sequential version of Omicron is to argue that the reliability requirements are of the same kinds for both parallel and sequential object-oriented systems.

To avoid repeating what has been said in chapter 3 to 6, this chapter just concentrates on those formal aspects of sequential Omicron which diverge from the definitions and proofs done for parallel Omicron.

Section 7.1 presents the syntax and formal semantics of sequential Omicron.

Section 7.2 presents a definition of observable similarity which are of the same kinds as those found in chapter 4.

Section 7.3 argues that the same reliability requirements are necessary for both versions of Omicron.

Section 7.4 argues that the reliability requirements are equal for sequential and parallel Omicron, and also for some other versions of Omicron which have been made to model different features of object component systems and languages.

7.1 Syntax and Semantics

The syntax of sequential Omicron is a version of the syntax of parallel Omicron in chapter 3. The formal semantics of sequential Omicron is defined by the same techniques as used to define parallel Omicron's formal semantics in chapter 3.

The main difference between parallel and sequential Omicron objects is the semantics of the sentences. Parallel Omicron has atomic action semantics while sequential Omicron sentences have expression evaluation semantics. This means that sequential Omicron sentences are expressions which are evaluated to a value, ie, an object name. There are five kinds of expressions:

- slot lookup - evaluates to the value of a slot
- message send - evaluates to the returned value
- clone - evaluates to the name of the new object
- if-test - v=w t f evaluates "if v = w then t else f"
- assignment - evaluates to the assigned value

Message sending in sequential Omicron is like a function call in that a sender waits for a return. The return include a value: an object name.

The syntax is created to give a model of a sequential object-oriented configuration where there is a deterministic choice as to which expression is executed. This is modelled by only having one execution mark in any configuration.

7.1.1 Sequential Omicron syntax

The syntax of sequential Omicron is defined through the use of extended BNF. Terminal symbols are given in **bold font**.

Configuration	::=	$\text{Object}^{*}_{\|}$		
Object	::=	name **: (** Slots, Body **)**		
Slots	::=	**[** $\text{SlotDef}^{*}_{,}$ **]**		
Body	::=	$\text{exp}^{*}_{;}$	|	expression list
		exeBody		Body with execution or return marks
exp	::=	name	|	slot lookup
		name **.**name **(** $\text{name}^{*}_{,}$ **)**	|	Send message
		name **clone**	|	the value of the exp is the name of the new object
		name = name name name	|	If-test: v=w t f =>"x := if v == w then t else f. ^x"
		exp => name		Assign the value of exp to the slot with name 'name'
exeBody	::=	name* Mark { => name } $\text{exp}^{*}_{;}$		
Mark	::=	ReturnMark | ExeMark		
ReturnMark	::=	$\mathbf{?_n}$		
ExeMark	::=	$\mathbf{\$_n}$		
SlotDef	::=	SlotName → Val		
Val	::=	name | **this**		
SlotName	::=	name | :name | name✯ | :name✯		
name	::=	{ char^{+} }		
char	::=	**a** | … | **z** | **A** | … | **Z** | **0** | … | **9** | **+** | **-** | ***** | **/** | **_** | **:**		

The syntax is close to the syntax of parallel Omicron, with the most notable exceptions being the execution and return marks and the assignment expression. The semantics is also very similar in the two versions of Omicron, but parallel Omicron has an atomic action semantic for the expressions, while the sequential version has an expression evaluation semantics. This is shown by the rules of action.

As hinted above, the execution and return marks in sequential Omicron are handled differently than the execution mark in parallel Omicron. Each execution mark has an integer subscript. This is used to keep track of where to return to after an object is finished executing and the execution mark has come to the end of the object's body. Initially there must be one and only one object body with an execution mark $\$_1$ in a configuration. The execution mark $\$_1$ defines where the execution is to start. Other system expressions are created by the transition rules and should not be part of the initial configuration.

When a message is sent an execution mark is placed in the beginning of the body of the newly created method object and the execution mark in the sender is replaced by a return mark. If the execution mark in the sending object was $\$_n$, the execution mark in the new method object will be $\$_{n+1}$ and the return mark will be $?_n$. When the execution mark is $\$_{n+1}$ and a return is made, the execution mark will be removed and the return mark $?_n$ will be replaced by an execution mark $\$_n$.

The meta functions for getting values from the Omicron expressions, what is called map notation in parallel Omicron, is similar in sequential Omicron. This means that the function for selecting an object in a configuration, eg, C(o), the slots of objects, eg, C(o).Slots, the body of an object, eg, C(o).Body, the owners of a slot, eg, owner(C,o,s), assigning a value to a slot, eg, C(o:s:=j) etc. are defined as for the parallel version of Omicron. The translation of these functions from parallel Omicron to sequential Omicron is intuitive since the representations of slots and objects in a configuration are equal in the two versions of Omicron. The only difference is the sentences / expressions in the bodies of the objects and these are not involved in the definitions of these functions.

Syntactically correct configurations

A *syntactically correct* configuration is a configuration which could be derived at by using the syntax rules and where each object has a unique name and where there is maximum one execution mark $\$_n$ and the return marks $?_1..?_{n-1}$. Also, any object-body has at most one execution mark or return mark.

7.1.2 Sequential Omicron Semantics

Definition: Transition and action

A transition is of the form:

$$A \xrightarrow{\alpha} A'$$

Intuitively, this transition means that the configuration A can evolve into A', and in doing so performs the action α. The set of all such actions is denoted $\mathscr{A}$.

Sequential Omicron have more different kinds of actions than parallel Omicron in that there are there are six types of actions a configuration can perform as opposed to four in the parallel version of Omicron. The kinds of actions which can come from execution of a sequential Omicron system are:

e->o.s	a *lookup action* from a sentence in the object named e: the slot s in the object o is looked up
e->o.m(j)/k	a *message-send action* from a sentence in the object named e: o gets the message m with j as a parameter, k is the name of the executing method which is the result of the message
e->return(o, j)	a *return action* from a sentence in the object named e where the value j returned to the object named o.
e->(o.s):=j	an *assignment action* from a sentence in the object named e where the slot s in the object e gets the value j
e->clone(k/j)	a *clone action* from a sentence in the object named e where the object j is copied and given a new name k
e->error	an *error action* from a sentence in the object named e.

The actions are described formally through a transition system found below.

The rules of action of parallel Omicron were not confluent, ie, the configuration resulting from applying a set of transitions to a configuration may depend on the order in which the transitions are applied. This is in

correspondence with the intuition that the result of executing a parallel program depends on the execution order. The transitions may also be applied to a configuration in any order, as long as the transition is legal by the premises in the rules of action below. However, due to the way the execution and return marks are handled, the rules of action are confluent since there is always only one or no rule applicable to a configuration.

Definition: Transition relation and rules of action
The transition relation, denoted $\xrightarrow{\alpha}$, is the smallest relation between object configuration expressions satisfying the below *rules of action.* All the names in the rules are meta-variables. Informal descriptions of the rules are given before each rule.

We here consider a system e:(M, S_1 $\$_n$ sentence; S_2) || C where S_1 and S_2 are sequences of sentences, possibly empty. For the return rule the system is a little bit different in that the execution mark is the last element in the body of the executing object. Note that || is ACI (Associative, commutative and has the empty configuration as identity). The rules are given by transitions for the different kinds of symbols and expressions which may come after an execution mark: slot value, message-send expression, end of body, assignment expression, if-expressions, clone expression or none of the listed symbols or expressions.

Comments to the rules and actions:
Compared to the rules of action for parallel Omicron, the sequential Omicron rules change the derived configurations more. In parallel Omicron, sentences were not changed since the only change to the body of the executed object was the movement of the execution mark. In sequential Omicron, an executed expression is replaced with the result of evaluating the expression.

All rules except the error rule have relatively weak requirements, requiring that the slots referred to in the executed sentence have owners. In addition the rules make requirements on the values of the different slots. These requirements depend on the kind of sentence they apply to. This is further discussed below.

There are six kinds of actions, while there are seven rules. This is because both the LOOKUP-rule and the test rules give lookup actions. The test rules for evaluating if-expressions says that the result of executing such an expression is three lookup actions.

The lookup rule:

The first rule defines how the simple expressions consisting of just a slot name is handled. The slot name expression creates a slot lookup action of the form e->o.s where e is the object holding the slot lookup expression, o is the name of the owner of the slot and s is the slot name. The rule is defined so that the value in the specified slot replaces the slot name in the list of expressions and the execution mark is advanced to stand behind the new value.

LOOKUP-rule:

$$\frac{@C(e{:}s)}{e{:}\,(M,\ S_1\$_n\ s;\ S_2)\ ||\ C\ \xrightarrow{e->o.s}\ e{:}\,(M,\ S_1\ j\ \$_{n+1}S_2)\ ||\ C}$$

where j = C(e:s) and o = owner(C,e,s)

The message-send rule:

The next rule models message sending. The sending of a message gives a message-send action of the form e->o.m($\bar{p}$), where e is the name of the object with the executed expression, o is the object receiving the message with selector m and parameters $\bar{p}$. For this rule to be applicable there must be an object named o. o is denoted the receiver of the message. Also, a slot named m must be found in the receiver and this slot must hold the name of some object which has as many input slots as there are parameters in the message ($\#\bar{v} = \#\bar{t}$). This object is copied and the copy is updated with the parameter values in the input slots (ie, the C(j).Slots[$\bar{v} \rightarrow \bar{p}$] part).

The message rule replaces the original execution mark $\$_n$ with a return mark $?_n$ and inserts an execution mark ($\$_{n+1}$) into the new copy. The new execution mark gets a new value in its index (n+1) to control which return-mark to return to. The message rule models that the sender will wait for a return.

MESSAGE-rule:

$$\frac{@C(e:\bar{t}) \wedge C(e:s) \in C.Dom \wedge C(o:m) \in C.Dom \wedge \#C(j).inputs = \#\bar{t}}{e: (M, S_1\ \$_n\ s.w(\bar{t});\ S_2) \parallel C \xrightarrow{e->o.m(\tilde{p})/k} e: (M, S_1\ ?_n\ S_2) \parallel k: (C(j).Slots[\bar{v} \rightarrow \bar{p}], \$_{n+1}\ C(j).Body^{-\$?}) \parallel C}$$

where j = C(o:m), $\bar{v}$ = C(j).inputs, $\bar{p}$ = C(e:$\bar{t}$), m = C(e:w), o = C(e:s) and k is a fresh name and $C(j).Body^{-\$?}$ means that any existing execution or return mark is removed

The return rule:

This rule shows how return is done from an execution mark at the end of an object body to a return mark. When the return rule is applied then a return action of the form e->return(o, j) is created. e is the name of the object returned from, o is the name of the object returned to and j is the returned value. The object which is returned from is changed into an object with no body, while the last value in the old body is placed in the position of the return mark and an execution mark is inserted behind it.

RETURN-rule:

$$\frac{}{e: (M, j_1 \ldots j_m\ \$_{n+1}) \parallel o: (N, S_1\ ?_n\ S_2) \parallel C \xrightarrow{e->return(o,j_m)} e: (M,) \parallel o: (N, S_1\ j_m\ \$_n\ S_2) \parallel C}$$

The assignment rule:

This rule models slot update. The assignment of a value to a slot gives an assignment action of the form e->o.s:=j, where e is the object where the executed expression is found, o is the owner of the updated slot, s is the name of the updated slot and j the new value of the slot. The result of evaluating the assignment expression is that the slot named s is updated to hold the value j. The value j replaces the assignment expression and the execution mark is advanced passed this expression.

$$\frac{@C(e:s)}{e: (M, S_1\ j\ \$_n => s;\ S_2) \parallel C \xrightarrow{e->(o\ s):=j} (e: (M, S_1\ j\ j\ \$_n\ S_2) \parallel C)\ [e:s := j]}$$

where o = owner(C,e,s)

The IF-test rules:

The below rules define tests of equality. If the test is true, the value of the third slot will be returned, and if false the value of the last slot will be returned. All rules create three slot lookup actions. The rules only change the internals of the executing objects' bodies.

IF-true rule:

$$\frac{@C(e:<v,w,t>) \wedge C(e:v) = C(e:w)}{e: (M, S_1\ \$_n\ v=w\ t\ f;\ S_2) \parallel C \xrightarrow{e->o_v.v} \xrightarrow{e->o_w.w} \xrightarrow{e->o_t.t} e: (M, S_1\ j\ \$_n\ S_2) \parallel C}$$

where j=C(e:t), o_v = owner(C,e,v), o_w = owner(C,e,w) and o_t = owner(C,e,t)

IF-false rule:

$$\frac{@C(e:<v,w,f>) \wedge C(e:v) \neq C(e:w)}{e: (M, S_1\ \$_n\ v=w\ t\ f;\ S_2) \parallel C \xrightarrow{e->o_v.v} \xrightarrow{e->o_w.w} \xrightarrow{e->o_f.f} e: (M, S_1\ j\ \$_n\ S_2) \parallel C}$$

where j=C(e:f), o_v = owner(C,e,v), o_w = owner(C,e,w) and o_f = owner(C,e,f)

The clone rule:

The next rule defines object creation by cloning. Object creation gives a clone action of the form e->clone(k/l), where e is the object where the executed expression is found and k / l means that an object named l is copied and the new copy gets the name k. k is a new unique name within the configuration. The new object is inserted into the configuration. The rule is applicable when the slot t holds the name of an object in C. A new object is then created with the same definition as the referenced object. The clone expression is replaced with the name of the new object.

$$\frac{@C(e{:}t) \wedge C(e{:}t) \in C.Dom}{e{:}\ (M,\ S_1\ \$_n\ t\ clone;\ S_2)\ ||\ C \xrightarrow{e->clone(k/l)} e{:}\ (M,\ S_1\ k\ \$_n\ S_2)\ ||\ k{:}\ C(l)^{-\$?}\ ||\ C}$$

where l = C(e:t) and k is fresh and
$C(l)^{-\$?}$ means that any existing execution or return mark is removed

The error rule:

When no other rules of action are applicable to an object which has an execution mark in its body, then an error action is given. The error action has the form e->error where e is the name of the object holding the erroneous sentence. The object named e is removed from the configuration. This models termination of the object.

$$\frac{\text{no other rule applicable to the first sentence in } S_2}{e{:}\ (M,\ S_1\ \$_n\ S_2)\ ||\ C \xrightarrow{e->error} C}$$

The informal requirement "no other rule of action is applicable to the first sentence in S_2" can be replaced by a set of rules with formal requirements. The rules in the set replacing the error rule would be equal to the premises of the other rules except that the requirements would be negated and the actions would be error actions instead of lookup, clone, return, assign and message-send actions.

Proposition P.7.1 : The rules of action preserve syntactic correctness
Applying a rule of action to a syntactic correct configuration gives a syntactic correct configuration.

Proof:
Done by cases for the different rules:

LOOKUP:	The change in the object comply with executing body syntax
MESSAGE:	The change in the object old object comply with executing body syntax and the new object comply with the defined syntax.
RETURN:	The returning object comply with object definition syntax and the changes in the receiving object comply with executing body syntax
ASSIGN:	The change in the object comply with executing body syntax
IF-rules:	The change in the object comply with executing body syntax
CLONE-rule:	The new object comply with object definition syntax and the change in the old object comply with executing body syntax
ERROR-rule:	The object with the erroneous sentence is removed. The resulting configuration comply with configuration definition syntax.

❑

7.1.3 Basic notations and definitions

This section gives some basic definitions and notations which are used when expressing properties of and reasoning about sequential Omicron configurations. Some of the concepts have been defined in chapters 1 and 2, and here they are defined specifically for sequential Omicron expressions. The following definitions are equal to the notation and definitions of parallel Omicron given in chapter 3.

Definition: Sequences of transitions and actions: $\xrightarrow{\bar{\alpha}}$, $\bar{\alpha}$, $\bar{\alpha}_i$

$C \xrightarrow{\bar{\alpha}} C'$ denotes a sequence of zero or more transitions $\xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n}$ from C to C' as defined by the rules of action and $\alpha_i \in \mathscr{A}$.

Definition: Derivation of a configuration

The configurations derived by one or more transitions from a configuration C, are denoted the derivations of C.

$$\text{Derivations}(C) = \{\ C' \mid C \xrightarrow{*} C'\ \}$$

The traces of a configuration are the sequences of actions derived from execution of a configuration. This can be formally defined as follows:

Definition: The traces of a configuration

The traces of a configuration C is the set of all sequences of zero or more transitions $\xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n}$ from C to some C' as defined by the rules of action and $\alpha_i \in \mathscr{A}$.

$$\text{Traces}(C) == \{\ \overline{\alpha} \mid \exists\, C' :\ C \xrightarrow{\overline{\alpha}} C'\ \}$$

We let $\alpha \in \text{Traces}(C)$ abbreviate $<\alpha> \in \text{Traces}(C)$.

The result of executing a sequential system is a possibly infinite sequence of actions. Therefore, in sequential Omicron each action sequence in a system's traces will be equal to the head of some other action sequence in the traces or it will be the longest action sequence in the prefix closed set. This is true because there will always be only zero or one execution mark in any given configuration.

Definition: Description and execution parts of actions

an action is seen as consisting of two parts:

$\alpha = \alpha.\text{exe} \rightarrow \alpha.\text{dcs}$

where $\alpha.\text{exe}$ is the name of the object where the executed sentence which gave the action was found and

$\alpha.\text{dsc}$ is the rest of the action - the description part of the action.

If $\alpha.\text{exe} \in C$ then C is said to be the *owner* of the executed sentence. It is also said that α is from execution of a sentence in C, or just α from C.

7.1.4 Some properties of configurations of objects

The two next observations state properties which relate actions and derived configurations. They state different aspect of the fact that equal actions give equal derived configurations. These properties are the same as for parallel Omicron.

Observation O.7.1 : A derived configuration is uniquely determined by the action

When two transitions from the same configuration are equal then the result configurations are equal, ie, the resulting configuration is uniquely determined by the action. This can be formally stated:

$$C \xrightarrow{\alpha} C' \wedge C \xrightarrow{\alpha} C'' \Rightarrow C' \equiv C''$$

Observation O.7.2 : Each object is deterministic and gives equal derived configurations

When the exe-part of two actions from the same configuration are equal then they are actions from execution of the same sentence. This is because there is only one sentence with an execution mark in front of it in each object and then only one rule of action is applicable to an object with an execution mark. Then the actions are equal and the transitions are equal. This can formally be stated:

$$C \xrightarrow{\alpha} C' \wedge C \xrightarrow{\beta} C'' \wedge \alpha.\text{exe} = \beta.\text{exe} \Rightarrow \alpha \equiv \beta \wedge C' \equiv C''$$

This property implies that each object has a deterministic behaviour.

Next we show that no rules of action are applicable to a configuration with no execution marks and at least one rule of action is applicable if the configuration has an execution mark. This shows that the rules of action may be viewed as describing execution of Omicron systems in an appropriate way. First we formally define termination by giving a definition of terminal configurations.

Definition: Terminal configurations

A configuration is terminal if there are no execution marks in any of the objects in the configuration. The set of all terminal configurations is denoted $\mathscr{C}_{\text{Term}}$.

Proposition P.7.2 A configuration is terminal iff no rules of action are applicable
$C \in \mathscr{C}_{Term} \Leftrightarrow Traces(C) = \{<>\}$

Proofs:
Proof of $\Rightarrow$: Since $\$_n$ is not in the body of any object in the configuration, then there are no actions from execution of sentences in the configuration and then the traces of the configuration only include the empty action sequence <> and the proposition holds.
Proof of $\Leftarrow$:
We show then when there are non-empty action sequences in Traces(C) then C is not a terminal configuration.
Since $C \notin \mathscr{C}_{Term}$ then there exists one object with an execution marks in front of a list of sentences. It must therefore have one of the following forms and then a rule of action is applicable:

Form	*Applicable rule*
Ref expressions:	
…$\$_n$s…	- LOOKUP rule or ERROR rule
Send expression:	
…$\$_n$s.w($\tilde{t}$)…	- MESSAGE or ERROR rule
Assignment expression:	
… j $\$_n$ => s …	- ASSIGN rule or ERROR rule
End of object body expression:	
… $j_1 \ldots j_m$ $\$_n$ >	- RETURN rule or ERROR rule
If expression:	
$\$_n$ v=w t f	- one of the IF-rules or ERROR rule
Clone expression:	
$\$_n$ t clone	- CLONE rule or ERROR rule

Thus $\$_n$ is in the body of some object in the configuration, and then the configuration is not terminal and the proposition holds for this case.
❑

7.1.5 Combined configurations

Below, the prime of a configuration is formally defined. This definition repeats some of what is defined in the rules of action, in order to precisely describe what D' means. The definition is quite similar to the definition of the prime of parallel Omicron configurations. The only difference is that the NewNames function refer to sequential Omicron actions.

Definition: The NewNames function

Given an action sequence $\bar{\alpha}$ and a set of object names O. We define the new names function to return the set of names of all objects created by the actions in $\bar{\alpha}$ and which are

- created by cloning objects in O, or created by cloning clones of objects in O,
  or cloning clones of clones of objects in O etc. and
- method copies which are created when the receiver is an object in O or a clone of an object in O,
  or a clone of a clone of an object in O etc.

The function is formally defined as follows:

NewNames(<>, O) == Ø

NewNames(<α> & $\bar{\alpha}$, O) ==

case α.dsc of
clone(k/o), o.m($\bar{p}$)/k : if o ∈ O then NewNames($\bar{\alpha}$, O∪{k}) ∪ {k}
else NewNames($\bar{\alpha}$, O)
otherwise : NewNames($\bar{\alpha}$, O)

We let NewNames($\bar{\alpha}$, D) abbreviate NewNames($\bar{\alpha}$, D.Dom)

Definition: The prime of a configuration

The prime of D, denoted D', is defined relative to a configuration B and an action sequence $\bar{\beta} \in$ Traces(B||D). We define it as follows:

$$\text{prime}(D, B, \bar{\beta}) == D' \Leftrightarrow B||D \xrightarrow{\bar{\beta}} B'||D' \wedge D'.Dom = (D.Dom \cup \text{NewNames}(\bar{\beta}, D))$$

This notation is used in the rest of this thesis in order to denote derivations of the different parts of a configuration.

Combinable configurations and visible object names are defined as for the parallel version of Omicron.

7.2 Observable Actions

This section defines and proves the same things for sequential Omicron as was defined for parallel Omicron in chapter 4.

7.2.1 Observable and hidden actions

Definition: Observable Action

Given an action α and a configuration of objects C. The action α is observable from the configuration C if the action change slots in an object in C, clone an object in C or is a message send to an object in C. The actions observable from the objects in C is denoted obs(C) and is formally defined:

obs(C) =

{ e->o.s := j	| o ∈ C } ∪
{ e->clone(k/j)	| j ∈ C } ∪
{ e->o.m($\overline{p}$)/k	| o ∈ C } ∪
{ e->o.s	| o ∈ C } ∪
{ e->return(o, p)	| o ∈ C } ∪
{ e->error	| e ∈ C }

Given a configuration C, then obs(C) abbreviates obs(C.Dom). We say that, "α is observable from C" when $\alpha \in$ obs(C). Similarly, "α is not observable from C" means $\alpha \notin$ obs(C). Note that there are observable actions $\alpha \in$ obs(C) where α.exe $\in$ C, while there may also be actions where α.exe $\in$ C such that $\alpha \notin$ obs(C).

Observable action sequences obs($\overline{\alpha}$, C), hidden action $\alpha \otimes$ D and silent action $\alpha \otimes$ D are as defined for parallel Omicron. There will be no difference in the definitions, except that for sequential Omicron the term obs(C) will refer to the above definition instead of the obs(C) definition given in chapter 4.

Proposition P.7.2.1 Silent actions are hidden actions

$\forall$ A, D, α:
$\alpha \in$ actions(A||D) $\Rightarrow$ ($\alpha \otimes$ D $\Rightarrow$ $\alpha \oplus$ D)

Proof:

Assume $\alpha \otimes$ D and A||D $\xrightarrow{\alpha}$ A'||D'. We show that D' $\equiv$ D and then by definition of hidden actions this gives $\alpha \oplus$ D.
Cases for the different kinds of actions α:

e->o.s : Since the action is not observable from D then o $\in$ A
Since e $\in$ A the executed sentence is in A and then D is not changed.

e->o.m($\overline{p}$)/k : Since the action is not observable from D then o $\in$ A
Since the action came from a send sentence in A and the receiver is an A-object, the method-copy will be placed in A by definition of primed configurations, and then there will be no change in D.

e->return(o i): Since the action is not observable from D then o $\in$ A
Since e, o $\in$ A the changes to execution and return marks only apply to objects in A and then D is not changed.

e->o.s:=j : Since the action is not observable from D then we have o $\in$ A
Since o $\in$ A, the updated slot is in A, and then D is not changed.

e->clone(k/j) : Since the action is not observable from D then j $\in$ A
Because the clone original is an object in A, then by the definition of primed configurations, the new clone is placed in A and then D is not changed.

e->error : Since the action is not observable from D then e $\in$ A
Since the action came from A, the terminated object is in A and this will not change D.

Proposition P.7.2.2 Hidden actions are silent actions except for trivial assignment

For all configurations A and D and action α where $\alpha \in$ Traces(A||D) and where α is not a trivial assignment action, ie, where the slots in D get the same values as they had, we have that if α is a hidden action, then α is a silent action, ie, we have :

$\forall$ A, D, $\alpha \bullet \alpha \in$ Traces(A||D) $\Rightarrow (\alpha \otimes D \Leftarrow \alpha \oplus D)$

Proof:
We can then show that except for trivial assignment we have for all cases where $\alpha \otimes D$ do not explicitly hold we have D' $\neq$ D where $\neq$ denote textual inequality. This means showing

$\neg (\alpha \otimes D) \Rightarrow \neg (\alpha \oplus D)$

This gives $\alpha \oplus D \Rightarrow \alpha \otimes D$. By definition of silent action we have

$\neg (\alpha \otimes D) = \neg (\alpha.exe \notin D \wedge \alpha \notin obs(D)) = \alpha.exe \in D \vee \alpha \in obs(D)$

We therefore show:

$\alpha.exe \in D \vee \alpha \in obs(D) \Rightarrow$ D' $\neq$ D where $\neq$ denote textual inequality.

If $\alpha.exe \in D$ then the execution mark is moved and we do not have D' $\equiv$ D and then D' $\neq$ D.

Cases for different actions when $\alpha \in obs(D)$:

e->o.s : Since the action is observable from D then $o \in D$
Since $o \in D$ the updated slot is in D giving D' $\neq$ D.

e->o.m($\bar{p}$)/k : Since the action is observable from D then $o \in D$
Since the receiver is a D-object, the method-copy will be placed in D by definition of primed configurations, the new method copy will be added to D giving D' $\neq$ D.

e->return(o, i): Since the action is observable from D then $o \in D$
Since $o \in D$ the changes to the return marks applies to an objects in D and then D giving D' $\neq$ D.

e->clone(k/j) : Since the action is observable from D then $j \in D$
Because the clone original is an object in D, then by the definition of primed configurations, the new clone is placed in D giving D' $\neq$ D.

e->o.s:=j : Since the action is observable from D then $o \in D$
Since $o \in D$, the updated slot is in D giving D' $\neq$ D provided the value of o.s was different from j.

e->error : Since the action is observable from D then $e \in D$
Since the terminated object is in D this object is removed from D giving D' $\neq$ D.

❑

7.2.2 Observable equality and refinements

Below the observably equal actions relation definition of chapter 1 is reformulated for Sequential Omicron actions. This relation is similar to the observably equal actions relation for parallel Omicron, except that the below relation is expressed for sequential Omicron actions.

Definition: Observable equality relative to a set of object names; $\sim_O$

Two object names, e,f, are observably equal relative to a set of object names O, denoted e $\sim_O$ f, if:

- either both are equal
- or none of them are names in O

This can be formally defined:

$$e \sim_O f \quad == \quad e \in O \vee f \in O \Rightarrow e = f$$

The definition of observable equality can be lifted to sequences of names as follows:

$$\langle e_1,..,e_n \rangle \sim_O \langle f_1,..,f_m \rangle == \quad n = m \wedge \forall i \in \{1..n\} : e_i \sim_O f_i$$

An action α is said to be observably equal to an action β relative to a set of object names O, denoted $\alpha \sim_O \beta$, iff $\alpha.exe \sim_O \beta.exe$ and $\alpha.dsc \sim_O \beta.dsc$. This is formally stated:

$$\alpha \sim_O \beta \quad == \quad \alpha.exe \sim_O \beta.exe \wedge \alpha.dsc \sim_O \beta.dsc$$

where observable equality of the description part of actions is defined as follows:

$o.x(\bar{q})/k \sim_O p.y(\bar{p})/l \quad == \quad o \in O \vee p \in O \Rightarrow \langle o, x, k\rangle = \langle p, y, l\rangle \wedge \bar{q} \sim_O \bar{p})$

$o.s := i \sim_O p.t := j \quad == \quad o \in O \vee p \in O \Rightarrow o.s = p.t \wedge i \sim_O j$

$clone(k/o) \sim_O clone(l/p) \quad == \quad o \in O \vee p \in O \Rightarrow \langle o, k\rangle = \langle p, l\rangle$

$(o.s) \sim_O (p.t) \quad == \quad o \in O \vee p \in O \Rightarrow o.s = p.t$

$return(o\ i) \sim_O return(p\ j) \quad == \quad o \in O \vee p \in O \Rightarrow o = p \wedge i \sim_O j$

$error \sim_O error \quad == \quad true$

The definition of observable equality of actions relative to a set of object names can be lifted to sequences of actions as follows:

$$\bar{\alpha} \sim_O \bar{\beta} \quad == \quad \forall\ i \leq \#\bar{\alpha} \bullet \alpha_i \sim_O \beta_i$$

These observable sequential relations for parallel and sequential Omicron have the same properties with respect to equivalence and concatenation properties. A refinement relation between sequential Omicron configurations can be defined as the refinement relation for parallel Omicron:

Definition: Refinement relation; $A \leq_D B$

Given two systems A||D and B||D. The configuration A is a refinement of B relative to D, denoted $A \leq_D B$, if the traces of A||D is a refinement of the traces of B||D relative to D. This can be formally defined as follows:

$$\forall\ \bar{\alpha} : \bar{\alpha} \in Traces(A||D)\ \exists\ \bar{\beta} : \bar{\beta} \in Traces(B||D) \wedge$$

$$\bar{\alpha} =_{D'} \bar{\beta} \wedge (endColab(A, D, \bar{\alpha}) \Rightarrow endColab(B, D, \bar{\beta}))$$

where also observably equal action sequences $\bar{\alpha} =_{D'} \bar{\beta}$ and the endColab()-function is defined equally for the two versions of Omicron.

7.3 Reliability Requirements

This section argues that the same reliability requirements are necessary for both parallel and sequential Omicron. All the examples of chapter 5 do not show properties which are particular to the parallel properties of parallel Omicron. Instead they are related to properties which are common to both the sequential and parallel versions of Omicron. To avoid repeating examples and text from chapter 5 here in this section, this section refers heavily to what is written in chapter 5. The subsections are named and listed in the same order as in chapter 5):

Name substitutions:
Name substitutions apply to configurations in parallel and sequential Omicron in corresponding ways. The problem with substitution of slot names will be equal in both versions of Omicron. Therefore the definition of safe names in sequential Omicron configurations will be similar to the definition for parallel Omicron. The only difference will be that the definition will refer to sequential Omicron sentences instead of the parallel Omicron sentences used in the definition in chapter 5. This will lead to reliable substitutions applied to configurations with safe names having the same properties as observed in chapter 5.

Cases with external inheritance
The examples in section 5.3.1 used inheritance and input slots together with message sending to illustrate problems with reliability when there is external inheritance. Inheritance and message sending is found in both versions of Omicron. We can therefore conclude that the requirement "no external inheritance" is also a reliability requirement for sequential Omicron. The formal definition of this requirement, in particular the noExt()-function will be equal for both versions of Omicron.

Cases with if-sentences
Assume that we have the following case where we have an if-sentence in a method in C:

B = b_1 : ([m->p],) || p : ([],) ||
b_2 : ([],)

A = a : ([m->p],) || p : ([],)

D = d : ([s->b_1, t->b_2, w->m], $ s.w();)

C = c : ([s->b_1, t->b_2, w->m, x->q], $ s=t x s => s; s.w();) ||
q : ([m->r],) ||
r : ([],)

The substitution:
$\sigma = \{a/b_1\}\ \{a/b_2\}$
is used to specialise C and D for combination with A, and the configurations fulfil the reliability requirements of all previous sections in this chapter.

When executing the configurations we then get the actions and action sequences reusing the conventions for α, β, $\bar{\gamma}$ and $\bar{\delta}$ from chapter 5 except the reliable if-sentence requirement:

α = < d->a.m() >
β = < d->b_1.m() >
$\bar{\gamma}$ = <c->p.s, c->p.t, c->p.s, c->p.s:=b_2, c->b_2.m() >
$\bar{\delta}$ = < c->p.s, c->p.t, c->p.x, c->p.s:=q, c->q.m() >

In $\bar{\gamma}$ we have the action c->c.s:=b_2 because $b_1 \neq b_2$ and then c.s get the value of s which is b_1.
In $\bar{\delta}$ we have the action c->c.s:=q because a = a and then c.s get the value of x which is q.

We then have $\bar{\delta} \leq_C \bar{\gamma}$, $\bar{\gamma} \leq_B \beta$ and $\alpha \leq_D \beta$, but we do not have $\bar{\delta} \leq_A \alpha$. To avoid situations such as this we must require that every if-test in C will give the same result both when combined with A and B. This must also hold for derivations of C from A||Cσ and B||C.

We then see that even though the semantics for if-sentences in parallel Omicron and if-expressions in sequential Omicron are quite different, the problems related to reliability are the same. In the example in chapter 5, the critical expression was the if-sentence with a following message-send expression which resulted in an unreliable observable action due to the unreliability of the if-sentence:

s := (s=t x y); s.w();

In sequential Omicron, and in the example above, this corresponds to:

s = t x y => s; s.w();

Both versions will result in the s-slot getting a new value depending on the result of the if-test The problem is related to the value of the s-slot after the execution of the if-sentence/expression, and the problem is then similar in both cases. We will therefore have the same reliability requirements associated with if-sentences/expressions in both cases.

From this we can conclude that the same reliability requirements on reliable if-sentences is necessary for both parallel and sequential Omicron. The formal definition of reliable if-sentences in the sequential version of Omicron is done as follows:

RelIfSentence(C, B) ==
$\forall$ i : C, $\forall$ v, w : $\mathcal{N}$ •
($\exists$ t, f, v, w, s S_1, S_2 •
C(i).Body $\equiv$ (S_1 $\$_n$ v=w t f => s; S_2) $\Rightarrow$
(C(i:t) $\neq$ C(i:f) $\vee$ C(i:v) $\neq$ C(i:w)) $\Rightarrow$ (C(i:v) $\notin$ B $\vee$ C(i:w) $\notin$ B)

Case with message not understood errors
The example in section 5.3.3 only used message sending to illustrate problems with reliability and errors from execution of message-send sentences. Corresponding message send errors can also occur in sequential Omicron. We can therefore conclude that the requirement "reliable message sending" is also a reliability requirement for sequential Omicron and it is formally written:

RelMessageSend(A, D) ==
$\forall$ i $\in$ A : $\forall$ t, w:
(($\exists$ S_1, S_2, $\bar{p}$:
A(i).Body $\equiv$ S_1 $\$_n$ t.w($\bar{p}$); S_2) $\wedge$ A(i:t) $\in$ D) $\Rightarrow$ D(A(i:t):A(i:w)) $\in$ D

Cases with external methods
The example in section5.3.4 used inheritance slots and message-send actions which are found in both versions of Omicron. We can then conclude that the requirement "reliable method lookup" is also a reliability requirement for sequential Omicron and its formal expression is similar for both versions of Omicron.

Cases with slot names as parameters
This example only used message sending which is found in both versions of Omicron. We can therefore conclude that the reliability requirements associated with slot names as parameters also apply to the sequential version of Omicron.

Cases with observably equal names as parameters
As in previous examples, this example used input slots and message-send actions which are found in both versions of Omicron. We can then conclude that it is necessary to make a stronger requirement on names than that the names are weakly equal. For parallel Omicron we defined an observably similar actions relation relative to a reliable substitution. This we must do for sequential Omicron as well in order to define a reliable refinement relation. We define this relation as follows:

Definition: Observably similar actions relative to a substitution; $\alpha \leq_{O,\sigma} \beta$
Given a set of object names O and a substitution σ which is reliable relative to the set of object names O, denoted, RelSubst(σ, O), we define observable similarity as follows:

o.x($\bar{q}$)/k ~$_{O,\sigma}$ p.y($\bar{p}$)/l == o $\in$ O $\vee$ p $\in$ O $\Rightarrow$ <o, x, k> = <p, y, l> $\wedge$ $\bar{q}$ = $\bar{p}\sigma$)

o.s := i ~$_{O,\sigma}$ p.t := j == o $\in$ O $\vee$ p $\in$ O $\Rightarrow$ o.s = p.t $\wedge$ i = jσ

clone(k/o) ~$_{O,\sigma}$ clone(l/p) == o $\in$ O $\vee$ p $\in$ O $\Rightarrow$ <o, k> = <p, l>

(o.s) ~$_{O,\sigma}$ (p.t) == o $\in$ O $\vee$ p $\in$ O $\Rightarrow$ o.s = p.t

return(o, i) ~$_{O,\sigma}$ return(p, j) == o $\in$ O $\vee$ p $\in$ O $\Rightarrow$ o = p $\wedge$ i = jσ

error $\sim_{O,\sigma}$ error == true

$\alpha \sim_{O,\sigma} \beta$ == α.exe $\sim_O \beta$.exe $\wedge$ α.dsc $\sim_O \beta$.dsc

Compared to the definition of observably similar actions we here have $\bar{q} = \bar{p}\sigma$ instead of $\bar{q} \sim_O \bar{p}$ and $i = j\sigma$ instead of $i \sim_O j$.

Note that this relation is not an equality relation as it is not commutative in that
we do not have $\alpha \sim_{O,\sigma} \beta \Rightarrow \beta \sim_{O,\sigma} \alpha$
This is because we will not necessarily have $\bar{q} = \bar{p}\sigma \Rightarrow \bar{p} = \bar{q}\sigma$.

Observably similar action sequences and the refinement relation with specialisation for sequential Omicron configurations are also defined as for parallel Omicron. In the definition of the relation we use the following notation:

$\bar{\alpha}$ /obs&exe(O) where obs&exe(O) = obs(O) $\cup$ { α | α.exe $\in$ O}

This denotes a sequence of actions which consists of all the action in $\bar{\alpha}$ which are O-observed and/or from O and where the actions are found in the same order as in $\bar{\alpha}$.

Definition: Observably similar action sequences relative to a substitution; $\bar{\alpha} \leq_{O,\sigma} \bar{\beta}$

An action sequence $\bar{\alpha}$ is said to be observably similar to an action sequence $\bar{\beta}$ relative to a set of object names O and a substitution σ, denoted $\bar{\alpha} \leq_{O,\sigma} \bar{\beta}$, if the following holds:

$\bar{\alpha} \leq_{O,\sigma} \bar{\beta}$ == $\bar{\alpha}$ /obs&exe(O) $\approx_{O,\sigma}$ $\bar{\beta}$ /obs&exe(O)

where $\bar{\alpha} \approx_{O,\sigma} \bar{\beta}$ == $\forall$ i $\leq \#\bar{\alpha}$: *if* $\alpha_i \in$ obs(O) *then* $\alpha_i \sim_{O,\sigma} \beta_i$ *else* $\alpha_i = \beta_i\sigma$

For sequential Omicron we also use the abbreviations $\alpha \leq_{O,\sigma} \bar{\beta}$ for $<\alpha> \leq_{O,\sigma} \bar{\beta}$ and $\alpha \leq_{O,\sigma} \beta$ for $<\alpha> \leq_{O,\sigma} <\beta>$.

The Reliable()-function used in the below definition of refinement with specialisation is defined as the corresponding function for parallel Omicron.

Definition: Refinement relation with specialisation; A $\leq_{D,\sigma}$ *B*

Given configurations A, B, D $\in \mathscr{C}$ and a substitution σ. We define a binary relation called *a refinement relation with specialisation*, denoted A $\leq_{D,\sigma}$ B, as follows:

A $\leq_{D,\sigma}$ B == $\forall$ $\bar{\alpha}$: $\bar{\alpha} \in$ Traces(A||Dσ) $\exists$ $\bar{\beta}$: $\bar{\beta} \in$ Traces(B||D) $\wedge$

A||D, B||D $\in \mathscr{C}_{Safe} \wedge \sigma \in$ B $\rightarrow$ A $\wedge$ Reliable(A, Dσ, $\bar{\alpha}$) $\wedge$

$\bar{\alpha} \leq_{D',\sigma'} \bar{\beta}$ $\wedge$ (endColab(A, Dσ, $\bar{\alpha}$) $\Rightarrow$ endColab(B, D, $\bar{\beta}$))

where D' = prime(D, A, $\bar{\alpha}$) and σ' = prime(σ, $\bar{\alpha}$, $\bar{\beta}$, A, B, D)

and a prime substitution relative to sequential Omicron configurations and actions is defined as follows (the only difference from the previous definition is the change from parallel Omicron clone action to sequential Omicron clone action):

prime(σ, α, β, A, B, D) ==
case α, β *of*
e->clone(k/i), f->clone(l/j) : *if* o $\in$ D $\wedge$ i,j $\notin$ D *then* σ + { k / l } *else* σ
e->o.m($p_1 \ldots p_n$)/k, f->o.m($q_1 \ldots q_n$)/k : *if* o $\in$ D *then* σ_n *else* σ
otherwise σ

where σ_n is defined as follows:
$\sigma_0 = \sigma$
and for i $\in$ {1..n}
σ_i = *if* $q_i \notin$ keys(σ_{i-1}) $\wedge$ $q_i \in$ B $\wedge$ $p_i \in$ A *then* σ_{i-1} + {p_i / q_i} *else* σ_{i-1}

prime(σ, α, <$\beta_1 \ldots \beta_n$>, A, B, D) == *if* n = 0 *then* σ else
prime(prime(σ, α, <$\beta_1 \ldots \beta_{n-1}$>, A, B, D), α, β_n, A, B, D)
prime(σ, <$\alpha_1 \ldots \alpha_n$>, $\overline{\beta}$, A, B, D) == *if* n = 0 *then* σ else
prime(prime(σ, <$\alpha_1 \ldots \alpha_{n-1}$>, $\overline{\beta}$, A, B, D), α_n, $\overline{\beta}$, A, B, D)

7.4 Discussion

7.4.1 Sufficiency of reliability requirements for sequential Omicron

No part of the proofs of the substitution proposition for parallel Omicron will be different if it was assumed that there is only one execution mark in any configuration, creating a deterministic system - as is the case in sequential Omicron. Also, from the above discussion it is evident that the reliability requirements do not have any relation to the parallel properties of the configurations expressed in parallel Omicron. It can therefore be assumed that the same kinds of requirements are necessary in a sequential configuration with deterministic selection of which expression to execute as in parallel configurations with non-deterministic selection of expressions to execute. The only difference in requirements is that return values have to be taken into account. This difference is not major in that the return values must be considered in the same way as parameters to messages in order to ensure reliable substitutions.

No formal proofs are given for the necessity and sufficiency of the reliability requirements for sequential Omicron. This is left for further study. However, the above observations indicate that such proofs can also be done for the sequential version of Omicron and that the reliability requirements will be necessary and sufficient also for sequential Omicron.

To see relations between reliability requirements it would also be interesting to define a translation from sequential to parallel Omicron. If the reliability requirements are similar, it should be possible to show that when the reliable refinement relation hold between the parallel Omicron versions of the configurations then this implies that a reliable refinement relation will hold between the sequential Omicron versions of the configurations as well.

7.4.2 Reliability requirements for other versions of Omicron and similar languages

To make Omicron more user friendly, ie, make it easier to translate from an object-oriented programming language or design notation to Omicron, a new version of Omicron could be created. The author has created several versions of Omicron which include statements commonly found in object-oriented languages and notations. Examples are:

- syntax for defining block-objects directly in sentences,
- automatic assignment of self-variables in methods
  (similar to how it is done in, eg, Smalltalk and SELF, C++, Java and others),
- system defined object names so that the programmer need not define object names.

When such versions of Omicron exist, then the same rewriting from the higher level language to the more primitive Omicron does not have to be done for each translation of a new language or notation. Instead the higher level Omicron can be used as a common formal language. However, for all the tested languages, it was always possible to translate to the parallel version of Omicron.

As the above discussion argues, and as have been seen by studying different versions of Omicron not presented here, it seems like the reliability requirements will be similar as long as the language and refinement relations model the object component concepts presented in chapters 1 and 2. The important properties which lead to the necessity for the reliability requirements are:

- objects as entities where the state and the name of the object is separated and where an object has an independent existence, typically existence independently of method executions (and function calls)

- message sending, where the receiver is an object and where the method to execute is decided at runtime, ie, there is dynamic binding

- object names as parameters in messages so that the collaboration structure of the objects can change during execution of the system

- inheritance between objects where the inherited elements can change during execution of the system

- if-sentence-like expressions where object behaviour depend on similarity of object names

The three first properties in combination gives the requirements on visible object names and the "reliable method lookup" and "reliable message-send" requirements. The inheritance properties gives the "no external inheritance" requirement, while the last properties gives the "reliable if-sentence" requirement.

It seems to be true that the reliability requirements of chapter 5 must hold for configurations expressed in all object-oriented languages and therefore language notation is not a major concern to us.

CHAPTER 8

How to Make Reliable Specifications and Reliable Refinements

This chapter has two purposes. One is presenting examples of how the work presented in this thesis can be used to give theoretic support to more informal work such as published advice for designing components, examples of Framework designs and practical solutions to implementing components. The other is presenting new practical advice for designing components. When the advice is used, we can hope to get more efficient development processes, easier maintainable systems and less errors in users' systems.

In the introduction chapter of this thesis, it was said that if the reliability requirements correspond to what is considered *good practice* among experienced object component system designers, this will be an indication that the presented formalisation captures important aspects of OCS design. Examples of "good practice" can be found in existing libraries of reusable components and to some extent in the design of object-oriented programming languages. "Good practice" are also described in various papers and books.

This chapter presents correspondences between the theoretically founded reliability properties and the practitioners view of good design practice. In this chapter the correspondences are shown by reformulating the theoretically expressed reliability requirements of chapter 5 to advice for making reliable refinements and reliable specifications. These advice are then compared to related work, ie, advice and practical solutions used by component designers.

One conclusion is that there are many similarities between the presented theoretic work and related practical advice. Another conclusion is that there are also some reliability requirements which are not covered by "good practises". This is in line with the findings that the component designers definition of "similar components", as presented in chapter 4, does not have the properties expressed by the substitution proposition. A third conclusion is that quite a lot of further theoretic and practical work is necessary to ensure reliable refinements in a general, practical case. However, there are also some simple lessons to be learned which can help create reliable systems and which should be easy to incorporate into existing development methods and OCS design practises.

Further details on the structure of this chapter are given in section 8.1: Overview of the chapter.

8.1 Overview of the Chapter

The following sections present the various reliability requirements and their practical consequences. The sections also present published advice for designing components, examples of Framework designs and practical solutions to implementing components. For most of the reliability requirements there are such existing work which draw conclusions similar to the practical consequences of the reliability requirement. The reliability requirements also introduce new properties which components must satisfy in order to get reliable substitution.

The reliability requirements can be used to give theoretic support for arguing that following many of the previously published advice and practical solutions actually enhances OCS design. An interesting thing to note in this relation is that the conclusions from existing work are not drawn as the result of theoretic reasoning. Instead they are the results of decades of practical experience with developing object component systems. This similarity in conclusions from the presented theoretic work and from practical experience can be seen as an indication that the Omicron framework and the substitution proposition capture and formalise important properties of OCS design practises.

If it is assumed that important OCS properties are captured by the presented formalisation, it can also be assumed that those theoretic rules which do not correspond to existing work, also could be useful in practise. Therefore, this chapter also presents new advice for how to specify and implement components based on the reliability requirements. It can then be hoped that this advice can be used to enhance methods and tools so that they give better support to developers of object component systems.

The advice is related to making reliable specifications and reliable refinements. A reliable specification was defined in chapter 1 as follows:

> We define the term *reliable specification* to cover the class of component descriptions for which it is possible to make reliable refinements.

Reliable refinement was also defined in chapter 1:

> We also say that when $A \leq_D B$, ie, A is a refinement of B relative to D, and the refinement relation is reliable, then A is a *reliable refinement* of B.

The relation is reliable if the substitution proposition can be shown for the relation. The substitution proposition in the most simple form, says that if there is a system specification consisting of two parts, eg, B||D and:

> the component A is a refinement of B relative to D, ie, $A \leq_D B$ and
> the component C is a refinement of D relative to B, ie, $C \leq_B D$

then we want the systems A||C and A||D to have the same A-observable behaviour, and likewise, the systems A||C and B||C to have the same C-observable behaviour. This can be stated:

> A should have similar observable behaviour to B relative to C, ie, $A \leq_C B$ and
> C should have similar observable behaviour to D relative to A, ie, $C \leq_A D$

Then the system A||C will not have any unanticipated effects or erroneous functionality when compared to B||D.

Section 8.2 presents a new and important advice for making *reliable specifications*. The advice says that a reliable specification must define the maximum number of *visible* objects any reliable refinement can have. A specification must also define how the visible objects are used. This means describing which parameters in which observed messages can hold the names of the visible objects. This advice is a consequence of the requirements on reliable substitutions and the requirement that observably similar actions must have equal object names relative to a reliable substitution.

In section 8.3 each subsection restates one of the reliability requirements for *reliable refinements*. Instead of using Omicron concepts, the requirements are restated using the more common concepts of classes, objects and methods. This will hopefully help readers who are not experienced in mathematical formalisms and/or very simplified object-oriented prototype based languages. For the benefit of the less mathematically inclined readers, this chapter also gives some intuition on why the requirements are necessary. However, a proof for the requirements' necessity and sufficiency can not be given without the mathematical rigour of the previous chapters.

Section 8.3 also presents some new, not previously published, advice on how to make reliable refinements. These are: message selectors should not be parameters in messages to a component's context and a component should not compare the values of variables holding names of objects in the context.

Section 8.4 discusses the use of class names in object component systems. The conclusions are that classes as parameters in observed messages should be treated as visible objects, and class names in the code should be treated as possibly shared variables.

Section 8.5 discusses how to get reliable substitution in practise. The first subsection sums up the requirements for making reliable refinements, while the second subsection presents advice for making reliable specifications. Then comes a subsection discussing how to make correct specifications, as opposed to reliable specifications. The following subsection makes some comments on how to check that a component is a reliable refinement of a reliable specification. The conclusion from this section is that there is a lot to be done before there are practical tools for helping developers ensure that they have reliable substitution of components. However, results such as the reliability requirements and the definition of the reliable refinement relation, must be available before such tools can be made. The results of this thesis can therefore be viewed as a small, necessary first step in the process of making such tools.

Section 8.5 ends by summing up some of the most important lessons learned in relation to practical use of the results of this thesis.

8.2 Controlling Visible Object Names

The reliability requirements introduce new knowledge into the OCS component design. This section presents new and important advice for making reliable specifications and refinements.

The new advice presented in this section stems from the requirements which together say that it is necessary to control the number of visible objects in a component. This knowledge can be used in relation to designing and specifying object components.

The visible objects in a component are those objects in the component of which other components has knowledge, eg, pointers or references to.

8.2.1 Reliability and visible object names

The definition of observably similar actions says that when an action α from a possible refinement component is observably similar to an action β from a specification, then the object names in α are equal to the names in β relative to a reliable substitution. This is written: $\alpha = \beta\sigma$. This means that the object names in the two actions are either equal, or the substitution σ replace a name in β with the corresponding name in α.

The practical consequence of this is that a reliable refinement must have no more visible object names than the specification. In addition, the visible object names have to be introduced to the observing component by the observably similar message-send actions and in the same parameter positions. When making a reliable refinement, it must also be described how each of the visible objects in the specification is replaced by an object in the refinement.

Applying this requirement to the example of chapter 4, means that if MyModel is to be a reliable refinement of TextModel then MyModel can not have more visible objects than TextModel. However, MyModel may have fewer visible objects than TextModel.

This requirement might seem quite different - or even contrary to - how refinement in the form of more detailed implementations, are done. When making more detailed implementations it is common to add new collaborators to an object. However, it is quite safe to add new collaborators to the objects of a component when making a reliable refinement, but it is unreliable to make any new collaborator visible to the rest of the system. In other words: a reliable refinement can have more objects than its specification. However, the number of *visible* objects must not exceed that of the specification.

To support reliability, a reliable specification of a component must describe the maximum number of visible object names of the component and which are known to other components. A reliable specification must also describe how the visible object names are used in actions observable from the component's collaborators. For each component in the system, the specification must also describe which visible objects are initially known to the component's context.

The intuitive understanding of the requirement on reliable specifications, is that the maximum number of visible objects must be known so that a programmer can use the correct number of different variables to hold references to the different objects and thereby distinguish correctly between them. If a correct distinction is not made, it is not predictable which objects gets which messages.

One way of explaining why a reliable refinement must have no more visible objects than a specification is as follows: If a component has more visible objects than a given specification, there might be too few variables in a refinement of the component's context. When there are too few variables, the different visible objects are not distinguished. Then, when a component is inserted with more visible objects, this can cause errors when an object in the component gets a message some other objects was intended to get.

A similar way of explaining why a reliable refinement may have fewer visible object than a specification is as follows: If there are fewer visible object names from a refinement, there will maybe be variables in the context / context's refinement which hold the same object name. However, this does not create any problems in relation to distinguishing between the different visible objects.

An example of unreliability when the number of visible objects is not known:
Assume that there are more visible objects in MyModel than in TextModel. Relative to TextEditor, MyModel and TextModel have observably equal behaviour. When TextEditor collaborates with MyModel, TextEditor gets

to know of more visible objects from TextModel. We also have that NewTextEditor is a refinement of TextEditor and has similar observable behaviour as observable from TextModel.

The names of the visible objects are stored in slots in TextEditor and in NewTextEditor. May be NewTextEditor has the same number of slots to store visible object names in as there were visible object names in TextModel. Then, when MyModel starts sending more visible object names some of the slots are reused. How these slots are used is not specified by NewTextEditor's observable behaviour relative to TextEditor. It is therefore no way to show properties of NewTextEditor's behaviour when it collaborates with a component with more visible objects than TextModel.

Another simple, but extreme example of this is: assume that you have tested a component with a collaborator with a finite set of visible objects. Then you replace the collaborator with a new one with infinitely many visible objects. The component may then use up all available memory and the system stops. This is a different observable behaviour than the component displayed when collaborating with a collaborator with a finite set of visible objects. The difference in behaviour is caused by the new collaborator having more visible objects than the first one had.

The bottom line is that the number of and use of visible objects from a component must be specified and under control. If not, components may display unanticipated behaviour when combined.

8.2.2 Related work

This section presents related work on specification of visible objects. It first presents two patterns concerning the control of visible objects. Then the law of Demeter is presented, which says that each component should only have one visible object. Next, various software engineering methods are presented.

Facade and Mediator patterns:
The problem with components consisting of many visible objects has also been the focus for some of the patterns in the book on Design Patterns. As the visible objects requirement has not been known, the tackling of such problems have been given some simple solutions, namely reducing the number of visible objects to one. This is done in the Facade pattern.

The Mediator pattern is also concerned with visible objects. It describes how a component's dependency on a particular number of visible objects can be reduced by creating a single object which all other objects communicate through. The Mediator Pattern is described as follows:

> "Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently."

Keeping objects from referring to each other explicitly means that the objects' reliability on a particular number of visible objects is eliminated. In the Mediator pattern, the number of collaborators is reduced to a single visible object, the Mediator object.

The Mediator object is however dependent on a particular number of visible object, but this is easier to handle since a change in the number of visible objects will only result in the change of the Mediator component and all other components can stay unchanged.

The law of Demeter
The Law of Demeter is presented in various forms in (Lieberherr et al. 1988) and promoted as a law to be followed in order to achieve good object-oriented programming style which in turn gives easily maintainable systems. A brief but adequate description of the law is given in (Wirfs-Brock and Johnson 1990) as follows:

> One should not retrieve a part of an object and then perform an operation on that part, but should instead perform the operation on the object, which can implement the operation by delegating it to the part. The result of following the Law of Demeter is that a method depends only on the interfaces of its arguments and its instance variables, but it does not depend on their structure.

This law (particularly the object version in (Lieberherr et al. 1988)) controls and restricts the number of visible objects. The number of visible object names of a component is restricted to one, ie, the object mentioned in the definition of the law above. The object names visible to a method in a component is limited in the Law of Demeter. The object names visible to an executing method are the parameters to the method and the instance variables of the object the method is executing on.

(Lieberherr et al. 1988) also mentions three key constraints enforcing good programming style, ie, better maintainable and reusable code. These constraints require minimising code duplication, minimise the number of arguments passed to methods and minimising the number of methods per class. The two latter requirements are arguments for minimising the number of visible object names.

Software engineering methods:
There is a rich set of software engineering methods for developing object component systems. This includes object-oriented design methods such as Objectory, OOram, RDD, OMT, Catalysis and UML. The methods' component specifications are not used as a basis for formal reasoning about component behaviour. They are mainly used for *documenting* such behaviour. The notations can support intuitive thinking about OCS components, but not formal reasoning as presented in this thesis.

The different methods support the definition of visible objects to varying degrees. This means that they support the definition of reliable refinements to varying degrees.

Some development methods such as *Syntropy* and *Foundation* do not seriously address many problems related to reliable substitutions such as system maintenance, the design of substitutable components and the design of larger systems where the whole system can not be specified at ones. They present the development of a specification as a one to one mapping between objects in the specification to objects in the final system. The number of visible objects in an implementation is therefore given by the specification. The specification does not give any room for flexibility in the code. In this view the specification is not abstract. The method can therefore be seen as a method for detailed design and implementation of a system rather than one for abstract specification of programs.

The *Fusion* method has a concept called visibility, meaning that one object knows of another object. Such object visibility is documented by stating which objects are visible to which other objects and how the objects become visible to each other. As argued here, this is important for making reliable refinements. The method presents details on how to make an implementation of a given specification by a one-to-one mapping of objects in the specification to objects in the implementation. The specification of visible objects is therefore not presented as an important aspect for specifications of substitutable components, but just as part of a single program's specification. It is presented as the initial step in the implementation of a complete system. This may then also be seen as a concrete, rather than an abstract specification of a program.

Other object-oriented methods, eg, *Objectory* and *RDD*, introduce a concept called subsystem to denote a part of the specification which might be substituted with some refinement at some later stage. These methods assume that the system to be designed is so large that a single design model is not enough. Instead, the design is divided into several subsystems which are developed separately. A subsystem is therefore meant to be an abstract specification of a component in OCS terms.

In these methods a subsystem may be replaced by one or more objects in the final system. However, subsystems are specified by giving the type of the component, ie, a specification of the available methods in the subsystem and the types of the parameters. Since the subsystem specifications do not describe the number of visible objects and how these objects are presented to other components, the subsystem specifications are not reliable specifications. Therefore, the refinements of the subsystems can not be reliable in our sense. This means that the different parts of the system can not be separately developed while at the same time knowing that the combined subsystems will behave as specified. To make separate development possible, the methods should include specifications of visible objects in their notations. The methods must also include support for making reliable refinements as discussed in section 8.5.

(Jacobson et al. 1995) has taken the Objectory notation of subsystems a step further in the direction of dividing a subsystem specification into different parts. In this paper the set of messages a component can receive is divided into separate contracts (figure 6 in (Jacobson et al. 1995)). However, the contracts do no describe objects, but types of objects. So the introduced notation defines how many types of visible objects there are in a component, but does not define the number of visible objects. However, with a supplementing view of the participating objects (figure 3 in the article) it is possible to deduce how many visible objects there are. However, the method does not give any attention to definition of visible objects. In addition to defining visible objets, the notation should be able to describe how object names are passed between the components and thereby become visible. This is also lacking in Objectory. Work on Objectory has now continued in the joint UML effort and the ideas presented in the cited paper has been worked into UML. It should be quite simple to adjust some of the UML and Objectory diagrams to incorporate the specification of visible objects and how they are handled. This would allow Objectory and UML to support the creation of reliable specifications.

The *OOram* method has given attention to how large system descriptions can be split into smaller descriptions and then how such small descriptions can be merged to create large system descriptions. Their solution is a technique denoted synthesis. The synthesis technique is based on OOram role models where each role represent

one object or a set of objects which will observe and display the same behaviour. A role model gives a complete picture of the behaviour of the objects in the roles in the role model. The role model describe all actions which these objects observe from each other. Also, a role model describe the visible objects which an object in a role must know of and which reliable refinements of the role must be able to distinguish between. Large system descriptions are created by assembling role models though the synthesis technique. A synthesis of two role models means merging one or more roles in one role model with one or more roles in the other role model.

Details of the synthesis technique is documented in the OOram book and also discussed in (Andersen and Reenskaug 1992) and (Andersen 1997). What is denoted safe synthesis is an operation corresponding to reliable substitution, as is explained in chapter 1 of this thesis. Andersen uses LTS (Labelled Transition Systems) with variations, to describe objects. Most of this work is focused on Ensuring safe synthesis operations with respect to avoiding interaction errors. Interaction errors are message-not-understood errors and illegal termination errors where an object has entered into a deadlock or a livelock. With Andersen's approach to modelling objects it is difficult to reason about liveness properties and reliable substitution as is done in the present thesis. As noted by Andersen in his thesis, the requirements for safe synthesis defined in his thesis are therefore only necessary, but not sufficient requirements for safe synthesis. As shown in the present thesis, it is also necessary to make requirements corresponding to the presented reliability requirements of chapter 5 to achieve safe synthesis. Further work is necessary to merge the reliability requirements with role modelling and synthesis, and also to see if these requirements are sufficient to ensure safe synthesis.

Catalysis (D'Souza and Wills 1995) is different to most object-oriented software engineering methods in that it has some reasoning power. It allows precise description of behaviour. However, at the time this text here is written, behaviour does not include message sending and instance creation behaviour, but only describe how data is handled. Typical examples of types of objects specified using (the formal parts of) Catalysis are stacks and lists and their operations such as push(), pop() and addFirst(), removeLast(). The behaviours are specified using pre- and post-conditions written in predicate logic. Pre- and post-conditions can refer to methods in the specifications, thus the specifications become quite similar to algebraic definitions of types (Abstract Data Types). The variables used in the specifications are seen as abstract in that they need not be found in implementations. Catalysis introduces the idea of a retrieval to describe how a specified behaviour is implemented by a given class. There are some tools to help in verifying that a class implements a type as described in a retrieval. Catalysis also define *refinement*. They define it as follows on page 11 in (D'Souza and Wills 1997):

> Refinement means creating or choosing a conforment type or class, and documenting why you believe it is conforment. ... Documenting a refinement means writing down:
>
> The reasons you believe the supposed abstraction really does describe the implementation accurately and
>
> the reasons for choosing this implementation from alternatives.

Catalysis refinements are therefore informal definitions of similarity. On page 13 they further write:

> Refinements and retrievals give a precise basis for traceability (aiding understanding, reverse engineering, and system maintenance and extensions), and substitutability (aiding team-work and parallel development). In addition, they solve a very real problem in software development today, particularly in light of the move towards iterative development and incremental delivery.

As this shows, Catalysis addresses many of the same problems as this thesis. The three main differences being that 1) Catalysis has a user-friendly notation and language, while Omicron is created for formal reasoning and therefore user-friendliness was sacrificed 2) Catalysis is able to reason about how classes of objects manipulate data while the work in this thesis reasons about how components send messages and create objects and 3) Catalysis is aimed at specifying the behaviour of an object of a given type while the substitution proposition is aimed at specifications of the behaviour of components consisting of one or more objects. Catalysis also has many features not covered in this thesis, like abstract behaviour specifications which are too abstract in the sense that they are far to vague to be reliable specifications. Even with these differences, Catalysis is the method which, at present, has most similarities to the present work. The creators of Catalysis is now incorporating their ideas into UML.

8.2.3 Further work

A practical useful topic to pursue, is how to enhance software engineering methods with processes and notations which help developers select and specify the visible objects of components. This would help developers of large systems and reusable components since it allows them to make reliable specifications. This in turn paves the way for reliable substitution which again can lead to less problems with component integration and system maintenance.

The understanding of the importance of the number of visible objects may also help in the development and/or description of particular patterns and other designs for reusable components. When "the number of visible objects" becomes part of the description, it might not be necessary to resort to simpler solutions such as limiting the number of visible objects to one. However, limiting the number of visible objects is important to make a component description more comprehensible for human readers.

8.2.4 So why not only have one visible object ?

The number of visible object names indicates to what extent a component's functionality may be seen as encapsulated. Many visible object names may be seen as poor encapsulation while few visible object name may be seen as good encapsulation of a components functionality.

The different informal component design rules cited above agree that there are advantages of minimising the number of visible objects. However, having a small number of visible objects means that the handling of incoming messages can only be delegated to a small number of visible objects. This might lead to less comprehensible designs of refinements. The refinement becomes less comprehensive when a larger numbers of messages are sent to a few visible objects and/or there are larger numbers of parameters in the messages. This is related to the problem outlined in (Liskov and Guttag 1986) which states that there can be too many operations on a type, ie, too many messages to an object. This problem is also pointed out as a disadvantage of the Law of Demeter in (Lieberherr et al. 1988).

Therefore, if having more than one, or a few, visible objects makes a specification easier to comprehend and/or gives simpler implementation and design of refinements, then more visible objects should be used. One example of this is a design where there are many similar objects, ie, many objects in the same role as seen relative to the collaborating component. If a single visible object should be used while it is necessary to distinguish between the objects, some kind of parameter value must be used to distinguish between the objects. This would give a more complex design. Also, the parameter values might just be representatives for the object names. Then these names must be carefully handled[11] by the programmer him/herself. This might cause a more time-consuming and error prone implementation process, one reason being that the run time system's support for handling object names catch some types of errors and such errors are not detected by the run time system when parameter values are used to distinguish between objects and these values are handled by the programmer herself.

Another example is when, in some cases, one wants to view the visible objects as components themselves. One example is the text editor design presented previously in chapter 4. The TextEditor component can sometimes be seen as consisting of the two objects: a view and a controller. Viewing the TextEditor as a single component can be useful when replacing the TextEditor. Viewing it as two components can be useful, eg, when the underlying software related to views change while things related to controller like menus, mouse and keyboard handling remain the same. In this case only the view component needs to be changed. In this case, it is then a good idea to let the TextEditor have two visible objects; the view and the controller.

Both (Johnson and Foot 1988), (Gamma et al. 1994) and others give similar arguments for splitting a component into many objects and argues that this gives more flexible and reusable code. At the same time, the same authors in (Gamma et al. 1994) give patterns for avoiding problems related to the number of visible objects. This reflects the fact that there are trade-offs between specification complexities and refinement flexibility related to the number of visible objects of components. Such trade-offs is an interesting topic to pursue, but is beyond the scope of this thesis. Presumably this is a less theoretically and more practically oriented topic as humans' abilities to cope with complex descriptions will influence decisions maybe even more than theoretic limitations.

[11] Presumably these parameter values must be handled as object names in order to get reliability, ie, the number of different parameter values is limited by the specification in the same way as the number of visible object names are limited.

8.3 Practical Implications

Each subsection below presents one of the reliability requirements from chapter 5 and relate them to practice in object component system design. In each subsection a conclusion is drawn. The conclusions give advice on how to make reliable specifications and reliable refinements. Section 8.5 will sum up and elaborate on the advice related to making reliable specifications and reliable refinements.

8.3.1 No external inheritance

In Omicron, objects share variables by inheriting from other objects. This is similar to how variable sharing is done in SELF. The reliability requirement "no external inheritance" says that no object in a component which is a reliable refinement can inherit from an object outside the component. This means that a reliable refinement can not share variables with its context.

Reliability properties ensure that no unanticipated errors may occur when a component is substituted with a reliable refinement component. In relation to inherited variables, unanticipated errors may occur when

- a context refers to variables inherited from, and then shared with, objects outside the context, ie, variables in the component is inherited by the context, and
- these variables are not part of the component's specification and/or
- the refinement component does not have variables corresponding to the ones referenced by the context

By the way "similar components" is defined, a specification only defines a component's observable behaviours. Therefore, variables are not part of a component specification and it is not explicitly checked if a refinement has variables corresponding to the ones referenced by the context. Therefore, there are no guarantee that the shared variables are found in a refinement. The conclusion is that a reliable substitution of components holding shared variables is not possible.

If one wants objects to be able to share variables with objects in other components, and at the same time have reliable substitution, it is necessary to specify which variables a component is expected to have. Then more than the components observable behaviour is specified. It must also be ensured that each reliable refinement of the specification contains the specified variables

Specifying both variables and the observable behaviour of a component will complicate the specification. This added complexity reduce the readability of a specification.

Requiring that a reliable refinement contains certain variables puts obligations on the implementers of the components. The developer must implement the components' behaviours by using the specified variables in specified ways. Such obligations reduce reusability since the implementers gets less freedom to choose implementation strategies.

Related work

The drawbacks with using shared variables have been used as an argument for introducing object-oriented techniques such as encapsulation of variables. Examples of such arguments are found below.

Designers of object component systems take the consequences of the problems with external inheritance by hiding shared variables (often denoted state) behind the message interfaces of components instead of accessing variables directly. This is widely applied to object-oriented designs and implementations and is termed encapsulation which (Blair et al. 1991) defines as follows:

> *p. 60: encapsulation : the protection of state behind a procedural interface*
>
> *p.13: Encapsulation is used as a generic term for techniques which realise data abstraction.*
> *Encapsulation therefore implies the provision of mechanisms to support both modularity and information hiding...*

The Smalltalk language is defined with strict encapsulation of objects so that it is impossible for one object to access the variables of another object. In languages such as Java, C++ and Simula, objects are allowed to access other objects' (public) variables and there is a clear distinction between calling a procedure (or function) and

accessing a variable. This gives unreliable configurations if not the access of variables of objects in other components are added as part of the specification of a component's observable behaviour.

(Bollay 1992a) gives guidance on how to create reusable CLOS code and presents a number of rules. The third reuse rule says:

> *Do not imbed explicit paths to objects throughout your code. Write an accessor and use it. This will make your code independent of the structure of relationships between objects.*

This is a warning against directly accessing variables if the goal is substitutable components and corresponds to the no external inheritance requirement. Encapsulation features have also been added to SELF and its use is recommended in (Chambers et al. 1991). Dylan also has extra encapsulation features as compared with CLOS (Bollay 1992b).

Abstract Data Types (Guttag 1980) is also a technique for hiding variables behind an interface and has inspired many later approaches. This is done to achieve implementation independent specifications and to support reliable substitution under certain circumstances, eg, when objects are collections of functions. The difference between Abstract Data Types and Omicron is discussed in the chapter on related formal work.

Instead of accessing variables, the general advice from component developers is that components should share data by sending messages. Variable access should be replaced by what would be two message-send actions in Omicron: one message to ask for data and an other message which return data. Both actions will form part of the components' observable behaviour.

In Beta an object may also access the variables of another object. However, Beta has its own way of reducing problems with variable access. In Beta variables can be declared as virtual and then accessing the virtual variable becomes the same as sending a message and returning a value.

The CLOS reusability rules in (Bollay 1992a) include the rules:

> *Reuse Rule 1 - Always use accessors to woze[12] information*
> *Reuse Rule 2 - Separate state information (slots) from class operations (methods)*

It is argued that the definitions of methods and the definitions of variables should be separated. Then the class operations (methods) become independent of the state information (slots or variables). This means that the variables are hidden behind an interface, ie, encapsulated, even as viewed from the object's methods. This is a further step in the direction of limiting dependencies made possible by external inheritance. In this case, the methods and the object are viewed as separate components and the methods should not access the object's variables, but send messages instead.

Conclusions

If a component is to be a reliable refinement of some specification, the no external inheritance reliability requirement limits the use of shared variables to sharing variables within a component. Shared variables between components should be avoided. Instead, when a component wants some data from a collaborator it should send a message which returns with the data.

8.3.2 Reliable method lookup

In Omicron, an object has a set of slots which are similar to traditional variables and method tables. A slot can therefore refer to a method. A method is selected for execution when an object receives a message. The selected method is the object referred to by the slot with the same name as the message selector. The slot may be found in the object itself or in some object the receiver inherits from. Because of the requirement "no external inheritance", an object in a reliable refinement component will never inherit from an object outside the component. Therefore, a slot will always be found in the receiver's component when looking up a method. However, a slot may refer to an object in another component. The selected method may therefore be part of a different component than the receiver of the message. The "reliable method lookup" requirement says that a reliable refinement component can not have methods which are found in the component's context.

Objects holding methods are often thought of as playing the role of classes, and we therefore refer to such objects as classes in the below discussion. The reliable method lookup requirement then says that *an object must be in the same component as its class(es).*

---

[12] Woze assumably means something like "looking at" or "accessing". Woze is not found in Webster's Dictionary or other dictionaries which have been consulted.

When an object and its class(es) must be in the same component, it is not possible or meaningful to let a class or set of classes by a reliably substitutable component. This can be explained as follows:

If classes are components in themselves and we want reliable substitution, we can have one of the following cases:

- the class component is not in use
  in the sense that none of the objects in the component's context use the class' methods or
- there are no reliable refinements of the class component's context components
  since reliable refinements of the context can not use methods in the class

One conclusion is then that a class is not a typical OCS component in that

- either it is not in use or
- the class' context can not be substituted with reliable refinements.

The context can not be substituted with reliable refinements because there can be no reliable refinements which use the methods in the class. It they did, they would not be reliable refinements.

When an object is in the same component as its class(es), a change in a class may result in new observable behaviour. Therefore, a component may not have the same observable behaviour before and after a class change. It is therefore necessary to check that a component still is a refinement of its specification after one or more classes are changed.

If classes are to be substitutable components it is necessary to give quite complex specifications of the classes in order to document exactly what may be changed without making the objects behave differently. The complexity stem from the necessity to include more than the components' observable behaviour in the documentation. With complex documentation it becomes impractical to use such components since the documentation becomes difficult to understand.

Related work

Static classes:
Reliable method lookup is always found in systems created from programs in class-based languages such as Java, C++ and Simula. In such languages classes are static during runtime. The class description can therefore be seen as copied into each component and then a method is automatically found in the same component as the receiving object.

A general rule when making CLOS-programs (Keene 1989) is to separate objects playing the role of classes and objects having a traditional object-role in the system. The differences in the roles are that the class-objects are only changed during the development phase of the software while object-objects are also changed during runtime. The class-objects hold methods while the object-objects hold variables which change during runtime. This is in accordance with what is shown in section 6.6 and is also recommended for SELF programs (Ungar et al. 1991).

The Dylan language resembles CLOS in many ways, among other things it is a prototype based language where objects are templates for object creation. Dylan have several additional features which give developers more control over the systems they are making and thus create systems with less errors. One such feature allows the developer to declare certain objects as read only (Bollay 1992b). This feature is typically used in relation to objects functioning as templates for other objects, ie, they function as classes. Read-only keeps the objects definition from being changed, Therefore, declaring template objects as read-only makes them as static as classes in C++ and Simula. This helps in Ensuring that the class objects are not changed during run time, thus giving more reliable components by fulfilling the no external inheritance and reliable method lookup requirements.

Block objects
Inserting new methods into existing classes during runtime is possible in, eg, Smalltalk. This might give unreliable components. In Smalltalk systems, this functionality is only to a very limited extent used. Instead, the standard way to insert new functionality or vary functionality in existing components is by sending block-objects as parameters. A block object holds a piece of code which is executed when the block gets a message named value (or value:, value:value: etc). A block object may be viewed as a function which is evaluated when it receives a value message.

Block-objects are then passed from a component to its context, or vice versa. The sending of the value-message is observed and must therefore be part of the specification. Sending observed messages is reliable. Therefore, using block objects instead of changing methods give reliable components, while preserving flexibility in object behaviours. Lately, a version of block-objects is included in Java.

An example of the use of block-objects is the sort-method of the Smalltalk classes for arrays and collections. The sort method takes a block as a parameter for calculating the sorting order of two objects. An alternative to

using the block would be to let the programmer provide the array object with an appropriate method for finding the sorting order. The new method could either be provided by making a subclass of the old class - which means that the new version of the array-component must be tested, or by changing a method of the old class during runtime - which is unreliable.

The idea of creating flexible code by creating an object such as a block has been further developed in the book on Design Patterns. In the pattern denoted Visitor it is recommended to create an object representing an operation to be performed on the elements of an object structure. In the Visitor pattern description it says: "Visitor lets you define a new operation without changing the classes of the elements on which it operates.". The Visitor pattern is useful in situations where it is necessary to add new code which will naturally fit in as methods in existing classes, but creating methods is perceived as bad design and feels error prone and insecure. The reliable method lookup requirement gives theoretical support for the Visitor pattern and shows that the feeling of insecurity related to changing methods in existing classes is correct. (There are also many other good reasons for the Visitor pattern which is not relevant to this point).

Framework design
The "reliable methods lookup" requirement can be applied to Framework design. A Framework defines the collaboration between a set of components. Usually, also a set of classes are provided. The classes can either be instantiated to create objects in the components or the classes can be subclassed to tailor the behaviour of components to the needs of individual applications. A Framework where the classes are mainly meant for instantiation is called a component-based framework, or a black box framework. "Black box" indicates that the user of the framework needs not know of the internals of the components. Instead the components are treated as black boxes. A Framework where the classes are intended for subclassing is called an inheritance based framework, or white box frameworks.

(Johnson and Foot 1988) argues for using components by their observable behaviour by using and creating component-based frameworks, rather than inheritance based framework. Inheritance based frameworks include specifications of more than the components' observable behaviour and are therefore much more complex and difficult to use.

Inheritance vs. Decomposition
(Johnson and Foot 1988) present the concept of "Inheritance vs. Decomposition". Decomposition means making software components which are used as black boxes and their functionality is available through a set of messages. The components' functionalities are specified by their observable behaviours. Inheritance, on the other hand, means making software components which are meant to be used as superclasses. This means that classes are viewed as components. Subclasses are then created which inherit behaviour and override methods, thus accessing the same variables as methods in the superclass and changing the methods in the superclass.

Both the "no external inheritance" and the "reliable method lookup" requirements can be used as arguments for preferring decomposition, since a subclass will break one or both of these requirements. In the paper (Johnson and Foot 1988), it is also argued that decomposition is better than inheritance. They write:

> *Since inheritance is so powerful, it is often overused. Frequently a class is made a subclass of another when it should have had an instance variable of the class as a component… Behaviour can be easier to reuse as a component than by inheriting it.*

Conclusion

Give priority to delegating functionality to a component rather than inheriting it from a class.

It is not possible or meaningful to have superclasses as components and at the same time have reliable substitution of such components.

If a class is changed by code in another component or by a programmer, the component with the class in it and all components which have subclasses of the class must be retested to establish if they still are refinements. One consequence of this is that it is best that methods are not changed by code in other components than the components holding the methods. Use such things as Smalltalk block-objects and/or the Visitor pattern when you want a component which should have different observable behaviour depending on which component it collaborates with.

If a class is changed by code within the component where the class is found, this can still give a reliable component. This means that it is *not* unreliable to let objects within a component change the component's own classes during runtime. However, even if a component has reliable behaviour when changing its own classes, there might be an argument against this: a human may have problems understanding and managing code which change or is changed by other code.

8.3.3 Reliable message selectors as parameters

The practical result of the definition of observably similar actions, where message parameters must be equal relative to a substitution, is that object names which are used as parameters in observably similar messages must be equal relative to a substitution. A consequence of this was the requirement on the number of visible object names in a refinement. In this section we look at the consequence of the definition of observably similar actions for message selectors when used as parameters.

To get reliable substitution of component, a substitution can only substitute object names with object names. In addition, names which are used as object names, must not be used as slot names. Therefore, slot names will not be substituted by the substitution, and must therefore be equal in parameters of observably similar actions. Message selectors are slot names and therefore message selectors must be equal when they are found as parameters in observably similar actions.

Message selectors as parameters are used to make some of the most used components in Smalltalk; the pluggable editors. This is due to a special feature in Smalltalk which is also found in later versions of Java. In Smalltalk, message selectors may be parameters to messages and stored in variables. Some basic primitive method; the perform:-method makes it possible to use the value of a variable as a message selector. If the variable *m* holds a selector, the expression *o perform: m* will send a message with the value of *m* as selector to the object referred to by *o*. The perform: feature of Smalltalk has been used to create some very use-friendly components such as, eg, the Smalltalk text editor and a view for presenting lists - the so called pluggable editors.

As mentioned, if message selectors can be parameters and vary in observably similar actions, the receiver will not be a reliable refinement. Therefore, the perform: feature of Smalltalk makes it possible to create unreliable components since it allows message selectors to be passed as parameters and stored in variables.

To illustrate why this creates a reliability problem, we can take a situation where we have a component TextModel, a context TextEditor and a reliable refinement of the context called NewTextEditor. The context and its reliable refinement send similar messages to the component TextModel. The messages are observably similar as defined in chapter 5 except that message selectors as parameters may be different. Then it is impossible to show that TextModel will have similar observable behaviour when collaborating with TextEditor and NewTextEditor. This is because we do not know how TextModel uses the message selector which is received as parameters. For example, TextEditor may send the selector *s* as a parameter and then get two messages back where both of them have *s* as selector. NewTextEditor may then send selector *t* as parameter and then also get two messages back. The two messages can either both have selectors *t* or both *s* or one message have *s* and the other *t* or vice versa. Therefore, if message selectors can differ when they are used as parameters, the receiving component may behave quite differently.

Conclusion

Parameters which are message selectors must be equal in observably equal actions. Therefore, having message selectors as parameters in observed messages is wasted effort since the parameters are equal in all cases.

As noted above, some very useful components have been developed by using message selectors as parameters, namely the pluggable editors of Smalltalk. Allowing message selectors as parameters adds flexibility in how components collaborate. The unreliability in the message selectors as parameters and the usefulness of the pluggable editors presents a contradiction between usefulness and reliability. The problem with unreliability of refinements of pluggable editors is, however, evident in the way pluggable editors are designed, coded and tested: The pluggable component developers are very conscious about how the slot-name variables and parameters are handled. Also, cautious tests of the components are done before they are distributed and used.

The positive experience from creating and using pluggable editors was the motivation behind having a common name space for objects and slots in the Omicron language. The conclusion that this homogenisation gives unreliability supports the experience that special care must be taken when building pluggable editors so that the use of slot-name variables is controlled.

Further study of how to create reliable pluggable components

Chapter 5, section 5.4.2 presented an alternative to the above restricted view, where message selectors must be equal. This alternative allowed message selectors to vary when used as parameters and it seemed that this still gave reliable substitution. The drawback with the alternative solution, and the reason it was not developed any further, was that it gave much more complex requirements on reliable refinements. For the theoretic work this gives much more complex proofs and in practice it means larger and more detailed specifications. However, the drawback with the simpler solution was that flexible components like pluggable editors are not viewed as reliable. This is an example of the fact that there is a dilemma between simple specifications and flexible

components. The study of the trade-off between the simplicity of specifications and flexibility of components would be an interesting topic to pursue.

8.3.4 Reliable if-sentences

Omicron and other object-oriented languages allow object names to be compared and the behaviour of the objects to depend on the result of the comparison. In this way it is possible to vary the behaviour of an object depending on whether or not two variables refer to the same object. If such comparisons are found in a component and the component is to be reliable there must be restrictions on the names being compared. As shown in chapter 5, there are two alternative ways of Ensuring reliability of object name comparisons (if-sentences):

1) Either there is an equal number of visible objects from a specification and its refinement
2) or the compared values are never both names of objects outside the component

Related work

Alternative 1) means that a specification and all the specifications refinements must have an equal number of visible objects. This restriction is contradictory to many practices where several objects may be substituted with a single object which is able to behave similarly to all the objects it replaces. This is sometimes done in the MVC-Framework when a single object takes the place of both a view and a controller object. This is done in order to limit the complexity (number of classes) and volume (number of objects) of a system.

Alternative 2) means that if-sentences are only used to compare object names of objects which are found inside the component. By examining the Smalltalk-libraries it is evident that object names are rarely compared, except to see if a variable is initialised or not. This supports a view that in reusable components one rarely compare object names. Instead of using if-sentences, the designer and programmers use messages and dynamic binding to vary the systems functionality depending on the systems state.

When if-sentences are used for other purposes than checking if variables are initialised[13], reliability of if-sentences is ensured by the way if-sentences are implemented. In Smalltalk and also in SELF if-sentences are only found in a method belonging to the object itself. The method tests if the receiver is the same object as the parameter to the message. The method is named '==' and can be defined using the Omicron if-sentence in place of the equivalent Smalltalk-primitive as follows:

```
== otherObject
 | bool |
        bool := (self = otherObject true false).
        ^bool
```

This ensures that at least one of the object names is local to the component as the pseudo variable 'self' always refers to the executing object and this object will always be an object inside the component. Therefore, reliability is ensured.

In languages where if-sentences may be placed in all parts of the system, one strategy to ensure reliable if-sentences is to type variables as local or external to the component where it is found. Then the type information can be used to check that only variables holding object names local to the component are used in if-sentences. This will ensure reliability.

Conclusion

Comparing object names should be avoided unless refinements are expected to have exactly the same number of visible objects as defined in the specification. Instead of using if-sentences it is better to redesign and use dynamic binding. If object names must be tested, at least one of the names must be local to the component where the test is done.

8.3.5 Reliable message sending

The reliable message sending requirement concerns errors when trying to send messages from a refinement configuration to a specification configuration. Using the text editor example to illustrate this, the requirement

[13] Initialization test check if variables refer to objects. This is usually done by testing the variable value with the constant representing no object, typically called NULL, none or nil. Such names are, in relation to Omicron, treated as any name and reliability requirements apply to such names in the same way they apply to a name which is not the name of an object.

concerns the situations where MyModel is trying to send messages to TextEditor. The requirement says that there must never be a situation where an error occurs because of "message not understood" in TextEditor as a result of a message from MyModel. In other words, a method must always be found in TextEditor for the messages sent from MyModel.

The reliable message sending requirement was necessary to ensure that a new context would not get a message from a refinement component which the old context did not get. If the old context did not have a method for a message from the refinement component, the refinement component would just have a non-observed error action. Now, if the new context had a method for this message, the result would be an observed message-send action. Such a message actions would be unexpected behaviour from the refinement and therefore the refinement doesn't have reliable behaviour.

As discussed in chapter 5, another error model and new definitions of observability and similarity of such error actions can be done in a way which eliminates the need for explicitly requiring reliable message sending. However, this alternative gives more complex definitions and also has consequences on how to make specifications and refinements which seems counter intuitive to common practice. Instead, it is quite common in practice that developers require that components meet the reliable message sending requirement. Type checking and/or type inference is usually employed to ensure such properties.

Related work: Type checking and type inference

Traditional checking that components are type safe can be used to eliminate message not understood errors. This is very useful in a sequential and/or deterministic system. However, if the system is parallel or non-deterministic traditional type checking/inference might find errors while the component still has reliable message sending. Below it is argued that traditional compile time type checking/inference with signature subtypes (as defined in, eg, (Wegner and Zdonick 1988)) is in some cases in conflict with the refinement relation in that a component may be a refinement but not a subtype of its specification.

There is an enormous amount of work being done on proving type safety of object expressions, eg, (Cardelli and Wegner 1985), (Graver 1990), (Ågesen et al. 1993), (Palsberg and Schwartzback 1994), (Abadi and Cardelli 1994) and (Bruce et al. 1997) comparing the different object encodings. Their focus is on making strongly typed languages allowing polymorphic types to be type safe languages, ie, no calling of undefined functions (methods). There is also work done on using type inference to ensure type safety. (Palsberg and Schwartzback 1994) give a good presentation of the various aspects of typing object-oriented programs in general and type inference in particular.

The conflict between subtype and refinement is due to the fact that a specification may specify alternative observable behaviours. Therefore, a refinement may only display some of the behaviour defined in the specification and then only get a subset of the messages the specification gets. In this way a refinement might get fewer messages than a specification. Then the refinement may have fewer methods than the specification without being erroneous. In type checking / inference terms this means that a specification may be a (signature) subtype of a refinement and the refinement may be a (signature) supertype of the specification.

Therefore, requiring that a refinement must be a subtype of a specification can lead to refinements being rejected by the subtype-test while they would pass the refinement test.

If a specification did not include alternative behaviours, a (signature) subtype relation would be a necessary but not a sufficient requirement for the refinement relation.

If a refinement was found to be a *supertype* of a specification, the reliability requirement "reliable message sending" would not be necessary. This can be examples as follows:

The reason reliable message sending was introduced was to avoid the following problem (we use the text editor example again):

> TextEditor and TextModel are specifications. MyModel is a refinement of TextModel and NewTextEditor is a refinement of TextEditor. MyModel tries to send a message to an object in TextEditor. The selector of the message does not match with a slot name in the inheritance graph of the receiver. The result is therefore an error action which is not observable from TextEditor.

Why a supertype would give reliable message sending in this situation can be explained informally as follows:

> If every method in NewTextEditor is also found in TextEditor then NewTextEditor is a supertype of TextEditor. This ensures that if an error occurs because a method is not found in TextEditor, then a similar error will occur for NewTextEditor and reliability is ensured.

However, if a refinement is a subtype of a specification, there might be reliability problems. An example:

> If there are methods in NewTextEditor which are missing in TextEditor then we have that NewTextEditor is a subtype of TextEditor. This will create reliability problems since: The missing methods can create error-actions which are *not* observable from the TextEditor in the configuration consisting of TextEditor and MyModel. When NewTextEditor and MyModel are combined then these errors can disappear and instead be message sends to NewTextEditor. Such messages might not be messages which are sent from TextModel to NewTextEditor and the behaviour of the MyModel||NewTextEditor system is therefore unknown.

This shows that a refinement being a signature subtype of a specification may be unreliable, while a refinement which is a supertype of a specification eliminates the need for checking reliable message sending.

On the basis of the work done by Palsberg and others (Palsberg and Schwartzback 1994) on compile time type checking/inference for languages similar to Omicron, it can be assumed that compile time type checking/inference can be developed for specification languages such as Omicron. Type inference can then be used to check that a refinement is a supertype of a specification. This will ensure reliable message sending in a specification configuration.

One way of finding out if a component has reliable message sending is by executing the component together with the specification of the context. This can be done provided that all message not understood errors are observable from the component developer. This will work because a message not understood errors will occur when the reliable message-send requirement is not met. Then, when an unreliable message-send action is observed, the developer can take appropriate action: either change the refinement and remove the message send which created the error or change the context specification so that it includes a method for the message.

Conclusion

Above it was shown that the traditional type safety is different from the reliable message sending requirement. However, techniques used for typing, and subtyping, can be used to ensure reliable message sending.

8.4 Use of Classes in Reliable Code

8.4.1 An example

What follows is an example of the use of class names in the code of a Framework which allowed subclassing and substitution of components. The Framework initially seemed to be very nice for developing a system by subclassing and substituting components. However, unexpected errors occurred so the components were obviously not reliable. Even though the example is relatively old, there are reasons to believe that similar problems can be encountered in more resent Frameworks with components of similar complexity and flexibility. The unreliabilities were related to:

- a particular class name was explicitly used in the code when objects were created
- the particular class was also available for subclassing, thus objects of this class were seen as substitutable components
- instance creation was not considered part of a component's observable behaviour and therefore not part of a component's specification

The example is taken from a project at the Centre for Industrial Research in Oslo in 1984. The project was to enhance a text editor which was part of a Framework for the Apple Lisa computer. The change to the editor was done to enable it to recognise selections of special words so that these words could not be edited at the character level, only inserted or removed as whole units.

The original text editor implementation was done in Object Pascal and was delivered as compiled code. However, most of the classes could be subclassed and objects of a class could then be substituted by objects of a subclass. In this way reusability and flexibility was supported, and this was very useful to the users of the Framework.

The Framework had a class called TTextSelection. Objects of this class handled the selection of parts of the text on the screen. This class was provided for subclassing. The change to make the editor recognise and select special words was therefore simple to do by subclassing the TTextSelection-class. It worked rather well except that on some occasions the user was suddenly allowed to change the characters of the special words.

After some debugging it was discovered that in some cases, an object of the new text selection class was replaced with an object of TTextSelection. In such cases the user was allowed to edit the characters of the special words. When the new text selection class was developed it was observed that new text selection objects were in general not created by the Apple's text editor component, but created by the new code of the project. However, on close inspection of the running program it was discovered that code in the text editor component created a new object. This new object was of class TTextSelection and this object replaced the old object of the new text selection class. The text editor code obviously explicitly used the TTextSelection class to create new objects.

The error this created in our new version of the text editor was very difficult to correct, particularly since Apple's text editor was only delivered as compiled code. The problem was, however, eliminated as the Apple Lisa project was discontinued at Apple and the text editor development project in Oslo moved over to a Smalltalk-80 system.

The conclusion from the project was that the Apple Framework had a lot of good intentions when it comes to flexibility for extensions and substitutions. However, the intentions were not fulfilled by the specification and implementation of the various parts of the Framework since they lacked important properties. At the time the project was ended, it was not clear what these properties were.

The above experiences with the Apple Framework was one of the motivations for the work in this thesis. As a result of the thesis it is now possible to conclude what important properties were lacking from the specification and implementation of Apple's text editor Framework: the developers of the Framework should have treated the TTextSelection class according to the requirements for making reliable specifications and refinements. Here are some details of what they should have done:

First of all, since it should be possible to replaces the TTextSelection class with a new class, the TTextSelection class and the objects of this class should be described as part of the context of the text editor component.

Above it is argued that shared variables can not be part of reliably substitutable components. The text editor contained direct references to the TTextSelection class. This is equal to referring to a shared variable which is found in the text editor's context. This means that since TTextSelection is explicitly mentioned in the text editor, the context component with TTextSelection can not be reliably substituted.

Instead of explicitly naming TTextSelection in the text editor code, the TTextSelection class could be treated as a visible object from the context known to the text editor. Also, the creation of new text selection objects should be part of the observable, and therefore documented, behaviour of the text editor.

Another solution would be to define a message to be sent to the text selection component whenever a text selection object was to be created. Then the TTextSelection class would not be visible and creation of text selection objects would not be part of the text editor's observable behaviour. Instead, the specification of the collaboration between the text editor and the text selection components becomes simpler. In addition, the developers of the text selection component get full control of text selection object creation.

Below, related work shows that many of these lessons have been learned by other practitioners who have experienced similar problems when trying to use and develop Frameworks.

8.4.2 Related work

Warnings against spreading class names in the code have been given by several tutorials held at different conferences, eg, the CLOS-tutorial at TOOLS '89, OORASS-tutorial at OOPSLA/ECOOP 1990 and Ralph Johnson's tutorial at OOPSLA '91. Class names are spread in the code of programs for three different purposes as classes are used in the following ways:

- as templates for object creation
- in relation to testing the class of an object
- for typing variables in compile time type checked languages

The following discussion is divided into these three topics.

Class names as templates for object creation

Examples of existing component designs which have more control of the use of class names is found in the Smalltalk-80 libraries. The trick used to control the use of class names, is that these names are not used directly. Instead, there are many statements of the form:

```
obj species new
```

where obj is a variable pointing to an object, 'species' is a message to the object which returns the object's class (the class can be returned as the classes are objects in Smalltalk). 'new' is then a message to the class object which return an object of the class. This is, eg, used when making general code in superclasses. In this way the code also becomes applicable to objects of subclasses. If this strategy had been chosen in the above mentioned text editor implementation, the reported problems might not have occurred.

Another strategy found in the Smalltalk libraries is based on hiding object creations behind message interfaces. Instead of writing 'obj species new' a message is sent. For example, instead of creating a controller explicitly, a view is sent the message *controller*. The specification says that the view is expected to return the name of an object in the view/controller component. The controller class is therefore not visible outside the view/controller component. It is therefore the view/controller component itself which controls the creation of controllers, and thereby also controls the template to use for creating controllers. This reduces the number of visible objects of the view/controller component and thereby simplifies the implementation of the view/controllers collaborators.

This idea has been further developed in the book on Design Patterns where several patterns, so called Creation Patterns give instructions on how to make a good design which controls the use of class names. The patterns Factory Method, Abstract Factory and Builder are good examples of controlling class names in relation to object creation. Factory Method says that particular messages should be defined so that the corresponding methods, and only these methods, contain the critical (substitutable) class names. Abstract Factory and Builder say that a particular object should be created which contains methods for creating objects and the critical class names should only be found in the implementations of the Abstract Factory and Builder objects.

Testing the class of an object

When testing the class of an object, eg, *object isKindOf: TTextSelection*, the class name is explicitly named. If the TTextSelection class is in another component it is a shared variable. Then this explicit check for the class of an object breaks the no external inheritance reliability requirement which in practice means that components can not share variables. Therefore, checking the class of an object should not be done. The same advice is given in (Johnson and Foot 1988) where is says:

> It is almost always a mistake to explicitly check the class of an object.

It is advised to use messages instead of checking the class of an object. By using messages, dynamic binding is used to let the behaviour of the system depend on the class of the receiver. This is as opposed to using an if-test to select a behaviour depending on the class of the object.

Class names used for typing variables
When variables are typed and a context is to refer to an object in a component, the context needs to type the variable referring to the object in the component. When variables are typed by using class names, a class name from a component which is used to type a variable in the component's context becomes a reference to a shared variable found in the component.

Generic types and type parameters provide ways to avoid using class names of other components to type variables. This technique is used to type the variables of a component when the component is created (instantiated) and not when it is defined. For instance, a component modelling an array is defined. When such a component is instantiated and an array is created, a type parameter is sent in order to define the types of the elements of the array. Since this is a way of avoiding class names in the code of a component, this is a way of avoiding shared variables. Instead, the class of, eg, the elements in the array, becomes a visible object which is passed from the component for the elements to the component implementing the array. This creates reliably substitutable element components.

There are people who advocate the use of abstract classes for typing variables. Also, many people argue for separating the class concept from the type concept and use signature types for typing variables. Abstract classes and types are used for typing variables and represent behaviour specifications, while classes are implementations of objects and are used as templates for object creation. This has been pointed out by many authors, eg, (Snyder 1986), (America 1987), (Cook et al. 1990) and (Blair et al. 1991). In the following paragraphs, the word type is used to denote abstract class or signature type.

Since types do not include any variable declarations or methods (only method specifications), the types of a component can themselves be components. This is because, even though classes implement one or more types, or inherit from one or more abstract classes, the class does not inherit any variables or methods. Therefore the classes meet the no external inheritance and reliable method lookup requirements. Likewise, the components referring to the types in type declarations of variables also meet the reliability requirements. Unless checking for the type of an object is done in components (in which case we run into a shared variable problem), the type component has no functionality or mission during runtime. It is purely used for type checking before the system starts executing. Therefore, typing variables by referring to types and not classes makes components more reliable in that this does not break any of the reliability requirements.

To avoid viewing types as shared variables and not create problems when a refinement has fewer visible objects than its specification, there is also one other requirement: when defining a refinement it must be possible for objects in a component to be instances of several types. This may be necessary if, eg, one visible refinement object takes the place of two visible specification objects. To avoid typing errors, the single refinement object must be of the same type as both of the specification objects. Therefore, it is necessary to have multiple inheritance of types in the language used for defining refinements.

The book on Design Patterns advocates using abstract classes for typing variables. Examples of languages which have separation between type and class are Trellis/Owl (Schaffert et al. 1985), Liskov's Theta language and the programming language Java. In Java, a type is called an interface. A class may implement many interfaces, thus creating multiple inheritance of interfaces.

8.4.3 Conclusion

When class names from other components are used, the class names should be seen as references to shared variables found in the other components. Therefore, to get components which can be reliably substituted, class names from a component should not be used in the component's context.

One way of explicitly referring to a class name when a component is to create an object from a class in another component, is to pass class names as parameters in messages. Then the classes are comparable to visible objects. Therefore, limiting the use of class names in the code makes it easier to make reliable specifications since the number of visible objects will be reduced. Reducing the number of visible objects makes it easier to control them and the reliable specifications become less complex.

The best solution to avoid using class names of other components for object creation is to let a component create all objects of its classes. Other components can then send messages to the component and in this way create new objects. In this way a component, or perhaps more correctly the component developer, gets control of which classes to use for object creation.

When "pure" types which do not include variable declarations and method bodies are used in type declarations, then this use of types does not break any reliability requirements. However, to be practically useful, the language used for defining refinements must allow a class to implement several types.

8.5 Reliable Substitution in Practice

8.5.1 How to make reliable refinements

The conclusions from the previous sections can be summed up in the following requirements on components which are to be reliable refinements:

A component which is to be reliable:
- can not inherit variables or methods from objects in the component's context
- can not compare the names of objects which are not in the component
- can only refer to class names within a component
- a reliable refinements can not send messages to context objects unless methods are found for the messages
- must have the same number or fewer visible objects than the specification and the names must be used as specified

In addition, to be a refinement, the component must have observable behaviour similar to its specification.

When a refinement has fewer visible objects than the specification, then one object takes the place of two or more of the specified visible object. The same refinement object must always be used in place of the two or more corresponding specified visible objects.

If variables and/or methods are inherited from the context, the variables and methods can not be changed during execution of the system in order to make the *component* a reliable refinement. Also, the inherited parts of the *context* can not be reliably substituted, ie, there is no way to make a reliable refinement of the inherited parts.

8.5.2 How to make reliable specifications

A reliable specification of an OCS component specifies both the component itself and the components making up the context of the component. This means that a reliable specification of a component must specify all actions observable from the context and all actions observable from the component. For each message which may be received by an object in the component, the specification must describe the resulting behaviour as observed from the context. Similarly, for each message an object in the context will receive, the specification must describe the resulting behaviour as observable from the component. In addition, a reliable component specification must specify:

- the maximum number of visible objects which refinements of the component may have.
- the maximum number of visible objects from the other components.
- how the names of the visible objects are sent and received in messages from and to the component and its context, ie, specify in which parameters in which messages the names of the different visible objects are found.
- which visible objects from each component is initially known to the other components.

In general this means that a reliable specification must identify objects, not just categories of objects such as types, classes, roles etc. This requirement is a consequence of the requirements related to reliable use of names, the reliable substitution and the definition of the reliable refinement relation using a reliable substitution. One way to intuitively understanding this requirement is that the number of visible objects must be known so that a programmer can use the correct number of different variables to hold the different object names (see section 8.2 for a more detailed explanation of this requirement).

As argued above, classes defined in one component and instantiated in another should be viewed as visible objects. Therefore:

When classes are parameters in observable messages, they must be treated as visible objects

Specifying visible objects and their use is an absolute requirement which is necessary and sufficient to make reliable specifications. However, there are some additional advice on how to make specifications which are the consequence of the requirements on reliable refinements. These additional advice are listed below. Thereafter some comments are made on the consequences of the fact that specifying visible objects is the only absolute requirement on reliable specifications.

Additional advice on how to make reliable specifications

A reliable refinement can not inherit variables from objects in its context. This means in practice that reliable refinements can not update variables in objects in other components. If such actions are found in a specification, it might be impossible to make reliable refinements of the specification[14]. Therefore, the following advice can be given:

> Specifications should only have actions which access and update variables within a component

Explicit use of a class name from another component is comparable to accessing variables outside the component. One exception is if the class is an abstract class, ie, an interface description, and the language used to create reliable refinements allows multiple inheritance. Therefore, the following advice can be given:

> In the description of a components behaviour, a specification should not explicitly name non-abstract classes from other components.

A specification's behaviour can depend on the number of visible objects from its context. When the behaviour depend on the number of visible object, then the specification includes if-tests comparing the names of objects in the context. If a component is to be a refinement of such a specification, the refinement must also include similar if-tests comparing context object names. This breaks the reliable if-sentence requirement. Therefore, it might be impossible to make reliable refinements of a specification with an observable behaviour which depend on the number of visible objects from its context. (Exceptions as for specifications which update shared variables as one of its alternative behaviours). Therefore, the following advice can be given:

> Specifications should only use if-tests to compare the names of objects within the component

The reliability requirement "reliable method lookup" says that a component which is a reliable refinement can not have methods which are found in the component's context. This limitation does not apply to components which are part of reliable specifications. However, there are reasons for wanting to make specifications with components with reliable method lookup, ie, specification components which do not inherit methods from their context. The main reason is that if the specification component inherits methods from the context, this can give unwanted specification behaviour when the context is substituted with refinements. The unwanted behaviour occurs when the context is substituted with refinements. Then the inherited methods will disappear from the system, unless they are replaced with visible objects in the context's refinement which give the same behaviour of the specification as the replaced methods give. To ensure that the new methods in the refinement give the same behaviour as the replaced methods, the behaviour given by the methods have to be specified in some way. Then the specifications become larger and more complex only to allow specification components, but not reliable refinement components, to inherit methods from their context. To keep specifications as small and simple as possible, it is therefore a good idea to follow the next advice:

> Specifications should only use methods which are local to the components.

The reliable message-send requirement says that a component which is to be a reliable refinement can not send messages to context objects unless methods are found for the messages. By the way observability of error actions is defined, this does not apply to components in reliable specifications in that a reliable specification component can send messages to the context which the context does not understand, ie, does not have methods for. However, if specification components send messages to objects in other components and no method is found, this might be an indication that the specification is not as the designer intended. Ensuring that specifications are as intended and describes behaviour as the designer actually wants it to be, is a different problem than Ensuring reliable refinements. This is discussed further in the last subsection of this section: Ensuring correctness of specifications.

In addition it is required that message selectors used as parameters in messages must be equal in observably similar actions. This means that a specification and its refinements send the same message selector to their collaborators. Since the message selector is equal, the collaborators will behave equally in this relation, independently of who sent the message selector. Therefore, sending a message selector as a parameter in a message is wasted effort. Therefore, the following advice can be given:

> Specifications should not have message selectors as parameters in observed messages

---

[14] It would be possible to make a reliable specification, if updating of shared variables is just one of the alternative behaviour of the specification. If other alternative behaviours do not update shared variables, then a reliable refinement can be made. In such situations, the updating of shared variables will just be a part of the specification which will never by found in a reliable refinement. Updating the shared variable can therefore be seen as an unnecessary complexity in the specification.

Conclusion

Below the advice on how to make reliable specifications is summed up in one list:

Reliable specifications must specify:
- all possible sequences of actions observable from the component and by the context
- the resulting behaviour as observable from the context for each message which may be received by an object in the component
- the resulting behaviour as observable from the component for each message an object in the context will receive
- the maximum number of visible objects which refinements of the component may have.
- the maximum number of visible objects from the other components.
- how the names of the visible objects are sent and received in messages from and to the component and its context, ie, specify in which parameters in which messages the names of the different visible objects are found.
- which visible objects from each component is initially known to the other components.
- when classes are parameters in observable messages, they must be treated as visible objects

In addition configurations which are to be specifications for reliable refinements
- should only access and update variables within a component.
- should only use if-tests to compare the names of objects within a component.
- should only use methods which are local to the component.
- should not have message selectors as parameters in observed messages
- should only use class names within a component, names of abstract classes might be an exception

Here are some advice from the conclusions of section 8.3. This advice gives suggestions on what to do instead of breaking the reliability requirements for reliable refinements:

If a component is to be a reliable refinement of some specification, the no external inheritance reliability requirement limits the use of shared variables to sharing variables within a component. Instead, when a component wants some data from a collaborator it should send a message which returns with the data.

Give priority to delegating functionality to a component rather than inheriting it from a class. It is not possible or meaningful to have superclasses as components and at the same time have reliable substitution of such components.

Methods in other components should not be changed during execution. Use such things as Smalltalk block-objects and/or the Visitor pattern when you want a component which should have different observable behaviour depending on which component it collaborates with.

Comparing object names should be avoided unless refinements are expected to have exactly the same number of visible objects as defined in the specification. Instead of using if-sentences it is better to redesign and use dynamic binding. If object names must be tested, then at least one of the names must be local to the component where the test is done.

Consequences of the requirements on reliable specifications

There are mainly three ways of making reliable specifications. One is to describe objects displaying the desired observable behaviour, the second is to define the desired sequences of observable actions directly, ie, the observable traces of the components and the third is to define the sequences of observable actions indirectly by for instance invariants. The first approach was the starting point for the presented formalisation. However, the two latter might in some cases be better alternatives since these might give more direct descriptions of the collaboration of the components.

Independently of which approach is taken it is necessary to specify the components' visible objects, and not just the types or classes of objects in the different components. As noted above, only two kinds of observable actions can be found in reliable refinements of a specifications. These are message-send actions and object-creation actions. Therefore, to avoid unnecessary complexity, the observable traces should only include these two kinds of actions.

8.5.3 Ensuring correctness of specifications

The reliable refinement relation is defined in order to be able to substitute components without introducing observable behaviour which is not specified. This is as opposed to Ensuring that a specification describe the observable behaviour the author(s) think they describe. In the first case, the specification is assumed to be correct. It is assumed that the initial step has been taken in the development of the system in that the system's

parts have been specified. Reliable substitution, and therefore the refinement relation, is focused on the later steps of the system development, when new versions of the system's parts are to be developed. This is different from theory and practice which are concerned about whether or not a specification is correct. Their focus is on the initial step when there is no previous system descriptions to compare with. Many different techniques have been developed for helping authors ensure that their initially made specifications are as intended.

Ensuring that a specification is correct has several aspects. One aspect is Ensuring that the specification describes a system which is useful and preferably as useful as possible. Another is Ensuring that the specification actually say what the author intended.

Creating useful systems usually requires careful analysis and may require the participation of users in order to understand their needs and requirements. There are a number of analysis methods which claim that they help system developers through this phase by prescribing good processes and by having notations which can be read and understood by the coming users of the system. Such methods are human centred and related to "soft sciences" such as sociology, psychology and anthropology.

Ensuring that a specification actually says what the author intended is another problem. One technique used in this case is allowing the author to express his/her ideas in more than one way. Today this strategy is supported by more or less automatic tools for Ensuring that the different specifications say the same thing.

One example where the developer is allowed to view and edit different aspects of a system specification is the OOram tools. Through the tools the various aspects of the system description are always kept consistent. This means that when a developer changed the description in an editor showing one aspect of the description, then all the dependent aspects are automatically updated. If other aspects are shown on the screen, these are redisplayed so that they show a correct version. A developer can then easily see the consequences of his/her decisions on the different aspects of the system.

Using algebraic specifications and first order logic as a complimentary way of describing what an imperative system description is expected to do, is found in various versions. One example of this is Eiffel contracts; not to be confused with contracts as used in chapter 2. Eiffel contracts consist of assertions which are seen as elements of formal specification. An assertion is the expression of some property of objects, notably properties of the variables of objects and relations between values the variables hold. The technique of Eiffel contracts is the idea of invariants, pre- and post-conditions developed for imperative languages applied to a concrete programming language (Meyer 1994).

The assertions are found as class invariants expressing properties which must be ensured when an object is created and maintained by every method. A simple example is:

```
x > 0
aBool = (x = 0)
```

which says that the variable x will always hold a value greater than zero, and the variable aBool will be true if x=0, false otherwise.

Assertions are also found as pre- and post-conditions in methods. An example (preconditions are denoted "require" and post conditions "ensure"):

```
method(x: OBJECT); is
        require y /= Void
        do …
        ensure y = x
```

where y is some variable declared in the class the method is created for. The precondition says that the variable y refer to some object, not Void (which is Eiffel's term for nil / none). The post condition says that y will refer to the same object as x when the method has finished executing.

As can be seen from the examples, assertions describe quite different properties than the properties modelled by Omicron. Assertions express properties of the internal state of an object while Omicron reason about the observable actions of objects. Eiffel contracts and other similar techniques are quite useful in Ensuring that an implementation is done correctly in relation to the use of variables and the change of the object's state. It also helps avoiding duplication of state tests for objects since the assertions explicitly express what can be expected about an object's state.

A method's pre-condition can be seen as specification of how callers of the method is expected to behave, ie, under what conditions they are expected to send the message. A methods post-condition is a specification of the

component's behaviour when the method is called and the caller behaves as expressed in the pre-condition. However, traditional post-conditions do not express properties such as message sending or object creation. Instead they focus on values; specifying return values or updates of the internal state of the receiving object. Therefore, they are used to express quite different properties than the observable behaviour described by Omicron configurations. In other words, Eiffel contracts and Omicron specifications fulfil different needs in relation to creating, documenting and using components. Invariants, pre- and post-conditions, and in general algebraic specifications, are useful for abstractly specifying how single objects manipulate data while Omicron is useful for specifying sets of objects collaborating by sending messages and creating new objects. As most object-oriented systems do both, combining the two might reveal itself as a useful combination.

However, by means of auxiliary variables such as history (trace) variables, one may specify abstract object behaviour by invariants and pre- and post-conditions. This is discussed in relation to ABEL in chapter 9 on related work.

8.5.4 Ensuring reliability of refinements

C. A. R. Hoare stated that:

> *There are two ways of constructing a software design:*
> - *One way is to make it so simple that there are obviously no deficiencies*
> - *and the other way is to make it so complicated that there are no obvious deficiencies.*

If a specification is simple, it might be easy to make a component which obviously is a reliable refinement of a component in the specification. On the other hand, if a specification is complicated, it might be impossible to convince oneself that a reliable refinement exists without automatic proofs. In practice, the quality of specifications is therefore an important factor when Ensuring that a component is the reliable refinement of a component specification. It is important that the specification is easy to understand, by both being as simple as possible and written using good language and appropriate notations. This is the concern of many of the object-oriented software engineering methods referred to in this thesis.

However, a good, readable and reliable specification is not enough to ensure that a component is a reliable refinement of the specification. It must also be shown that it meets the reliability requirements and that it has the observable behaviour described in the specification. Below are some rough sketches of how one in practice could test weather or not a component is a reliable refinement of a specification.

To ensure that a component is a reliable refinement, two aspects of the component have to be verified; it must be tested that the component meets the reliability requirements and it must be tested that it has similar observable behaviour to its specification. Some reliability requirements can be checked at compile time while others have to be checked at runtime. Similar observable behaviour must be tested at runtime, or through some simulation of runtime. There might also be some cases where static program analysis can be used to confirm that the component has similar observable behaviour to its specification. This is the topic of much work on incremental modification techniques which ensure that the observable behaviour is maintained by certain kinds of modifications. These techniques mainly focus on functional behaviour of components and do not consider message-send and object creation actions.

Compile time checking and static analysis has the advantage over runtime checks in that it is usually faster and takes a finite amount of time. A sketch of an idea on how to statically check some of the reliability requirements is as follows:

The main idea is to type slots in order to recognise those which can only hold names of objects which are local to the component. We call such slots local slots. The type checker/type inference algorithm can then check that local slots are only assigned the names of local objects. The type checker / type inferer must also do the following checks to ensure that the different reliability requirements are met:

Reliable if-sentences	The if-sentences only compare the values of slots where at least one is a local slot.
No external inheritance	Inheritance slots are local slots. If static classes are used, then no external inheritance is always ensured.
Reliable method lookup	A slot must be a local slot if its name is equal to a message selector in the component and/or the component's context. When classes are static, reliable method lookup is always ensured.

Reliable message sending In this case traditional type checking can be used, with the danger that the type checker rejects a component as unreliable, while it is not[15].

When these four reliability requirements are fulfilled, the "only" thing which remains to check is that the reliable component has similar observable behaviour to the specification. For OCS components, this usually means testing the component in the context of a test environment or as a part of a complete system. R.V. Binder defines two aspects of component testing in the paper "Design for Testability with Object-Oriented Systems" (Binder 1994) where he writes:

> *"To test a component, you must be able to control its input (and internal state) and observe its output. If you cannot control the input, you cannot be sure what has caused a given output. If you cannot observe the output of a component under test, you cannot be sure how a given input has been processed."*

When we have a reliable specification of the context of the component and reliability requirements of refinements of the context, we have control over the input. The limit on visible object names and the requirement not to use message selectors as parameters in messages from the context to the component may be viewed as examples of how it is necessary to restrict input in order to have control. Controlling the internal state of the object is done by the no external inheritance, reliable if-sentences and reliable method lookup requirements.

The reliable message sending requirement is linked to the observability of the output. This is because of the error model defined for Omicron where message not understood errors are not observed, even when the intended receiver is an observer. As discussed in chapter 5, reliable message sending is a necessary and also very common requirement. Type checking and type inference is commonly used to check this property.

Observability of actions was formally defined in a previous chapter. The formal definition was easy to do compared to observing actions in practice. Actions are observed when a specification and its refinement actually exist and are to be executed together with a context. To verify similar observable behaviour, actions have to be observed and compared. Actions must be observed and compared as follows:

> For each action sequences from the system consisting of the refinement and the context,
> there must be an observably similar action sequence in the system consisting of
> the specification and the context.

Since there may be an infinite number of actions in the action sequences from the system with the refinement in it, there will in general be no practical way to fully ensure similar observable behaviour. However, some approximations may be done. Usually infinite sequences of actions occur as the system perform the same action sequence infinitely many times. Then the components can be inspected after a certain number of repetitions of some action sequence. The state of the components can then be studied to see if it can be assumed that the components are able to repeat the action sequence indefinitely or at least repeat it as many times as is necessary in practical situations. A technique which can be used to reason about components' behaviours are invariants and pre- and post-conditions. By using this technique it can be easier to establish if action sequences can be repeated indefinitely.

There are many problems related to showing observable similarity of infinite sequences of actions. However, even the simple problem of showing observable similarity of finite sequences of actions is not solved in practice. To solve this problem it is necessary to create test environments which can observe actions and compare them. To be practical, such environments could be integrated with tools used for component specification. Since the specifications are operational, they can be executed. Therefore, the specified context of a component can be transformed into a test driver for refinements of the component.

Before good test environments exist, test drives have to be made by hand. This can be a quite large amount of work. The last resort to testing for observable similarity of behaviour might be running the component with a context and check this systems' visible output. Visible output is such things as error messages, updates of the display or output to file or network. For instance we can test a component as follows:

> Run the systems with a new component and an existing context and
> run the system with the existing context and a component which is seen as a specification
> and then see if the two systems have similar visible output

The visible output of the systems can be tested, eg, by going thought the use cases of the systems. One can then hope that if the visible outputs are similar, the new component and the specification component have similar observable behaviour when collaborating with the context component.

---

[15] see discussion about subtypes vs. refinements in the section 8.3.5 on reliable message sending

Compared with non-reliable components, it is a larger chance that a component which is a reliable refinement will also give similar visible output to its specification when combined with a new context and where this new context has similar visible output to the old context when executed together with the specification component. This can be illustrated as follows:

The new component
is reliable relative to an old component and
the systems with the new component and an existing context has similar visible output to a system with an old component and the existing context

A new context
is reliable relative to the existing context and
the system with a new context and the old component has similar visible output to a system with the old component and the existing context

The reliability properties of the new component and new context will increase the chances that:

the systems with the new component and the new context has similar visible output to a system with the old component and the new context
and also that
the systems with the new component and the new context has similar visible output to a system with the new component and the existing context

However, the starting point for the development of new versions of components are specifications. Therefore, it is important that specifications are presented in a form which is easy to understand and that the specifications are as simple as possible. To make the specifications as simple as possible it is important that only necessary aspects of the components are described. The above rules for making reliable specifications lists the necessary aspects of component specifications. By taking the most readable software engineering methods and developing them further based on the rules for making reliable specifications, one might get one step further in making better specification languages for OCS components.

8.5.7 Reliability and reusability of components

As mentioned in chapter 1, in relation to reuse, the substitution proposition says that a component which is to be reused must have a reliable specification and the implementation must be a reliable refinement of the specification. If not, the implementation might not function as planned when the component is collaborating with some new component.

If the reused component does not have a reliable specification, the reused component can function as its own reliable specification. It might be difficult to read and understand, but it meets the requirements to specify visible objects and their use. If it is its own specification, it has no more visible objects than itself and use them as it does and it therefore meets this reliability requirement. However, to be a reliable refinement it must also meet the other reliability requirements. It must therefore not inherit from its collaborators since these may change when the component is reused. For the same reason, it can not use methods found in collaborators of compare the names of objects in its collaborators.

To support the reuse of a component, the component should come with a reliable specification of its collaborators. If a reliable specification is not available, the developer of the new collaborator must guess what the reused component expects of visible objects. If the guess is wrong, the component user can experience unanticipated behaviour from the component. Similar problems will occur if the observable behaviour of the component's collaborator is not specified and the creator of the new collaborator must guess.

Since it is required that the reused component is a reliable refinement, it is expected that it meets the reliable message sending requirement. This requirement implies that the specification of the reused component's collaborators must include specifications for all messages the reused component sends to the collaborators. The collaborator specification should also specify the maximum number and use of visible objects from the collaborator which are known by the reusable component.

Also, as mentioned in chapter 1, the substitutability proposition does not imply that a new collaborator of a reusable component needs to be a reliable refinement of the collaborator specification. The new collaborator need only be a refinement of the specification. This holds as long as the reused component is not substituted with a new version.

The new collaborator need not be a reliable refinement when the reused component is not substituted since the slots, methods and objects which are in the reused component will not be replaced. Then it is not necessary to have reliable method lookup, reliable message-send and reliable if-sentences in the new collaborator. The new collaborator can also inherit from the reused component. However, to be sure that the reused component behaves as specified, the new collaborator must have no more visible object names than specified and the visible object names must be used as specified. If the new collaborator has more visible objects than the specification and/or use them differently, the reused component *might not* behave as specified. On the other hand it might. However, if the new collaborator has visible objects as specified, the reused component is guaranteed to behave as specified.

For example assume that the TextEditor is reused with various refinements of TextModel. For the collaboration between the reused TextEditor component and a refinement of TextModel to function as planned, it might be necessary that the refinement of TextModel must have no more visible objects than TextModel and use them in similar ways as observable from the TextEditor. However, if the developer is lucky, the TextEditor will function as planned even if the refinement has more visible objects than TextModel and/or use them in different ways.

Conclusion

A reusable component must be a reliable refinement of its specification. If it does not come with a reliable specification of itself, it can be a reliable specification itself. Then, however, the specification may be unnecessary detailed.

A reusable component should come with a reliable specification of the components in the context, ie, a specification of the component's collaborators. When a new collaborator component is made, the new collaborator should have the no more visible objects than the specified and use the visible objects in the same way as described in the specification. A new collaborator component need only be a refinement of a collaborator specification. It does not need to be a reliable refinement of the specification provided the reused component is not later substituted with a new reliable refinement of itself.

8.5.7 Summary of lessons learned

What follows is a summary of the most important lessons learned in relation to practical use of the results of this thesis.

The number of visible objects can not be abstracted away:

When specifications describe components in extensible systems, the specifications must be a safe starting point for implementing the substitutable components of the extensible system. Therefore, if a specification is to describe a component of an extensible system, the number of visible objects can not be abstracted away. If the number of visible objects is abstracted away, the specifications are in some sense incomplete since it is not possible, or difficult, to make implementations which will function as planned. The difficulty is concerned with knowing how many objects in other components a component is expected to know of and distinguish between. If an implementation makes an error in distinguishing between objects from other components, the result is typically wrong messages to the wrong objects. The best to hope for in such a situation is that a system error occurs immediately. The much worse alternative would be that the wrong message to the wrong object lead to an inconsistency in the system which only surfaced days, weeks or years after the message was sent.

Inheritance between components creates trouble:

Use of inheritance between components complicates component development, designs and specifications. Typically this happens when a class is considered a component and developers are allowed to make subclasses and use them in place of the superclass. The extra complexity is added by the necessity to specify more than the components' observable behaviours. Also, it must be known how to make reliable refinements of the specified components. It must then be known what can change and what must be remain stable, ie, what is the implementer allowed to change and what must s/he keep the same. In particular, it is not clear how to reliable implement or change methods which are inherited by other components. A lot of work remains before one is able to make reliable specifications and refinements of components which are allowed to inherit from each other.

If-sentences in object oriented languages can create trouble:

It is not considered good object-oriented practice to use if-sentences to test if two variables refer to the same object. It was therefore an interesting discovery that the use of if-sentences also created problems in our theory. Therefore, there now exists both practical and theoretic support for not using if-sentences to test if two variables refer to the same object. Instead, it is considered better in both theory and practice to use dynamic binding to control the behaviour of a system.

Using message selectors as parameters are useful but can create trouble:

The use of message selectors as message parameters in the Smalltalk pluggable editors makes these editors very useful, easy to reuse and it is easy to make new collaborators for the editors. It was therefore a small surprise when our theory concludes with results which say the opposite of the practical experience of the users of these components. However, makers of pluggable editors, and particularly people which have converted a non-pluggable editor to a pluggable editor, has experienced that this conversion is not always straight forward. A lot of effort have to be put into getting the editor implementation right so that errors don't appear when the pluggable editor is made to collaborate with a component which uses a new message selector as parameter.

How objects are grouped into components will influence maintainability

The observable behaviour of the various components in a system is defined when objects are placed in components. Therefore, when defining components one is at the same time making decisions about which parts of the system should be possible to substitute independently of each other. Therefore, the partitioning of a system into components is one of the factors which will influence the maintainability of a system and the reusability of the components.

A discussion of where to place an object holding the set of views for the model object in the model-view contract example of chapter 2 can illustrate this point:
If the set-object is placed in a component by itself, the specification will include a precise description of the interaction between the set-object and the model. By having this precise description of these interactions, the job of making an implementation of the set-object is easy compared to *not* having such a precise description. This would therefore be a good choice if it would be common to want to replace the implementation of the set-object, or it is to be developed by a separate team.

The drawback with placing the set-object in a separate component, compared to placing it together with the model in a component, is that the specification becomes more complex in that there will be more components. Also, having the set-object and the model in two separate components would mean that changing the interaction pattern between these two should be taken more seriously since it is part of a system specification. The interaction pattern should therefore first be changed after long and careful considerations. If the set-object and the model were one component, their interaction would not be seen as part of the system specification, and could therefore be seen as details in the specifications of the model component. This interaction could therefore be changed by those who refine the model component, provided the observable behaviour of this combined component is in accordance with the specification.

The general and obvious rule is that objects which are assumed to be developed or changed together should be placed in the same component, while sets of objects which it should be possible to develop and change separately from the rest of the system should be placed in a component by themselves.

Tool support for ensuring reliable substitution

Quite a lot of further theoretic and practical work is necessary to ensure reliable refinements in a general, practical case. It is particularly difficult to make complete and easy to use tools which help developers ensure that they have reliable substitution of components. The main difficulty or amount of work will be related to observing and comparing sequences of actions, especially since components may have infinite sequences of actions.

However, results such as the reliability requirements and the definition of the reliable refinement relation, must be available before such tools can be made. The results of this thesis can therefore be viewed as a small, necessary first step in the process of making tools which help component developers make reliably substitutable components. Also, a practising component developer can learn some simple lessons from the presented theoretic work, where the most important lessons are summed up above. These lessons should be easy to incorporate into existing development methods and OCS design practices.

CHAPTER 9

Related Work

This chapter presents different formal approaches to describing software and their relation to Omicron.

There are mainly three different traditions in modelling computer systems: the distributed systems tradition, the functional or algebraic tradition and the object-oriented tradition. Omicron belongs to the last tradition. The distributed systems and functional traditions have a long history of formal work which are solidly based on the concepts of the traditions. Comparing this with the object-oriented tradition, we see that most formal work done on objects is firmly rooted in one of the other traditions. There are some exceptions which are presented in section 9.3.

The problem with rooting formal work in other than the object-oriented tradition, and then reasoning about objects, is that the different approaches have different views of a component and specify similarity of behaviour in distinctively different ways. Therefore, it is not clear how object-oriented concepts map to the concepts found in other traditions. What typically distinguish functional and process formalisms, and object-formalisms based on these, from Omicron and the reliable refinement relation are:

- they define a total equality relation and not a partial context dependent refinement relation
- they do not explicitly define a context - which may be substituted by a refinement
- they do not take object creation and inheritance/shared variables into account
- they specify single objects, not components which are configurations of objects, or there are no distinct object and/or component boundaries

Because of such differences, it is not clear whether components are reliably substitutable in our sense when found similar when formalised by using functional or process modelling languages.

In section 9.1 Omicron is compared with other models of distributed systems such as process models and Actors.

Section 9.2 compares object-oriented models in general and Omicron in particular to traditional state based functional models.

Section 9.3 presents various other formal models of objects and object behaviour. Conclusions include that the formal models focus on other aspects of objects than their observable sequences of actions or the formal models are not worked out to the extent that they can be used to define and reason about reliable substitution.

Section 9.4 compares the substitution proposition to assumption/guarantee specifications, introduced in (Jones 1983). Martín Abadi and Leslie Lamport have worked on showing properties related to composition and decomposition of components specified by assumption/guarantee specifications. They have formulated a composition principle (Abadi and Lamport 1993). It is shown that the Omicron reliable refinement relation is in line with the composition principle.

9.1 Models of Distributed Systems

In this section Omicron is compared with other models of distributed systems. Subsection 9.1.1 compares Omicron to general models of distributed systems, while the next two subsections compare Omicron to particular models of distributed systems. The models selected for comparison are those models which most resemble Omicron, namely the Actor model (subsection 9.1.2) and the π-calculus (section 9.1.3).

9.1.1 Discussion of distributed system models

This section will compare Omicron to general models of distributed systems. The comparison is done by describing different features of distributed systems. For each feature, Omicron and other models are compared. An article by Lamport and Lynch (Lamport and Lynch 1990), hereafter referred to as (L&L), gives a thorough presentation of different features of distributed systems. The L&L article is used as a basis for the below discussions which compare Omicron with other models. The L&L article is used as basis instead of making up a new description of these features.

L&L write:

> *Underlying almost all models of concurrent*[16] *systems is the assumption that an execution consists of a set of discrete events, each affecting only part of a system's state. Events are grouped into processes, each process being a more or less completely sequenced set of events sharing some common locality in terms of what parts of the state they affect.*

This corresponds well with the Omicron (and in general the object-oriented) model of computing.

They divide the models into two categories according to the mechanism employed for interprocess communication:

> those in which processes communicate by message passing and
> those who don't

The Omicron calculus is a message passing model between components. L&L further define a taxonomy for classifying message passing models by the assumptions made about four separate concerns: network topology, synchrony, failure and message buffering. These four concerns are each handled in the four subsections below:

Network topology:

The topology describes which processes can send messages directly to which other processes and is described by a communication graph where the nodes are processes and the arcs denote channels for message passing. L&L do not mention models where this topology can be changed during the processes' lifetimes. In Omicron the topology can change during execution by objects passing object names as parameters in the messages.

L&L's limitation to fixed topologies corresponds with most message passing models in that these models assume a fixed topology, eg, CCS(Milner 1989), LOTOS (Bolognesi and Brinksma 1987) etc. The notable difference is Milner et.al.'s π-calculus (Milner et al. 1989a), (Milner et al. 1989b) where systems with an evolving topology are called mobile processes. Also CHOCS (Thomsen 1993) allows network topology to change.

There are many approaches to modelling objects using process modelling formalisms with fixed topologies, eg, (Moreira and Clark 1994), (Allen and Garland 1994), (Papathomas 1991), (Papathomas 1992). All has the same limitations as the traditional process specification languages in that it can not handle topology changes. None of the papers have proofs showing how relations between processes map to relations between objects and how properties of the relations can be used to prove reliable substitution of components. (Honda and Tokoro 1991) and (Sato and Tokoro 1992) define a calculus called RtCCS for real-time object-oriented computation. It extends Milner's CCS by introducing a tick action and a time-out operator. In relation to Omicron it therefore has the same limitations as CCS in that it can not handle topology changes and instance creation actions and the defined relations are bisimulations, not refinements.

---

[16] It seems that L&L do not make a clear distinction between distributed and concurrent systems. "Distributed" is the term they use both before and after this paragraph. "Concurrent" is only used in this paragraph.

Other approaches create new languages inspired by process languages, particularly the π-calculus, eg,

(Honda and Tokoro 1991) define a bisimulation relation, but show no properties of this relation,

(Vasconcelos 1994) does not define anything similar to a monotonic relation or a refinement relation

(Nierstrasz 1993) (earlier version in (Nierstrasz and Papathomas 1990a)) presents an object calculus, but leave the following problems open (page 167):

> When are two object descriptions behaviourally equivalent ?
> What is an appropriate type theory for "plug compatibility" of objects ?

Some answers to these questions are suggested in this thesis.

Synchrony:

L&L define a complete asynchronous model as:

> *A completely asynchronous model is one with no concept of real time. It is assumed that messages are eventually delivered and processes eventually respond, but no assumption is made about how long time it may take. Other models [not complete asynchronous] are models which introduce the concept of time and assume known upper bounds on message transmission time and process response time.*

Asynchronous message passing with no time constraints is modelled in Omicron by each active object having an equal probability of being executed. L&L introduce synchronous communication as follows:

> *[CSP-example] Unlike the case of ordinary message passing, the input and output commands are executed synchronously. Execution of a j!v operation (a message send in process i) is delayed until process j is ready to execute an i?x operation, and vice versa. Thus, a CSP communication operation waits until a corresponding communication operation can be executed in another process.*

Omicron is completely asynchronous in that an object can always receive a message. Therefore, a sender does not have to wait for a receiver to be willing to communicate. Synchronous communication is found in the π-calculus and all the above mentioned calculi inspired by this calculus. As omicron, the Actor model (Hewitt 1977), (Agha 1986) has asynchronous message passing.

Failure:

In message passing models both process failures and communication failures may be considered. Communication failure occurs when a message sent is not delivered for some reason or other. This is not a topic in Omicron (or π). However, process failure where the failure is due to an error in a process is modelled in Omicron. Omicron allows only halting failures, where a failed process does nothing.

Omicron has the same fault tolerance as asynchronous communication models, since an object in an Omicron system will not be removed when one of its sentences gives an error action when executed.

It is not clear whether the failure semantics of the parallel and sequential versions of Omicron correctly model errors in object component systems. That this is difficult to judge is supported by L&L which write:

> *"Failure models are problematic because it is difficult to determine how accurately they describe the behaviour of a real system. "*

Further work has to be done to see if the Omicron error models are the best and most useful failure model for object component systems.

Message buffering:

L&L writes:

> *"In message-passing models, there is a delay between [the time] when a message is sent and when it is received. Such a delay implies that there is some form of message buffering. Models may assume either finite or infinite buffers."*

Omicron has an infinite buffer in that an object is always willing to receive a message, ie, a method copy is created so there may be infinitely many method copies at any one time. The Actor model also assume infinite message buffers.

L&L further writes:

> *"If a link's buffer can hold more than one message, it is possible for messages to be received in a different order than they were sent."*

In Omicron messages are received in the order they are sent, since a method copy is created when an object receives a message. However, there is no guarantee that the method copies will be executed in the same order as they were created because the transition rules non-deterministically select which rule of action to apply at each step. Thus the observable result is that the order the messages were sent is not preserved in the order of execution of methods. The Actor model has message buffer semantics which allow messages to be sent and received in different orders. However, in the Actor model an object will only have one method copy executing at any one time as new messages will not be received before the previous method has terminated. In Omicron there may be any number of executing messages at the same time.

Shared variables

L&L define other models of communication than the message passing model, namely shared variables. Communication through shared variables is not found in the π-calculus or Actor model. However, Omicron enables communication through shared variables in that several objects share slots through inheritance.

A problem with shared variables is that many algorithms need to somehow control the access to the shared variables. To be able to control access to the shared variables it is common to use semaphores [L&L page 28]. Examples of such control is implemented in Omicron by using the possibility to assign the same value to several slots in one atomic operation. This possibility is used in the translation from π to Omicron found in appendix B.

Another way to control shared variable access is through monitors [L&L page 47]. Actor objects are monitors in that an Actor object only executes one method (equivalent) at the time. Omicron objects are not monitors. This is common to many object-oriented languages. This is because an object may send a message and then wait for a return and while waiting for the return get a new message. This new message will result in the execution of a new method while the other method for the same object is waiting for a return. This can lead to unanticipated access of shared variables.

By introducing synchronisation primitives, synchronisation problems emerge such as "contention" problems and co-operation problems. When there are contention problems, a process is not able to make unlimited progress when other processes fail to progress, and when there is co-operation problems, the progress of one process depends upon the progress of another. Such problems can lead to deadlocks.

Since Omicron has asynchronous message passing then synchronisation problems do nor emerge directly from the way message passing is done. However, synchronisation can be modelled such as is done in the translation from π to Omicron in appendix B. Here a semaphore controls the access to a shared variable. The use of such synchronisation mechanisms in Omicron can give contention and co-operation problems. Such problems occur when an object actively waits to receive a message in order to start or continue one of its tasks.

Specification language vs. verification system

The Omicron language can be used to make specifications of object component systems. However, the Omicron language is mainly intended as a formal framework for reasoning about and verifying congruence properties of OCS components. (Lamport and Lynch 1990) page 1167/1168 describes specification and verification as follows:

> *"In verification, the properties to be proven are stated in terms of the algorithm itself - that is, in terms of the algorithm's variables and actions. [In] the related field of specification, […] the properties to be satisfied are expressed in higher-level, implementation independent terms […]. Specification methods must deal with the subtle question of what it means for a lower-level algorithm to implement a higher-level description. This question does not arise in the verification methods that we discuss, since the description of the algorithm and the properties to be proven are expressed in terms of the same objects."*

Omicron configurations were seen as specifications in the previous chapters. Such a specification is not any different from an implementation. Therefore it is not done in "implementation independent terms". However, by the way the reliable refinement relation is defined, the specification and its reliable refinements may have completely different local structures. In this perspective, an Omicron specification can be seen as implementation independent, even if it is made using the same terms as implementations. Omicron can therefore be used as a specification language as defined above.

Omicron is also a verification formalism, as defined above, since the description of the algorithm and the properties to be proven are expressed in terms of the same objects.

Abstraction in relation to verification and specification.
A specification is an abstraction of things it describes. If a specification of observable behaviour should be abstract, then message send actions and object creation actions should be abstract. Omicron does not give abstract specifications of object components' observable behaviour. Therefore Omicron can not be used to describe abstract designs where observable actions are abstract. Instead, the observable behaviour is exactly defined in full detail by concrete actions.

However, the description of the object components are abstract in that the components internal details are hidden, ie, the Omicron refinement relation describe abstractions of software components.

Omicron's relation to abstraction can be summed up as follows:

The Omicron language is not abstract
- it is an (abstracted) programming language which have variables and statements which are executed.

The Omicron language semantics is not abstract
- it is defined by an (abstract) operational semantic which specify execution steps.

The Omicron refinement relations are abstract
- it only focus on externally observable behaviour and hide internal details of the configurations

9.1.2 The actor model

The Actor model (Hewitt 1977), (Agha 1986) is created as a foundation for concurrent object-oriented programming. The actor model corresponds in many ways to the object component design's idea of an object. The Actor model is mainly a specification (description) language, but has also been given formal definitions which support reasoning about Actors.

An actor is an object which carries out its actions in response to receiving a message. The actions it may perform are:

Send communications to itself or to other actors (similar to sending messages)
Create more actors
Replace its behaviour with a new replacement behaviour (similar to changing state)

The replacement behaviour is yet another term to describe the new "actor machine" produced after processing the communication, ie, the new state of the actor.

The correspondence between actor terminology and object-oriented terms as follows:

Actor	*Object-oriented*
Script	Class declaration
Actor	Object
Actor Machine	Object state
Task	Message
Acquaintances	Attributes

The most notable difference is that an actor is a monitor which processes one message at the time. Also, an actor's description explicitly defines which messages the actor is willing to receive. The messages an actor is willing to receive may change as part of the actor's replacement behaviour. An actor is usually seen as having a static set of methods which corresponds to the messages other objects might send it. This is different from how Omicron and other traditional object-oriented language or design notation specify the sequences of messages an object will receive. In this latter case, the messages the object will receive is not defined in the description of the object itself, but given by a specification of a context of other objects sending it messages.

This difference in describing which messages objects and actors should get or are willing to receive is also reflected in the difference in definition of the Omicron refinement relation and the actor interaction equivalence relation of (Agha et al. 1993). In (Agha et al. 1993) configurations (components) are collections of actors (objects). The interaction equivalence relation is defined between two actor configurations by comparing their input and output actions. Input and output actions are described by the name of the receiver and the message sent from the configuration (output action) or received by the configuration (input action). For two Actor configurations to be interaction equivalent, the two configurations must have the same number of visible objects and must also know the same number of visible objects from the context. In addition, the two configurations

must have sequences of equal in and out actions where two actions are equal if the receivers and the messages are equal. Messages are equal if both selector and parameter values are equal.

The interaction equivalence relation only compares actor configurations by how they collaborate with actors outside the configuration. This is similar to Omicron's notion of observable actions and only comparing configurations by their observable behaviour.

When an actor is not executing (has no executing methods) and it is selected as the next actor to do something, then the actor will do an input action provided the actor is willing to receive at least one of input action. Since actor descriptions include specifications of the sequences of messages the actor is willing to receive, this is natural to do. In Omicron this is different in that if input-actions should be created from an object's description, then the result would be input actions describing messages for all the slots in the object's slot map hierarchy (or in an object-oriented language with classes: input actions for all the methods in the object's class hierarchy). This corresponds to requiring refinements to have equal observable behaviour to a specification for all contexts.

The main difference between the OCS model as formalised in Omicron and the Actor model in relation to the definition of the refinement relation, is that Actor definitions may be seen as including information about how they expect the context to behave and restrict what communications they are willing to engage in, while Omicron object configurations do not include any such information. Instead this information is found in the definition of objects intended to collaborate with the objects in the component. Furthermore, Actors are monitors while Omicron objects may have any number of executing method objects. These differences results in quite different styles of programming and object/actor descriptions. An interesting topic to pursue is to see if system design is equally different in the two formalisms and what are the strengths and weaknesses of the two styles.

9.1.3 The π-calculus and Omicron

In this section the Omicron calculus is compared in some detail with the π-calculus (Milner et al. 1989a), (Milner et al. 1989b), (Thomsen 1993). First the π-calculus is shortly presented. The concepts of the two languages are compared on the basis of work done on translating parallel Omicron to π and from π to Omicron. These translations are presented in appendix B. Finally, the Omicron refinement relation is compared to bisimulation relations of the π-calculus.

π-calculus syntax and informal semantics:

The π-calculus was created to model mobile processes in order to better understand the semantics of processes and similarity of processes. Below is given a short summary of the π-calculus expressions with informal semantics. For a more detailed description see (Milner et al. 1989a) and (Milner et al. 1989b). The π-calculus is a process calculus in which processes with changing communication structure may be expressed. The processes share communication channels and such channels can be passed on to other processes. Each channel has a name and the only "values" which may be communicated are such names. This means that in the π-calculus "variables", "communication channels" and "values" are all *names*. An infinite set N of names is presupposed, and in the below description, single letters such as x,y,v (possibly with subscripts) range over names. Below the informal semantics of π will be expressed using words such as channel, parameter, etc. to reflect the role the name has in the expression.

The basic building blocks of π-expressions are process expressions. P,Q range over such expressions that are built from the following expressions:

P ::=

$\underline{x}y.P$	output action: send name y on channel x and behave like P[17]
x(y).P	input action: receive an unknown name (say v) on channel x, and then behave like P{v/y} (P with v for y, v must be a new name not occurring in P)
(νy)P	y is a private name for P, making it unique within the total system. y may be passed to other processes so that they can communicate on this channel.
[x=y]P	behave like P if name x is equal to name y else terminate
P | Q	behave as if P and Q act independently in parallel. P and Q may share channels and communicate on these.
P + Q	nondeterministic choice: behave either like P of like Q
0	terminate (usually left out at end of expressions)

The formal definition of the semantics of the π-calculus language is done in the same way as the semantics of Omicron was defined; by giving transition rules. In many ways, the definition of the π-calculus has been used as a pattern when defining the Omicron calculus. Therefore, there are many similarities between the two definitions.

[17] In π-calculus "overscore" is used, instead of "underscore".

Comparing the concepts of the two languages

Process calculi, such as the π-calculus, do not reflect object-oriented concepts. However, it is rather simple to describe the semantics of an object-oriented language using the π-calculus as, eg, done in (Walker 1991), (Walker 1992) and (Nordhagen 1992). Published translations of object-oriented and object based languages to π all follow the same pattern. Also, the translation of parallel Omicron to π is presented in appendix B follows this pattern. It is also rather simple to translate from π to the parallel version of Omicron. Such a translation is also presented in appendix B.

The π language gives names to channels while Omicron gives names to objects and slots within objects. The Omicron language syntax gives clearly defined boundaries between objects. In the π-calculus syntax, processes are similarly distinctly identified. When translating from Omicron to π, object names can be simulated by π channel names and channel names of π simulated by Omicron names. However, When an Omicron object is translated into the π-calculus, the object boundaries become blurred, since a single object becomes more than one process. Similar blurring also occurs when translating from π to Omicron, since one π-process is translated into several Omicron objects.

Communication between different parts of a system is quite different in the two approaches. In Omicron a receiver can never refuse to receive a message. In π a receiver will not take input before the receiver reaches an input sentence. Another difference is that in Omicron a message is always sent to a specific object, while in the process model a sender just puts a message on a channel for anyone to listen to. Omicron may in this relation be seen as having asynchronous message passing while π has synchronous. As is generally known, and as can be seen from the translations between the two languages, one form of communication can be simulated by the other.

The conclusion from the above and the translation of appendix B is that the concepts in one language can be simulated by the concepts in the other. However, the translations show that there is no simple mapping between the two. Below are some comments on the translations.

From Omicron to π-calculus

The translation of Omicron objects into π processes has to take into account slots which store values and inheritance between objects. Slots can be translated in the same way as variables are translated in (Walker 1991) and inheritance is done along the lines of (Nordhagen 1992). The largest part of the translation concerns the translation of slot lookup in an inheritance tree and simulating asynchronous message passing.

When the basic Omicron mechanism for looking up slots and asynchronous message passing are implemented using π, the translation of the different sentences is rather straight forward.

From π to Omicron

As much of the job in translating from Omicron to π involved modelling asynchronous message passing in π, the translation from π to Omicron has a significant part which models synchronous message passing in Omicron. In the simulation of synchronous message passing, slots are used as semaphores. Slots can function as semaphores since the Omicron language allows simultaneous assignment of values to more than one slot, as defined in the IF-rules in the operational semantics in chapter 3.

When the basic π mechanism for synchronous message passing is implemented using Omicron, then translation of the various π constructs are rather straight forward. The most complex translation was the non-deterministic choice (P + Q) where a slot was used as a semaphore to model the non-deterministic choice between P and Q.

Comparing relations between π agents with the Omicron refinement relation

In π there are four kinds of actions:

τ	- silent action
$\underline{x}$y	- free output
x(y)	- input
$\underline{x}$(y)	- bound output

Based on these actions, the π-calculus papers define different relations between process descriptions (π agents).

The simulation relation defined for π allows a variation in determinism. The Omicron refinement relation allows the same. One of the main differences between these two relations is that the publications on the π calculus show no properties similar to the substitution proposition for the simulation relation. Instead bisimilarity is defined based on the simulation relation, below denoted ~, and all presented propositions and theorems are stated and proved for different versions of bisimilar π agents. Bisimilarity does not allow one of the agents to have more possible actions sequences than the other, ie, the agents must be equally deterministic in their behaviours.

There are various versions of bisimilarity, but the most discussed is the strong bisimilarity. Theorem 2 in (Milner et al. 1989b) shows that the following hold for strong bisimulation (where P, Q and R range over π agents):

if P ~ Q then
$\alpha.P \sim \alpha.Q$ where α is a free action
$P+R \sim Q+R$
$[x=y]P \sim [x=y]Q$
$P|R \sim Q|R$
$(\nu y)P \sim (\nu y)Q$
if P ~ Q then P|R ~ Q|R for all R

The last statement is the most interesting property when comparing bisimilarity to the refinement relation of Omicron. It shows that strong bisimilarity is a monotonic relation over all π agents. This statement expresses quite different properties than the properties of the Omicron refinement relation in that the reliable refinement relation is only monotonic relative to Omicron configurations with a specified behaviour.

When translating an Omicron configuration into π and then executed it in π, then all objects will actively listen to their own message reception channels. When the component has done all local actions, then the objects will continue to listen on their message reception channels. The way simulation, and then bisimulation, is defined, gives that when processes listen to channels then the context is expected to eventually send signals on the channel. As an example, assume that we have two configurations, eg, A and B, which are translated to π giving, eg, P and Q. To show P ~ Q would mean to show that P and Q send the same signals on the same channels for all possible inputs from the context. This would mean that A and B must behave similarly for all messages to all objects in A and B, ie, no requirements on the context. This is a quite different property than A being a refinement of B as defined by the Omicron refinement relation. Therefore, even if A is a refinement of B it is not very likely that we have P ~ Q.

From the above it is evident that the refinement relation of Omicron do not map to relations defined for the π-language in a straight forward way. Therefore it is hard to conclude things about object compositionality and reliable substitution from π simulation relations. However several interesting commonalties and discrepancies can be found. Exploring these in more detail would be an interesting topic to pursue.

Behaviour equivalence in the Polymorphic π-calculus

A lot of work have been done on typing π-calculus expressions and on defining various kinds of equivalence classes based on the processes' behaviour. The report (Pierce and Sangiorgi 1996) presents work which combine typing and definition of behaviour equivalence of processes expressed in the polymorphic π-calculus. In this work, the behaviours of a process' observers are specified by using abstract types. The types restrict how the observers may interact with the process. This is analogous to how the reliable refinement relation of this thesis is defined. By the properties expressed in the substitution proposition, the context specification can be viewed as a type specification for all its reliable refinements. In this way the poliadic π-calculus relation which is a behaviour equivalence relation defined relative to the types of observers can be viewed analogous to the Omicron reliable refinement relation defined relative to a context specification.

One very interesting thing to note is that the report (Pierce and Sangiorgi 1996) reports observation of "some surprising interaction between polymorphism and aliasing". Polymorphism stems from the fact that the observers are only known by their types and can therefore have different realisations. Aliasing means that different names used in the process specification can be replaced by one and the same name when the process is run with an actual context. They express this as "A process can always test for inequality between two values of the same type". They formulate the problem they encounter as follows: "When the process's knowledge of the type of the two values is partial, this permits a "leakage of information" that gives receivers of polymorphic communications some unexpected discriminating power." The example they give of this problem resembles the problems encountered in relation to the use of if-sentences in refinements and reliability problems encountered when a component tests if two values hold the names of two different context objects. They further write : "The real significance of these examples of information leakage is not at present clear to us. Nor is it clear whether they can be avoided, e.g., by identifying syntactic or typing restrictions on processes that would guarantee that information leakage cannot occur." They also show examples of similar problems in Standard ML and claim that similar examples can be constructed in any setting with both polymorphism and aliasing. It would therefore be interesting to compare the results in the report with the reliable if-sentence requirement presented here and see if they together can give a better understanding of this problem.

9.2 State Based Functional Models

9.2.1 Objects as collections of functions

Objects can be modelled as collections of functions, ie, as instances of abstract data types (ADTs), initially defined in (Guttag 1980). In this tradition, each object is seen as a set of functions taking values as parameters and returning values. (Danforth and Tomlinson 1988) surveys a number of existing type theories for abstract data types and examines their applicability to object-oriented models. The ADT approach has been applied to different system designs, see, eg, (Lano and Haughton 1994) for some case studies. Properties of ADT objects are specified using, eg, algebraic, type theoretic specification techniques.

Traditionally, when functional models are used as a basis for modelling objects, objects collaborate by one object calling the functions of other objects. Each instance only takes action when one of its functions is called, otherwise it passively waits for a call on one of its functions. This approach limits the types of collaboration structures which can be reasoned about to tree-like structures of callers and callees. The limitation to tree-like collaboration structures is needed in order to avoid the aliasing problem (Hogg 1991), (Hogg et al. 1992) in relation to verification of the instances' behaviours. If the object structure is not tree-like, there are in some cases ways of restructuring the objects and give them additional functionality so that they collaborate in a tree-like structure (Owe 1988). There are also other strategies which reduce the aliasing problem. Some are listed in (Hogg et al. 1992). However, Hogg et.al. claims that there are still no complete solution to the aliasing problem with respect to object-oriented designs.

The functional approach's main focus is on the relation between input parameters of a message and the output values returned from the message - or values returned from messages sent the object at some later time. In general, the algebraic methods define a subtype relation which assure reliable substitution provided (most of) the following hold:

- no aliasing and/or shared variables
- liveness requirement is "return a value" and nothing else
- no instance creation requirements
- single objects are specified although an object may contain internal objects not visible, this corresponds with components only having a single visible object
- no specification of context other than legal messages and (rarely) legal sequences of messages

Some later methods remove some of these restrictions, eg, (Liskov and Wing 1993) eliminates some aliasing requirement and (Briggs and Werth 1994) allow instance creation requirements to be part of a type definition.

However, there are many aspects of object behaviour which can not be modelled and thereby reasoned about using the traditional functional approaches (Nierstrasz and Papathomas 1990b), (Wegner 1993), (Wegner 1994). These approaches focus on object states and functional aspects of objects. They are therefore not well suited for modelling and reasoning about object behaviour which include message passing *from* the objects and creation of other objects.

9.2.2 Examples of functional object models

The list below gives some examples of object models which are based in the ADT tradition.

There is a huge amount of work on specifying the behaviour of objects by defining the relations between input and output values of object functions. These approaches use algebraic specifications, type theory or rewrite logic as their reasoning bases. Objects are modelled as records with associated variables and functions. By using these formalisms it is possible to prove properties like equality of values returned from function calls. Examples of such models are:

(America and Rutten 1990) presents what is (claimed to be) the first language to explicitly include subtyping and inheritance as two completely separate language mechanisms and the subtype relation definition is based only on the externally observable behaviour of objects. Their definition of externally observable behaviour is more than the signatures of the object's methods. The paper present a preliminary formalism in which such properties can be defined: "it states under which conditions a certain message may be sent to the object (possibly constraining the values of the parameters) and what are the possible result values." The language to express such properties has the full power of first order logic and the subtype relationship can therefore not be statically

checked by a compiler. Instead property "identifiers" are introduced to denote these specifications. They explicitly say "A specification exclusively in terms of sequences of messages is clearly infeasible". This is quite the opposite of the Omicron approach.

(Liskov and Wing 1993) define an object's observable behaviour as the values returned from messages. They present a subtype relation which allows multiple, possibly concurrent, users to share mutable objects, ie, aliasing is allowed. They define types based on relations between input and output values, and do not specify actions and liveness properties, except obligations to return values. Their formalism is proof theoretic in that they reason directly in terms of specifications, where as most other approaches are "model-theoretic" in that programmers are expected to reason in terms of mathematical structures like algebras and categories. The formalism in (Liskov and Wing 1993) uses pre- and post-conditions as assertions about the state of the objects, somewhat analogous to Eiffel's (Meyer 1988) use of pre- and post-conditions. However, in Eiffel the pre- and post-conditions are given more operationally as executable boolean expressions about the object's state, rather than as non executable assertions.

(Briggs and Werth 1994) define abstract object types and handle objects, not just values. The language they define is called ObjLog and is sufficiently expressive to specify value-based message passing and instantiation behaviour exhibited by objects defined by sequential object-oriented programming languages. However, there is no support for showing liveness properties, as they only show relations between input and output values in messages to an object of a given type.

(Lano and Haughton 1994) give an overview of many of the algebraic specification methods which claim to be object-oriented such as Object-Z, VDM++, OOZE, MooZ, Fresco, Z++ and Small VDM. The algebraic specification language Larch (Guttag 1993) has also been customised to object-oriented languages such as Larch/Smalltalk (Cheon 1991) and Larch/C++ (Cheon and Leavens 1994). These languages have all the characteristics of algebraic specification methods and are therefore quite different from the Omicron approach.

(Hogg 1991) presents the idea of *islands*. In Omicron terminology, the islands idea is a strategy for controlling the visibility of object names. The strategy eliminate some aliasing problems and thereby give the objects properties which make it easier to prove functional properties of objects.

Other examples are (Leavens and Weihl 1990), (Gunter and Mitchell 1994) and (Abadi and Cardelli 1994)

9.2.3 Traditional functional models are not sufficient for modelling object behaviours

What distinguishes an object in OCS design from an object in a traditional functional model, is that each object has a unique identity, or name, which distinguishes it from all other objects. This is discussed in, eg, (Khoshafian and Copeland 1986). Also, objects continue to exist between executions of their functions, a difference which is pointed out in, eg, (Wegner 1994) and (Wegner 1995).

Peter Wegner's works has inspired much of the present work, eg, the principle of substitutability in (Wegner and Zdonick 1988) and a differentiated view on object typing and language design (Wegner 1987). Later works, notably (Wegner 1993) and (Wegner 1994) address many of the problems Omicron is aimed at solving.

His later works support the below presented view that traditional state based functional models (which Wegner tend to call algebraic models) are inadequate for modelling interacting objects since functions model instantaneous, atomic and serializable algorithms while objects are persistent entities which continue to exist between the execution of their methods and can model concurrent and/or overlapping executions. In (Wegner 1994) Wegner writes (page 3 and 4):

> *"Objects model not only the behaviour of algorithms in their interface, but also the periods of time between the execution of algorithms. Whereas algorithm behaviour is defined only for one input at a time, object behaviour is defined for multiple interacting messages executing in sequence or concurrently. By explicitly modelling persistence and concurrent (overlapping) execution, objects can capture the behaviour of real-time actions in a concurrent world."*

Wegner further writes on the same topic on page 13:

> *"Objects achieve their persistence and temporal modelling power by separating existence and execution, while procedure invocations self-destruct when their execution is completed because they tie*

existence to execution. Separation of existence and execution allows time to be a first class notion and introduce new kinds of (serial and fully abstract) semantics for objects that has no analogue for procedures.

Autonomous existence is the basis not only for persistence but also for concurrency. Persistence implies concurrent existence of the persistent entity and its environment, while concurrent existence in turn provides a framework for concurrent execution. Object composition is based on the notion that objects being composed exist concurrently and can therefore execute concurrently. The composition of procedures aims to capture the effect of their sequential execution and is therefore noncommutative, while the composition of objects aims to capture their concurrent existence and is therefore commutative."

He also supports explicit definition of objects' context to get tractable object specifications (page 10/11) and (on page 31) writes:

"The space of all possible interactions of an object is generally too rich for neat mathematical characterisation. But projections of the interactive computation space such as client-server interaction or particular computation scenarios and use cases[18] *determine tractable subspaces of all possible interactions"*

Also, the importance of incremental modifications and substitutability is supported, eg, by (page 19):

"Because maintenance and enhancement are dominant parts of the software life cycle, resource requirements have as much to do with the capacity of the system to change its behaviour as with the delivery of a given behaviour"

He also argues that algebraic models are aimed at programming in the small, while object models are aimed at programming in the large and better at modelling large software system and writes (page 19):

"Large software systems are non-algorithmic, open and distributed:

non-algorithmic: they model temporal evolution by systems of interacting components
open: they manage incremental change by local changes of accessible open interfaces
distributed: requirements as well as components are locally autonomous."

Wegner gives arguments and motivations for the present work. However, due to the properties, or rather lack of properties, which he says that algebraic models have, it seems that he only consider traditional state based algebraic models. It seems that he does not consider other later approaches to modelling the dynamic behaviours of systems by creating new specialised algebraic formal systems where a notion of time is explicit or implicit. Some such formal systems are mentioned in the following subsection. These formal systems do not have the same weaknesses as traditional state based models.

[18] "Use cases" are defined in (Jacobson et al. 1992). Use cases are used to model system functionality. The description of system functionality is divided into different parts. Each part describe how the system is to respond to a certain kind of user interaction; a use case.

9.3 Other Formal Object Models

This section presents formal approaches which model systems as consisting of a set of objects where each object has a unique name and exists longer than a function call. In all other ways the approaches are different. The creators of these works have backgrounds in traditional theoretical computer science and have applied their skills to reasoning about some property of objects. Before Omicron was created, a number of these approaches were carefully studied. Attempts were made at modelling OCS components and at defining and reasoning about reliable substitution as described in chapter 1 of this thesis. We found the different approaches difficult to use, mainly because their creators have their roots in functional or distributed systems traditions. Therefore, the approaches focus on properties which are commonly the focus of these traditions and therefore what one traditionally reason about - and therefore have developed mechanisms to reason about. This is different from approaches such as Omicron which focuses on the properties which component developers see as important.

9.3.1 Approaches using specialised logics

Various attempts at modelling objects by various logics have been made. Bellow are some examples with some short comments related to what aspects of objects are described:

Modal logic and objects:

(Morzenti and Pientro 1991) State based, no instance creation and messages

(Wieringa et al. 1994) Specify changes in the object's roles over time, no messages or instance creation

Temporal logic and objects:

(Arapis 1992) Behaviour based.
Describes sequences of in and out messages. In many ways a similar object component behaviour model as Omicron, but no shared variables, inheritance or instance creation. Verify consistency of specifications and monitoring adherence to the specification during run-time. Does not consider problems related to reliable substitution.

(Fiadeiro and Maibaum 1990) State based. Reasons about object behaviour in terms of proving properties of the object's attributes.

Rewrite logic and objects:

(Meseguer 1993) A specification language named Maude for specifying concurrent objects.
Focus on defining clear semantics for the specification language, rather than showing properties of the specifications Objects are modelled as records with associated functions. It is possible to prove properties like equality of values returned from function calls.

Deontic logic and objects:

(Reghizzi and Paratesi 1991) Specification by defining constraints on method activations.
Use Petri nets and deontic logic.

9.3.2 Approaches using traces

Trace modelling have been used by many researchers since the early 70ies, most notably people such as Kahn, Hoare, Dahl and Broy. Various attempts at modelling objects' behaviours by using traces have been made. Bellow are some examples with some short comments:

(Nierstrasz and Papathomas 1990a) A general discussion on trace based relations between objects, based on traces consisting of messages. Also, relate such specifications to CCS. This work only presents definitions and no propositions or proofs of properties.

(Ehrich et al. 1990) Compare traces consisting of in-messages and observations of variable values. Response messages are not considered.

(Skuce and Mili 1995) Traces specify objects' observable behaviour. The behaviour consists of sequences of messages. The trace notation specify the class of the object receiving each message and in the presented examples parameters in messages are values.

These trace based approaches share many of the same ideas as presented in this thesis. However, they only present specification notation, while there are no discussions or reasoning about substitutability of components. They all have static network topologies since they do not consider messages containing object names. Also, they do not include object creation actions.

9.3.3 Demeter-Contracts

The paper (Helm et al. 1990), (Holland 1992) describe "Contracts" as a way of capturing multi object behavioural collaboration, but does not provide a precise semantics for Contracts. ObjChart (Gangopadhyay and Mitra 1993) is a development of the Contract ideas and give precise semantics to object collaboration patterns. ObjChart is a graphical notation ("visual formalism") for defining object behaviour based on observable actions. Their definition of refinement is based on legal changes to a given chart which yields a refinement behaviour description. ObjChart as presented in the paper is based on fixed networks of components and cannot handle the semantics of dynamically created objects. Otherwise their notion of refinement corresponds with the definitions used in this thesis.

9.3.4 ABEL

Ole-Johan Dahl and Olaf Owe have developed the ABEL language (Dahl and Owe 1991), (Dahl 1992), (Dahl and Owe 1998). The applicative core of ABEL is a strongly typed first order expression language with main elements being variables, functions and types. This allows abstract requirement specifications. Using the imperative class concept of ABEL, a more low level concrete module can be proven to simulate an abstract module and in such cases the abstract module may be used as an abstract specification of the more concrete module. In general, this is used to specify safety aspects of an implementation and value manipulation of functions.

ABEL allows reasoning about messages being received by and sent from objects. This is done by including fictious history variables in the specifications. These variables represent traces of messages to and from an object or a collection of objects. This gives a rely/guarantee formalism with abstract specification (and prototyping) based on history variables.

9.3.5 POBL (or $\pi o \beta \lambda$)

C. B. Jones has developed a design notation named $\pi o \beta \lambda$ and used it to reason about object-based programs (Jones 1995). Jones has chosen to focus on a different set of object-oriented concepts than the present notation. For example $\pi o \beta \lambda$ has classes for object creation and each object can only have one method executing at the time, while Omicron uses objects as templates for object creation and any number of methods can execute at the same time. The main difference is, however, that $\pi o \beta \lambda$ is typically used to reason about the state of shared variables while Omicron is created for reasoning about substitutability of objects with reactive behaviour such as message passing and object creation.

9.3.6 Other approaches

Other approaches which express properties of objects use state transitions (Barbier 1992), (Andersen and Reenskaug 1992), (Andersen 1997), object and object types as values (Dami 1993) etc. Most of these efforts are found to be in the same direction as the presented work, but has not been adequately worked out to give any useful support for reasoning about requirements necessary to Ensuring reliable substitution.

9.3.7 Object-oriented languages

Examples of object-oriented languages are Smalltalk, CLOS, C++, Simula, Eiffel, SELF, Beta and Java. As argued in earlier chapters, Omicron and object-oriented programming languages have much in common. The communality is related to expressing object behaviour, and not reasoning about such behaviours. Below, the Omicron language is compared with some object-oriented programming languages.

Prototype languages like SELF have many similarities to Omicron. The main difference is that SELF is sequential and has a more user friendly syntax. To show similarities of SELF and Omicron, an attempt has been made at translating SELF to the sequential Omicron presented in chapter 7. This translation was rather straight forward.

CLOS also share commonalties with Omicron. However, a major difference is that CLOS has multiple message dispatching and programmable language features through the meta object protocol. A further study into such features, relating them to reliability and reliable substitution would be interesting.

It is interesting how Beta resembles Omicron when taking Omicron and Beta's different histories into account. Beta was developed generalising Simula's idea of active objects working as coroutines and focused on flexible, but strong typing, while Omicron was developed generalising the idea of objects sending messages which are bound dynamically to methods with runtime typing, typically found in SELF. Beta is a representative of the strong compile time type checking tradition while (eg) SELF is a representative of a "lassies faire" tradition.

Beta does not have prototypes, but has the homogeneity of active objects and executing methods. Beta is strongly typed based on signatures but due to the properties of the typing system, great flexibility in the code can be achieved, as long as it is planned when the component is programmed, ie, before it is compiled. SELF (and Omicron) is untyped and planning is therefore unnecessary. The block structure of Beta can be mimicked by using object-inheritance in SELF and Omicron. When looking into these languages the basic primitives of Beta is quite similar to Self and Omicron, apart from two things:

- The most significant difference is that in Beta there are explicit constructs to start the execution of an object without sending another object a message as one must in Omicron.

- To create a new object in Beta one instantiates a pattern which can not be changed during execution, ie, it will remain as initially defined, while in SELF and Omicron a new object is created by cloning an object which may have changed since it was initially defined.

An interesting topic to pursue is to include new aspects of object-oriented languages and systems in Omicron. An interesting goal for such an effort would be to find new reliability requirements and through these findings get a better understanding of how to make reliably substitutable components in languages with such features. It could also lead to new language features or new combinations of features which could ease the creation and maintenance of extensible systems.

9.4 Assumption / Guarantee Specifications

An assumption/guarantee specification, introduced in (Jones 1983), asserts that a system $\prod$ provides a guarantee M if its environment satisfies an assumption E. This corresponds to the following expression using the refinement relation of Omicron:

$$\prod \leq_E M \wedge F \leq_M E \Rightarrow \prod \leq_F M$$

where F is the environment.

Manfred Broy, Ketil Stølen and others, have worked with assumption/guarantee specifications for describing components which collaborate by communicating values over channels, see eg, (Broy 1996), (Broy 1997) and (Stølen 1996). They have developed mathematical models for making assumption/guarantee specifications and show properties of their models such as safety and liveness aspects of specifications, completeness and relationships between their model and the stepwise development of specifications. They also relate their work to objects. Their model of objects is rather simple compared with the Omicron model in that they view object collaboration as communication of messages which are values. This gives a static topology. There is no object creation or shared variables. These differences between the Omicron model of OCS components and this object model are quite substantial. Further work is therefore needed to see if and how their results are applicable to OCS components.

Martín Abadi and Leslie Lamport have worked on showing properties related to composition and decomposition of components specified by assumption/guarantee specifications. They have formulated a composition principle (Abadi and Lamport 1993) and a composition and a decomposition theorem (Abadi and Lamport 1995). Below, these are compared to the reliable substitution property of the reliable refinement relation. Abadi and Lamport present a state-based object model and use this model to exemplify their composition principle and theorems. This state based model define a state as a set of variables holding values and an action as a change of one or more of the variables' values. This is quite different from the Omicron model of OCS components. However, their composition principle and theorems have interesting relationships with properties of the reliable refinement relation defined for Omicron configurations.

9.4.1 The composition principle

Abadi and Lamport discuss composition of specifications in their paper (Abadi and Lamport 1993). In the paper they present the Composition Principle which, at the semantic level, is independent of any particular specification language or logic. They write:

> *The fundamental problem of composing specifications is to prove that a composite system satisfies its specification if all its components satisfy their specifications.*

They denote a specification of an interactive system $\prod$, and asserts that the system guarantees property M only under the assumption that the environment satisfies some property E. The Composition Principle is presented as follows:

> *The Composition Principle:*
>
> *Let $\prod$ be the composition of $\prod_1$,…, $\prod_n$, and let the following conditions hold.*
>
> *(1) $\prod$ guarantees M if each component $\prod_i$ guarantees M_i.*
>
> *(2) The environment assumption E_i for each component $\prod_i$ is satisfied if the environment of $\prod$ satisfies E and every $\prod_j$ satisfies M_j.*
>
> *(3) Every component $\prod_i$ guarantees M_i under environment assumption E_i.*
>
> *Then $\prod$ guarantees M under environment assumption E.*

To give a concrete example of this principle, Abadi and Lamport present a state-based approach to system specification and restate the composition principle for a case n=2, in terms of their formal definitions. This

restatement takes the form of a proof rule for showing that the composition of the two components' specifications implements the system's specification. Their main theorem shows that their proof rule is sound.

Below, we restate the composition principle for a case of n=2, in terms of the Omicron formal definitions and show that the composition principle holds also in this case.

We assume that we have a specification which says that an interactive system (in Omicron terms: a component) will guarantee M provided the environment satisfies E. An implementation of such a system is denoted $\prod$. For $\prod$ to guarantee M under environment assumptions E, the implementation $\prod$ must be a reliable refinement of the specification M relative to an environment with behaviour E. The Composition Principle must therefore give:

$$\prod \leq_E M$$

We further assume that we have two components which guarantee M_1 and M_2 respectively provided their environments satisfies E_1 and E_2 respectively. The composition principle can the be restated as follows (we ignore the substitutions for simplicity):

(1) This condition asserts that $M_1||M_2 \leq_E M$

(2) The environment of $\prod_1$ will consist of $E||M_2$ and we must therefore show that this environment is a reliable refinement of E_1, ie, $E||M_2 \leq_{M1} E_1$. Similarly we must show that $E||M_1 \leq_{M2} E_2$.

(3) This condition means asserting that the implementations fulfil their specification. This means asserting that implementations of the two component making up $\prod$, here denote $\prod_1$ and $\prod_2$ are reliable refinements of the specifications:

$$\prod_1 \leq_{E1} M_1 \qquad \text{and} \qquad \prod_2 \leq_{E2} M_2$$

To show that the Composition Principle hold, we must show that provided the assertions in (1), (2) and (3) hold, then we have $\prod \leq_E M$, where $\prod = \prod_1||\prod_2$. This means showing the following proposition:

$$M_1||M_2 \leq_E M \wedge E||M_2 \leq_{M1} E_1 \wedge E||M_1 \leq_{M2} E_2 \wedge \prod_1 \leq_{E1} M_1 \wedge \prod_2 \leq_{E2} M_2 \Rightarrow \prod_1||\prod_2 \leq_E M$$

By the simple substitution proposition for systems with two components, theorem T.6.1, we have:

$$\prod_1 \leq_{E1} M_1 \wedge E||M_2 \leq_{M1} E_1 \Rightarrow \prod_1 \leq_{EM2} M_1 \quad \text{and}$$
$$\prod_2 \leq_{E2} M_2 \wedge E||M_1 \leq_{M2} E_2 \Rightarrow \prod_2 \leq_{EM1} M_2$$

Because of reliable substitution, theorem T.6.4 we further have:

$$\prod_1 \leq_{EM2} M_1 \wedge \prod_2 \leq_{EM1} M_2 \Rightarrow \prod_1 \leq_{E\prod 2} M_1 \wedge \prod_2 \leq_{E\prod 1} M_2$$

and also:

$$\prod_1 \leq_{E\prod 2} M_1 \wedge \prod_2 \leq_{E\prod 1} M_2 \Rightarrow \prod_1||\prod_2 \leq_E M_1||M_2$$

Because we have $M_1||M_2 \leq_E M$ from (1) and the reliable refinement relation is transitive, observation P.5.5.3, we have:

$$\prod_1||\prod_2 \leq_E M_1||M_2 \wedge M_1||M_2 \leq_E M \Rightarrow \prod_1||\prod_2 \leq_E M$$

We then have $\prod \leq_E M$. This shows that the reliable substitution property follows the Composition Principle of Abadi and Lamport.

In the referenced paper, the principle was exemplified and illustrated by a formal model which is very different from Omicron. It is different in that it is state based where as Omicron is action based. This difference causes refinement relations and other definitions to be quite different in the two formalisms. In spite of these differences, the Composition Principle also holds for reliable substitution of Omicron components. This shows the generality of the Composition Principle. In addition, the Composition Principle expresses an important property of reliable refinement relations.

9.4.2 Composition and decomposition

Abadi and Lamport show decomposition and composition theorems for assumption/guarantee specifications within TLA (Abadi and Lamport 1995). In TLA, the Temporal Logic of Actions (Lamport 1994), a state is an assignment of values to variables, and a behaviour is an infinite sequence of states. Syntactically TLA is built up from state formulas using Boolean operators and three special operators. An action is a Boolean-values expression which describe how values of variables are changed. As this shows, TLA has focus on the values of the variables of a system. A specification typically makes statements of the relationship between a set of input variables e and a set of touples m of output variables. In this way, TLA is used to reason about relationships between input and output values of systems. This is different from the Omicron formal model which focus on message sending and object creation. In spite of this difference, there are strong relationships between the properties of assumption/guarantee specifications using TLA and properties of the reliable refinement relation. To illustrate this relationship, two theorems which corresponds to the general decomposition theorem and the composition theorem of (Abadi and Lamport 1995) will be formulated and shown for the Omicron reliable refinement relation.

T.9.1 The general decomposition theorem

This theorem is formulated on the background of decomposing a complete specification into a number of parts. For each part there will then be a lower level specifications. The system and each low level specification is specified using assumption/guarantee specifications. The general decomposition theorem says:

Let $E \Rightarrow M_1||\ldots||M_n$ be the system specification which says that
the system guarantees $M_1||\ldots||M_n$ if the environment satisfies assumption E
let $E_i \Rightarrow N_i$ be the lower level specifications for i = 1,..n

Then we have that
(1) if the environment assumption E_i is satisfied by E and all N_j where $j \neq i$ and
(2) N_i implies M_i under the assumption that the environment satisfies E_i
then
$N_1||\ldots||N_n$ will guarantee $M_1||\ldots||M_n$ under the assumption that the environment satisfies assumption E

This can be formulated using the Omicron reliable refinement relation as follows (we ignore the substitutions for simplicity and let M-i denote all M_j except M_i):

If, for i = 1,…n,
(1) $E||M\text{-}i \leq_{Mi} E_i$
and
(2) $N_i \leq_{Ei} M_i$

then
$N_1||\ldots||N_n \leq_E M_1||\ldots||M_n$

This can be shown by using the reliable specification theorem where we get

$$\forall i \in \{1..n\} : E||M\text{-}i \leq_{Mi} E_i \wedge N_i \leq_{Ei} M_i \Rightarrow N_i \leq_{E||M\text{-}i} M_i$$

and then we have

$$\forall i \in \{1..n\} : N_i \leq_{E||M\text{-}i} M_i \Rightarrow N_1||\ldots||N_n \leq_E M_1||\ldots||M_n$$

This shows that the Omicron reliable refinement relation shares the decomposition property with TLA assumption/guarantee specifications.

T.9.2 The composition theorem

This theorem is formulated on the background of composing a system specification from a number of lower level specifications. The system specification and each low level specification are done using assumption/guarantee specifications. The general composition theorem says:

Let $E \Rightarrow M$ be the system specification which says that
the system guarantees M if the environment satisfies assumption E
let $E_i \Rightarrow M_i$ be the lower level specifications for i = 1,..n

Then we have that

(1) if the environment assumption E_i is satisfied by E and all N_j where $j \neq i$ and
(2) $M_1||\ldots||M_n$ implies M under the assumption that the environment satisfies E
then
when we have
if N_i will guarantee M_i under the assumption that the environment satisfies E_i
then $N_1||\ldots||N_n$ will guarantee M under the assumption that the environment satisfies E

This can be formulated using the Omicron reliable refinement relation as follows (we ignore the substitutions for simplicity and let M-i denote all M_j except M_i):

If, for i = 1,…n,
(1) $E||M\text{-}i \leq_{Mi} E_i$
and
(2) $M_1||\ldots||M_n \leq_E M$

then
$\forall\, i \in \{1..n\} : N_i \leq_{Ei} M_i \Rightarrow N_1||\ldots||N_n \leq_E M$

From the decomposition theorem we get:

$\forall\, i \in \{1..n\} : E||M\text{-}i \leq_{Mi} E_i \wedge N_i \leq_{Ei} M_i \Rightarrow N_1||\ldots||N_n \leq_E M_1||\ldots||M_n$

Since we have (2) and the reliable refinement relation is transitive by proposition p.5.5.3 we have:

$N_1||\ldots||N_n \leq_E M_1||\ldots||M_n \wedge M_1||\ldots||M_n \leq_E M \Rightarrow N_1||\ldots||N_n \leq_E M$

This shows that the Omicron reliable refinement relation shares the composition property with TLA assumption/guarantee specifications.

CHAPTER 10

Conclusions and Further Work

This chapter gives some concluding remarks.

In section 10.1 the contributions of this thesis are highlighted and evaluated with respect to the initial goals.

Section 10.2 sums up remaining problems and issues for further work. Section 10.2.1 and 10.2.2 present some ideas of how to develop the theory so that it may become applicable to other aspects of extensible systems and object-oriented modelling. Section 10.2.3 discusses how to combine the presented model with other models such as functional and process models. Section 10.2.3 presents ideas for further work on applying the theoretic results to enhance development methods and processes.

Section 10.3 draws some final conclusions.

10.1 Main Conclusions

The main focus of this thesis is an investigation of substitutability of components in object component systems (OCS). This investigation has lead to the following conclusions:

Conclusions from chapter 1:

Object component systems has a set of characteristics distinguishing them from other kinds of systems. The main characteristics are:

> An object component system is viewed as consisting of *objects*, where each object is associated with a name distinguishing it from all other objects in the system. An object has variables and methods. To make a system of objects more manageable, objects are grouped into *components*. In some cases a single object will be a component, but in general a component will consist of several objects. Components form *dynamic graph-like collaboration structures* as objects in different components send messages to each other, create objects from templates in other components and update variables in other components. An object do not move from component to component. A component may be a client of the other components, the server for other components or both clients and servers.

There are also some special characteristics of how object component systems are specified and designed. The main specification and design characteristics are:

> An object component system specification from a closed system when users and external devices are modelled as objects with *non-deterministic behaviour*. The system is closed in the sense that the objects in the system specification only collaborate with each other.
>
> Components are specified by their *observable behaviour* which typically include actions which send messages to objects in other components, actions which create new objects from templates in other components and actions which update shared variables. We call a component whose observable behaviour is similar to the observable behaviour described in a specification, a *refinement* of the specification. If specifications have non-deterministic behaviour, then a refinement may have a more deterministic behaviour than the specification and still be seen as having a similar[19] observable behaviour. When implementations are seen as specifications, there may also be refinements of implementations.
>
> Usually all components in a collaboration are designed at the same time. This is done since the quality of a component's design is judged by the flexibility and simplicity of the total collaboration pattern, not just the design of an individual component.
>
> A component's *observable behaviour is specified relative to a context*. The context consists of other components. A component's specification will also include the specification of the observable behaviour of the component's context. A component specification may then be viewed as both a specification of the component and as the specification of the context. In the latter case the objects which were originally found in the context are viewed as one or more components and the objects in the original component becomes a part of these components' context. There is therefore *a symmetry of component and context*.

There are also some particular ways in which components are used and manipulated:

> There is component and context symmetry also in that both the component and the context may be substituted with components / contexts which have similar observable behaviour.
>
> When creating a design it is presupposed that new versions of the components will in general be created separately from each other in space and/or time. The underlying idea is that it should be possible to define standards and have a market for components.

At the centre of component substitutability lies a refinement relation which define similarity of components. The refinement relation is partial in that similarity of components is defined relative to a context of other components. When a component is to be a refinement of some other component, here called a specification, the refinement has similar observable behaviour to the specification. A specification may define a rich set of alternative behaviours and a refinement does not have to display all of the alternative behaviours to have similar

---

[19] "similar" is here used to denote a non-symmetric relation. This is also done in (Milner et al. 1989a) and others. "bisimilar" is in this tradition used to denote a symmetric relation.

observable behaviour to the specification. Therefore, similar behaviour is defined so that the following must be fulfilled for a component to be a refinement of a specification relative to a context:

> For each possible action sequence from the system where the specification is substituted with the component, there will be a sequence of observably similar actions from the system without the substitution.
>
> Observably similar actions are actions which are of the same kind; message send, object creation, assignment or error, and similar message send actions are similar messages to the same objects in the context of the component and specification, similar object creation actions creates objects from the same template found in the context and similar assignment actions update the same variable in the context with similar values.
>
> Similar error actions can be defined in various ways and a simple error model was chosen which defines two error actions to be observably similar if they are errors from execution of the same object in the context. Errors in the refinement and specification components are not observable actions and therefore will not be involved in determining if a refinement and its specification have observably similar behaviour.

Since any component may have more alternative behaviours than its refinements, then a component it not necessarily a refinement of its own refinements. Therefore the refinement relation is not symmetric. However, it is reflexive and transitive and therefore a pre-order.

To avoid unanticipated system behaviour, it is important that similarity of components is maintained when the components in the *context* is substituted with similar components. The term *reliable substitution* means such a substitution where similarity is maintained. The reliable substitution property was expressed in the central proposition of this thesis which was called *the substitution proposition* which expresses a monotonicity-like property for refinement relations.

Chapter 1 showed relationships between reliable substitution and reuse of components, maintenance of extensible systems and development of large systems. This showed that when components and component specifications have the properties expressed in the substitution proposition, the components are easier to reuse, extensible systems are easier to maintain and it can lead to more efficient development of large systems.

Conclusions from chapter 2:

Chapter 2 presented the classical example of object component system design: the Model-View-Controller Framework (MVC). It was shown that this design and its use in practice have the characteristics presented in chapter 1. In addition, similar actions and similar observable behaviour was presented in detail and illustrated with examples taken from the Model-View-Controller framework.

Conclusions from chapter 3:

Chapter 3 introduced the object-oriented language Omicron and showed how the simple concepts of slot, object and three kinds of executable sentences could model variables, method tables, executing message, message sending, inheritance, object creation, dynamic self-binding and most other concepts common to object-oriented languages.

Conclusions from chapter 4:

In chapter 1 and 2, similar observable behaviour was informally defined. In chapter 4 formal definitions of observable similarity was given for Omicron components. "Similar components" was formally defined by a refinement relation. It was shown how this definition corresponds with the informal definition from the previous chapters.

Conclusions from chapter 5:

The main conclusion from chapter 5 is that the definition of "similar components" given in chapter 1, 2 and 4 does not give reliable substitution of components. It is therefore necessary to introduce requirements on how to define configurations which are to be reliable refinements and also to strengthen the refinement relation of chapter 4. A central concept in getting reliable substitution of components is *the visible objects of a component*. A visible object is an object which is found in one component but known to the component's context.

It is important that the reliable refinement relation is not too strong, since this will set unnecessary restrictions on implementors of specified components. Chapter 5 therefore argues that the presented reliability requirements are strictly necessary. The reliability requirements can be summed up as follows; here stated informally to avoid formal notation at this stage:

No external inheritance: An object in one component can not inherit from an object in another component

Reliable method lookup: When an object receives a message, a method must be found within the component of the receiver.

Reliable message sending: A method must always be found in the context when there is a message from the component to an object in the component's context. This can be interpreted as:

1) a reliable specification must specify the context's behaviour for every message from an object in the component to an object in the context.
2) a reliable refinement must not send more messages to the context than its specification does.

Reliable if-sentences: When the object names in two variables are compared in an object in a component, then both values can not be names of objects in the component's context. (Alternatively, a component and its refinements must have the same number of visible objects.)

In order to get reliable substitution of components it is also necessary to set some requirements on the use of object names and to strengthen the refinement relation of chapter 4. In the new refinement relation, called refinement with specialisation, there are additional requirements on the similarity of actions from execution of sentences in the context which are observed by the component and its specification. There are also additional requirements on the use of names of visible objects in observably similar actions. This strengthening results in requirements on how to make reliable specifications and on relationships between the visible objects of a refinement and its specification. How to make reliable specifications and refinements are topics of chapter 8 and conclusions from this chapter are presented further below.

Conclusions from chapter 6:

Chapter 6 showed that the reliability requirements of chapter 5 are sufficient for showing the substitution proposition for the refinement relation with specialisation defined in chapter 5. This is shown in the central theorem of the thesis, theorem T.6.3. Other theorems in chapter 6 shows that we can reliably substitute all or just some of the components in a system with their reliable refinements. The last section of this chapter argues that traditional class libraries are reliable since they are not changed or substituted during runtime.

Conclusions from chapter 7:

Chapter 7 presented a sequential versions of Omicron with expression evaluation semantics of sentence execution. This is opposed to the parallel version with atomic operation semantics presented in the foregoing chapters. The main conclusion from this chapter was that the reliability requirements of chapter 5 also apply to components defined by using sequential Omicron. It can also be assumed that similar requirements are necessary when using any language to define OCS components which are specified by the component's observable behaviour consisting of sequences of message send actions, object creation actions and assignment actions.

Conclusions from chapter 8:

Chapter 8 relates the results of the foregoing chapters to practice. Advice on how to make reliable specifications and reliable refinements were given. These advice were:

Reliable specifications of a set of collaborating components must follow the rules:

- the behaviour of the components in a system are described as the set of all possible sequences of actions from execution of the system. The action sequences contains actions which are messages to objects sent from one component to another component and actions which are creation of objects where the executed sentence is in one component and the template used for object creation is in another component.

- for each component describe the maximum number of visible objects which refinements of the component may have.

- for each visible object describe where it is found in the action sequences describing the system components' behaviour. Visible objects are used as message receivers and templates for object creation and their names can appear as parameters in messages.

- for each component describe which visible objects from other components are initially known

It was pointed out that when classes are parameters in observable messages or initially known to other components, they must be treated as visible objects.

In addition, configurations which are to be specifications for which there should be created reliable refinements should follow the same rules as reliable refinements. These are the reliability requirements presented below. They are not strictly necessary to get reliable specifications, but when the reliability requirements is not followed in the specifications, the specifications becomes more complex than necessary and do things which can not be done (or copied) in any reliable refinements of the specification.

Reliability requirements for a component which is to be reliable:

- can not inherit variables or methods from objects in the component's context
- can only have methods which are local to the component
- can not compare the names of objects which are not in the component
- can only refer to class names within a component
- can not send messages to context objects unless methods are found for the messages
- must have the same number or fewer visible objects than the specification and the names must be used as specified
- when a refinement has fewer visible objects than its specification, then one object shall take the place of two or more of the specified visible object. The same refinement object must always be used in place of the same specification visible objects.

Here are some advice giving suggestions on what to do instead of breaking the reliability requirements:

> If a component is to be a reliable refinement of some specification, then the no external inheritance reliability requirement limits the use of shared variables to sharing variables within a component. Instead, when a component wants some data from a collaborator it should send a message which returns with the data.
>
> Give priority to delegating functionality to a component rather than inheriting it from a class. It is not possible or meaningful to have superclasses as components and at the same time have reliable substitution of such components unless a different and less abstract style of specification than the one presented in this thesis is used.
>
> Methods in other components should not be changed during execution. Use such things as Smalltalk block-objects and/or the Visitor pattern when you want a component which should have different observable behaviour depending on which component it collaborates with.
>
> Comparing object names should be avoided unless refinements are expected to have exactly the same number of visible objects as defined in the specification, for instance (or typically) infinitely many visible objects. Instead of using if-sentences it is better to redesign and use dynamic binding. If object names must be tested, then at least one of the names must be local to the component where the test is done.

One conclusion which was drawn is that there are correspondences between component developers advice on how to make reusable components and the practical consequences of the reliability requirements. This can be taken as an indication that the Omicron framework and the definitions of observable behaviour and the refinement relation are formalisations of important aspects of components and component similarity as found in the OCS design tradition.

Chapter 8 also explained why a reliable specification may be defined as a trace set consisting of a set of sequences of message send and object creation actions. This is an alternative to giving an operational specification through a set of configuration expressions which are executable descriptions of the components of a system.

Chapter 8 showed that the practitioners' advice for making components does not present a complete understanding of necessary reliability requirements. One of the new understandings is that a reliable specification must describe how objects from one component become visible or known to the objects in other components. This means that when a designer makes a reliable specification s/he can not "abstract away" the number and use of visible objects. The reliable if-sentences requirement and the fact that message selectors as parameters creates reliability problems also gave new understanding on how to ensure reliable substitution. The new understanding may hopefully facilitate the definition, control, management and use of more complex - and is in some ways easier to understand and/or reuse - components. This in turn might make it feasible to handle more complex components with richer functionality without introducing uncertainty and doubt whether the finished system will work or not.

Conclusions from chapter 9:

Chapter 9 gives an overview of related work in formalisation of computer systems. There are mainly two directions: formalisation of processes and of functions. Both these have been extended with ideas taken from object-oriented languages. Various formalisation techniques are used such as computational frameworks (such as λ and π calculi), rewrite systems, algebraic specifications and temporal logic. None of these formalisations define OCS components as described in the presented thesis.

There are many alternative ways of specifying software and defining relations between specifications and between a specification and implementations of the specification. Most related work focus on total equality relations which are quite different from the refinement relation defined in the presented thesis. The main difference is that the refinement relation is defined relative to a context and is therefore partial. (Also, the refinement relation is not symmetric and is therefore only a pre-order and not an equality relation.)

The ideas behind the refinement relation are also reflected in what is called assumption/guarantee specifications. Such a specification asserts that a system Π provides a guarantee M if its environment satisfies an assumption E. This corresponds to the simple substitution proposition which defines reliable substitution for systems with two components. Martín Abadi and Leslie Lamport have worked out a Composition Principle for assumption/guarantee specifications. Even though their component model is quite different from the Omicron component model, their Composition Principle has interesting relationships with properties of the reliable refinement relation defined for Omicron configurations. This shows the generality of the Composition Principle. Also, the Composition Principle expresses an important property of reliable refinement relations.

An interesting observation is that existing formal works on objects and components in distributed systems do not touch the ideas represented by the reliability requirements, while chapter 8 shows that informal traditions of practising software engineers present intuitive understanding of many aspects of the reliability requirements.

10.2 Further Work

Even though the presented theory already has been able to form a theoretic basis for practical advice for component design, the theory can be further developed to cover more of the aspects and problems related to object component systems. Challenges are for example other error models and modelling of various features which are found in object-oriented languages but which are not covered by Omicron. Another challenge is to apply the presented theoretic results in practice. Many different such challenges have been discussed in different previous chapters and many topics have been mentioned and left for further work. Below, some of the most interesting and important challenges and topics are presented.

10.2.1 Other languages and rules of action

When working to find a way to formalise OCS components, many alternative languages and rules for describing the semantics of the language were attempted before the two presented versions were selected. Most alternatives became very complex and the conclusion is that only two versions were interesting to present. The two versions reflect the two basically different ways of viewing the execution of a sentence in a program: a sentence may be seen as an atomic operation to be performed, or as an expression which is to be evaluated. However, there may be cases where it is important to have both sentence semantics in one and the same language and where a formal model of the language should reflect this. Such languages would then hold rules for both kinds of actions and therefore the semantic definition would include more rules than any of the languages presented in this thesis.

As mentioned in previous chapters, the present versions of Omicron is rather cumbersome to use when defining components holding sets of objects and/or doing arithmetic and other kinds of calculations. In practice, these are important properties of many components, and a formal language which made specification of such properties simpler then Omicron as presented here, would be advantageous. Incorporating syntax and formal semantics of constructs which define set-operations and calculations would therefore give a more complex, but more practically useful specification language than the Omicron languages presented here.

As explained at the end of chapter 3, transition rules defining the semantics of languages are usually organised differently than what has been done in this thesis. The main difference is that the rules also define transitions of parts of systems, not only whole systems as is done in chapter 3 and 7. At the end of chapter 3 is was explained why this traditional way of organising the rules was not followed, as it created substantially more rules and more complex rules. However, there might be at present unknown advantages of using the traditional organisation and this should be looked into in more detail.

As also discussed at the end of chapter 3 there are other way of defining the semantics of a language than using transition rules. Other alternatives should also be looked into as they might give new insight into the area of object component systems.

10.2.2 Other refinement relations and substitution propositions

The reliability requirements defined in chapter 5 only applies to configurations which are reliable refinements and specifications as defined by Omicron and the substitution proposition. If one or more of the definitions are changed, the reliability requirements may not be necessary and/or new requirements must be introduced to get reliable substitution. What follows below are some examples of how the definitions may be changed.

New definition of component similarity

New view on similar actions

The presented definition of similar actions may be viewed as an example. The example was chosen since it seems to correspond well to most component developers view of similar components. However, there are alternatives, but most seems rather exotic. For instance:

- two message send actions may be similar even when the message selectors are different
- two message send actions may be similar even when the receivers are different, but in the same component
- two actions may be similar even when they stem from execution of different sentences in the observer, or one action is from an object in the observers and one is from an object in the component
- one message-send action may be replaced by one or more message-send actions. This is typically done when detailing designs

If the definition of similar actions is changed, the requirements necessary to get reliable substitution will in most cases change. This is left for further study.

New view on where to place new objects
In the above, new objects were placed in the part of the system where the template object was found. There are other alternatives. For instance, the object may be placed in the part of the system where the object creation sentence which created the object is found. This will alter how components develop over the time of the system's execution and will therefore influence how components are perceived. This will change the definition of similar components and also probably change reliability requirements. This is left for further study.

Objects moving between components
In the present work it was assumed that any object belonged to the same component at all times. An alternative would be to allow objects to move between components. This would change the status of an object from observed to observable or vice versa. This is left for further study.

A traditional monotonic relation

A traditional monotonic relation defines similarity of components relative to any context the components might be placed in. This can be expressed using the refinement relation of chapter 4 as follows:

$$\forall D : A \leq_D B$$

saying that for any D, A||D and B||D will have similar observable behaviour relative to D

A consequence of this is that A and B must be similar for any sequences of messages and any number of visible objects used in all possible ways. The reliability requirements of chapter 5 would be necessary, but far from sufficient to ensure this kind of traditional monotonicity properties of reliable refinements and specifications. Finding reliability requirements in such cases is left for further work.

Investigate other error-models

The Omicron error model is rather simple in that whenever a sentence might give an action which is not meaningful, eg,

- message to a non-existent object,
- cloning of a non-existent object,
- sending a message to an object where there is no method for the message
  ie, a message not understood error

then the object with the erroneous sentence terminate. All errors lead to the same kind of error action and error actions are not observable from others than the component where the sentence which gave the error action was found.

As noted in chapter 8, the reliable message sending requirement was linked to the error model in Omicron. However, as noted in chapter 8 and in the discussion in section 5.3.3, the only other natural alternative is to require reliable message sending for the specification components. This in turn indirectly ensures reliable message sending in the refinements. Therefore, reliable message sending is still a requirement for refinements, but it is ensured indirectly.

The other reliability requirements are not so directly linked to the error model, as reliable message sending is. However, there might be other error models which can alter the reliability requirements. Finding such error models is left for further study.

Other dispatching mechanisms

Omicron has single dispatch, ie, the method lookup algorithm is based on a single object: the receiver of a message. In many object-oriented languages such as CLOS and Dylan, more objects than the receiver are used when a method is looked up. This is called multiple dispatch. In such cases the parameter objects may also influence which method is selected as the result of a message send action. It would be interesting to study a formal model incorporating multiple dispatch and define refinement relations and reliability requirements for such a model.

More flexible systems - are they reliable ?

Several different techniques which are used to make highly flexible software, breaks one or more of the reliability requirements of chapter 5. Example are the meta object protocol of CLOS (Kiczales et al. 1991) and Traces which makes it possible to add new code to an object after it has been created (Kiczales 1993). The perform-feature of Smalltalk and later versions of Java is another example mentioned in chapter 8. A study of how to combine such flexibility while ensuring reliable substitution might result in a better understanding of how to

create components which are very flexible but which also allow reliable substitution of itself and of components in its context.

10.2.3 Omicron's relationships to other models

Omicron is a formalisation of objects as found in object component systems. There exists a number of other formalisations of other types of systems, such as distributed processes and functional systems.

So far there have been formal frameworks for defining and reasoning about for instance algebraic properties, signature properties, data representation, data structures, state transitions and signals on channels. All of these formalisms can be used to describe some properties of (some) objects. However, none of them allow an intuitive description of objects' observable behaviour. Also, the formal frameworks have not been used to reason about such things as reliability properties of object descriptions and about reliable refinement relations.

Compared with Omicron, the other formalisations have been substantially more studied. These studies have revealed many results which might be interesting to compare with similar results which can be obtained from studies of Omicron. By a further study of Omicron and comparing the results with results from other formal models contribute to a better understanding of the differences, weaknesses and strengths of the various concepts.

Including functional expressions in Omicron

The different kinds of systems have their strengths in modelling different aspects of computer systems. For example, functional models are the easiest to use when describing data structures and data manipulation. Object component system models are better when describing how a system is partitioned into substitutable components and how these components collaborate. However, most large computer systems both contain data structures which are manipulated and are partitioned into components. Therefore, both models are useful when developing computer systems. While Omicron was developed, attempts were made at including functional expressions in the language. This was syntactically easy and some kinds of component behaviour were easier to express. However, the definitions and proofs became more complex. Therefore, these functional parts were left out.

A small example of how to include a kind of set notation is shown in chapter 2 when defining the model-view contract using Omicron. As shown in the example, including functional expressions in Omicron makes it simpler to express the model-view contract. To make Omicron easier to use when defining object behaviour, functional expressions should be added. However, this is more syntactic sugar than fundamentally new concepts since behaviour which is the result of using functional expressions can also be defined using the presented versions of Omicron.

Integration of other object properties

Objects, or more precisely the concepts modelled by objects, have many aspects which cannot naturally be expressed as sequences of observed actions. It may nevertheless be interesting to specify and prove properties of these aspects. The article (Nordhagen 1994) presents four different aspects of objects:

the Conceptual dimension	a description of the concepts the objects model
the Observable behaviour dimension	the aspect described by observable behaviour in Omicron
the Implementation dimension	the aspect described by Omicron expressions
the Representation dimension	a description of how the objects are represented, ie, the syntax of the *object* descriptions.

The article gives some direction as to the relationships between these aspects. Further study is needed to understand how the different aspects should be separated and linked to get practical solutions for describing complex systems. A result of the work reported in the article also indicates that it is possible to exploit the strengths of various formalisms in describing different aspects of objects and then describe how the formal descriptions are linked to allow synchronisation of the different views of the objects.

Some development methods allow developers to make models which describe different aspects of the objects making up the system. To varying degrees, the development methods allow a developer to define the relationships between the different models and help a developer ensure that the models are consistent. An example of a rather good tool is a tool for the OOram method. In this tool a developer works on one consistent model of the system, but is allowed to manipulate this model through editors focusing on different aspects of the model.

However, none of the development methods cover all aspects of objects and it is not clear how to integrate all aspects into a common framework. Neither do the methods focus and support developers in making reliable specifications.

The observable behaviour of components must be described in order to get reliable specifications. However, the observable behaviour is not well suited to describe other aspects of components. Much could therefore be gained in terms of efficient software development if appropriate mappings between the observable behaviour aspect and the other aspects could be found. This is an interesting topic for further study. Some attempts have been done at combining different aspects into a common model, see eg, (Nordhagen 1989) which gives an example. In the example a graphical language is used to describe a conceptual model of objects which are to be edited by users. This conceptual model is then used to configure user interface components which are specified and recognised by their observable behaviours. The user interface components use the conceptual model as a grammar and thereby becomes a syntax directed editor which supports the user in manipulating the system's data correctly. In this case a more traditional data model which is good for defining the semantics aspects of objects in a system is combined with a model of the observable behaviour of the objects. In the concrete system referred to in (Nordhagen 1989), the implementation dimension is presented as well. Information about the implementation aspect of objects is used in the algorithm creating objects. In such a case a user is presented with choices as found in the semantic dimension. When the user chooses an alternative, the editor looks for an implementation which both match the semantic aspects and the behaviour aspect necessary to insert the new object into an existing object structure and manipulate it in the editor. This strategy has lead to systems and components which are very flexible and easy to tailor and change to comply with different users' needs.

10.2.4 Applications of the theoretic results

Incorporating reliability properties into development methods

Only describing the necessary aspects of a component

It is important that specifications are presented in a form which is easy to understand and that the specifications are as simple as possible. To make the specifications as simple as possible it is important that only necessary aspects of the components are described. The rules for making reliable specifications lists the necessary aspects of component specifications. By taking the most readable software engineering methods and developing them further based on the rules for making reliable specifications, one might get one step further in making better specification languages for OCS components.

For a specification to be reliable it must specify the traces of observable actions between the components in the system. Observed actions are usually defined by making interaction diagrams as found in, eg, OOram and UML. In order to be able to make reliable specification by defining interaction diagrams, the diagrams must show actions involving visible objects, not just classes of objects something which is often the case. In addition, development methods must give attention to object creation and include such actions in their diagrams. This is not common in interaction diagrams found in most existing development methods. Also, the diagrams must allow a developer to specify parameters by referring to visible objects and not just refer to classes, types or roles of objects. This also lacks in most existing development methods.

Exactly how to change existing notations to allow reliable specifications will off course depend in the notation. Two sketches of what must be done for OOram and Catalysis are given below as preliminary suggestions, but this is mainly left for further study.

OOram

The OOram method has given attention to how large system descriptions can be split into smaller descriptions and then how such small descriptions can be merged to create large system descriptions. Their solution is a technique denoted synthesis. Details of the synthesis technique is documented in the OOram book and also discussed in (Andersen and Reenskaug 1992). What is denoted safe synthesis is a synthesis operation which closely corresponds to reliable substitution. A further study would reveal if the safe synthesis actually gives reliable substitution, and if not, prescribe necessary changes to the safe synthesis operation to achieve reliable substitution.

Catalysis

Catalysis is the method which, at present, has most similarities to the present work, and Catalysis addresses many of the same problems as this thesis. However, Catalysis is aimed at specifying the behaviour of an object of a given type and the most formal parts of Catalysis describe traditional functional properties of object behaviour. Omicron and the substitution proposition is aimed at specifications of the behaviour of components consisting of one or more objects and focus on more than just the functional aspects of object behaviour. So far, formal models of object behaviour as found in this thesis, have not existed. It can therefore be assumed that it has been difficult to incorporate such things into Catalysis. An interesting task would therefore be to incorporate the results of this thesis into the Catalysis development method. As the creators of Catalysis is now incorporating their ideas into UML, it would also be interesting to incorporate ideas from this thesis into UML as well.

Complexity, flexibility and visible objects

Many developers give arguments for splitting a component into many objects and argues that this gives more flexible and reusable code. At the same time, many of the same developers give patterns for avoiding problems related to the number of visible objects. This reflects the fact that there are trade-offs between specification complexities and refinement flexibility related to the number of visible objects of components. Such trade-offs are an interesting topic to pursue, but is left for further study. Presumably this is a less theoretically and more practically oriented topic since humans' abilities to cope with complex descriptions will influence decisions more than theoretic limitations.

Ensuring that a component is a reliable refinement

When there are reliable specifications of components, it is also possible to make tools to help verify that some code is a reliable implementation of a specification. However, making such tools is not trivial. This topic was presented in chapter 8. The conclusion was that type checking and type inference techniques could be developed so that many of the reliability requirements could be checked at compile time. To check that a component is a refinement, ie, has the same observable behaviour as its specification, can not be done by static checks at compile time. Such checks must be done by, eg, creating test drives from the system specifications or by other means. Another alternative is to make test environments for running the components and observing the actions resulting from executing a component in a certain context.

Component testing

Ensuring that a component is a reliable refinement is in many aspects comparable to testing a component, since both is concerned with the component's behaviour in a context. The definition of the reliable refinement relation and the reliability requirements might therefore be useful as a basis for developing test-strategies for components. It might be possible to simplify integration tests and/or comparisons of reference implementations. Reliability requirements will ensure that the test will also hold when/if the test environment is replaced by a reliable refinement.

In the paper "Design for Testability with Object-Oriented Systems" Communications of the ACM, 37(9):87-101, 1994 R.V. Binder writes: "To test a component, you must be able to control its input (and internal state) and observe its output." When reliability requirements are fulfilled we have control over the input and the internal state of the object. The observable output is also defined by the refinement relation. This analogy between design for testability and the definition of reliable specifications and refinements would be an interesting issue to pursue.

More user friendly specifications

Omicron is not particularly user friendly since the language was created so that it was easy to give a formal definition of the language's semantics. Specification and design notations and also programming languages have usually taken the opposite approach giving priority to the user friendliness of their language. To get more user friendly specifications, one should therefore combine the knowledge of how to make reliable specification with the knowledge of how to make user friendly notations. How to create such languages is left for further study.

Many user friendly informal approaches lack reasoning power to show necessary and sufficient reliability requirements and to make reliable specifications. However, the results from using Omicron can be applied to the less formally defined notations and languages in order to make them better - even though they are not perfect. Changing a user friendly design notation to make it more precise can be motivated by the possibility to introduce a formal basis like Omicron. Earlier, the extra precision necessary to make reliable specifications might only seem like unnecessary detail since common sense solved most problems. However, with a formal bases the extra precision can be used to verify that what is supposed to be a reliable refinement of a more over-all design actually is a reliable refinement. This might be interesting, at least in larger and/or critical systems.

From informal to formal specifications - sketches vs. details

A system specification may be difficult to understand if it includes too much detail. Also, in initial stages of the development of a system, there are few details, just sketches of the various parts of a system. Sketches are also used when presenting a design, before going into detail. Reliable specifications include many details and should therefore only be used when a reliable specification is actually needed. Different specification notations have different levels of detail, eg,

Least details:	Medium details:	Most details:
Pattern language	OOram Role model	Omicron reliable specification

The alternatives with least details are often easier to read and understand for humans in that they are usually more pedagogical. The introduction to a given (sub)system might therefore be done by starting with the least detailed description to get an overview and then moving on to the more detailed descriptions when details are needed. The

most detailed model may, eg, be of no particular interest other than as a basis for reasoning and verification of properties of (versions) of the description.

To ease the understanding of a specification, a given formal specification should also be available in more informal forms giving more user friendly, high level descriptions of the system's design. The challenge is to integrate the presented formalism into a framework of techniques with varying formality and detailing aimed at different kinds of audiences and serving different purposes.

Safer system building

Some object component designers have said that the possibility of getting reliable specifications and refinements of the components is the most interesting results of this thesis. This is because when reliable specifications and reliable refinements are available, it may be possible to get safer building of systems from prefabricated components. Safer building of systems is achieved since it is possible to give exact specifications of more complex - and hopefully more functional - components. In the long run, it might also be possible to verify that the implementations of the components are done according to the specifications - and that each implementation is a *reliable* refinement of the specification. Then it is ensured that it will work in all contexts which are reliable refinements of the context in the component's specification. Actually achieving safer system building is outside the scope of this thesis, and also definitely more than can be achieved by a single individual such as the author of this thesis. However, an important step is taken: we now know what a reliable specification is and we know what the reliability requirements are for the most common way of designing and implementing OCS components.

Specification and implementation of libraries of components

Existing libraries of components do not document their components by reliable specifications and there is no guarantee that a component implementation is a reliable refinements of its documentation. Usually, however, this is compensated for by having very simple components, eg, with single visible objects and only functional behaviour. In other cases the specifications and/or implementations have not been reliable and the users of the component experience unanticipated behaviour. On the other hand, there are examples of well documented and implemented library components which are used successfully. However, by using what is now known about reliable specifications and refinements, libraries of components might become easier and safer to use. Particularly when defining standard components, eg, the Java libraries, it is important that the components are reliably specified and that the implementations are reliable refinements of the specifications.

10.3 Summary of Conclusions

This thesis has presented a formal model of object components as viewed by designers of object component systems (OCS). Object component systems consist of objects which are grouped into components. OCS developers usually plan to create systems by combining components and therefore specify a system as a set of collaborating components.

The main focus of the thesis is *substitutability* of OCS components, both in the design and in the maintenance phases of a system's life cycle. System designs evolve when more general component descriptions are substituted by more detailed ones. Object component systems are maintained by substituting the different components of a system with new components which either are better versions of existing components or when the system must be adjusted to changes in its surroundings.

The formal model of the present thesis is given by the Omicron language and a calculus describing the semantics of the language. The calculus allows reasoning about components' behaviour, ie, the sending of messages, creation of new objects and updating of variables. The formal model includes object-oriented concepts such as object identity as separate from an object's state, message sending with dynamic binding, and also inheritance between objects.

There have been many attempts to give a formal model of objects. Few have modelled object-oriented concepts as used by developers of object component systems. Most attempts have built on traditional formal models such as communicating processes and algebraic methods. Such models are significantly different from models of object component systems. These differences makes it difficult to describe and reason about the behaviour of object component systems using a formalism based on the traditional models. The formal models which come closest to modelling object component systems, have either lacked reasoning power or do not focus on substitutability of OCS components.

The main distinction between the present OCS formalism and other existing object formalisms, is the way components and similarity of components are defined.

A *component* is not just a single object, but consists of one or more objects. A component can both send and receive messages from other components, create objects from templates in other components, and update variables in other components.

Similarity of components is defined relative to a context of other components. The objects in the context components observe the actions they are involved in. When two components are similar, they have similar observable behaviour in that the context objects observe similar actions. Two similar actions are either two object-creation actions which create new objects from the same context templates, or two update actions which update the same context variables with similar values, or two message-send actions which has the same context object as receiver and the same message selector. A typical example of observably similar behaviour is that the same context objects get similar messages in the same order.

In this thesis, the similarity concept of component designers is formalised through a *refinement relation*. This relation may be seen as a partial relation, in that two similar components are not defined similar for all contexts, but just for a given kind of context. This gives a trinary relation between a specification, a refinement of the specification, and a specification of a context.

Any component in an object component system may be substituted. For the system to continue to function as planned, the new component must have similar observable behaviour to the component it replaces relative to the context of other components. This means that a component can be replaced by a component which is a refinement of itself relative to the context of other components. However, as time passes, components in the context may be substituted with their refinements. It is therefore important that the new components also will have similar observable behaviour in the cases when context components are replaced by their refinements. When this holds, we have *reliable substitution* of components.

Figure 10.1 below illustrates reliable substitution in a system with two components. The original system is formed by the two components C_1 and C_2. The component R_1 is a refinement of C_1 relative to C_2 while R_2 is a refinement of C_2 relative to C_1. Typically, if R_1 and R_2 are developed by separate teams and tested relative to the rest of the system (C_2 and C_1 respectively), it is important that the refinements behave as planned when they are combined in $R_1||R_2$.

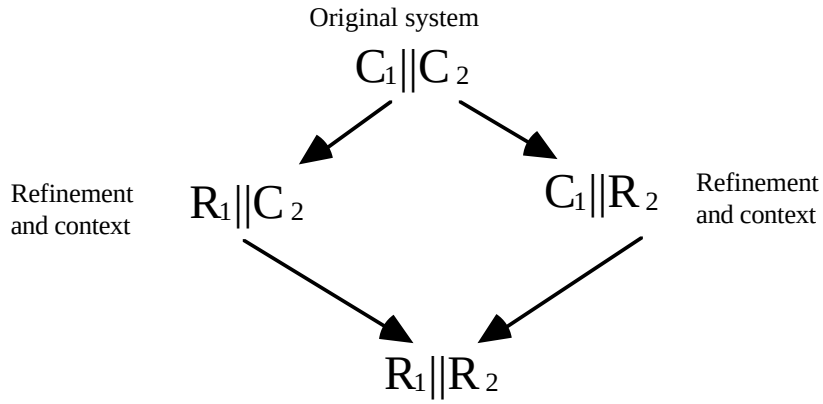


Figure 10.1 Components in a system may be replaced by refinements.

Defining a refinement relation based on component developers view of similarity was rather straight forward. There is rather common agreement on what similar behaviour is. However, showing that this definition gives reliable refinement is more difficult.

First, reliable substitution is formally expressed in the present *substitution proposition*. If a refinement relations assures reliable substitution, then it should be possible to prove the substitution proposition for the refinement relation. If the substitution proposition can be shown for a refinement relation, we say that the relation is a *reliable* refinement relation.

When trying to prove the substitution proposition for the refinement relation which was defined based on component designers' similarity concept, it soon became evident that the refinement relation was not reliable. Therefore, a new, stronger, reliable refinement relation was defined and the substitution proposition shown for this relation. However, finding a sufficient set of additional requirements on similar observable behaviour to assure reliable substitution was not a simple task. It was also important that the new reliable refinement relation was not too strong, since this would unnecessarily restrict implementors of refinements. The present thesis therefore presents a set of necessary and sufficient requirements which ensure reliable substitution of object components.

The definition of the refinement relation reflects how components are specified by object component designers. Therefore, the definition of the reliable refinement relation can be applied as rules for how to make specifications for components which can be reliably substituted. We call such specifications for *reliable specifications*. The most important lesson learned about reliable specifications is that traditional component specification is not reliable. Traditional specification typically defines the *types* of receivers and parameters of a component. This is not sufficient to ensure reliable specification of refinements. To be reliable, a specification must define the *visible objects* of each component and how these visible objects appear as receivers and parameters in messages sent between the components.

The requirements on reliable specifications may be seen as a limit on how much and how little must be said about reliably substitutable components in cases where these components consist of more than one visible object and where the implementor is to have maximum freedom in choosing the internal details of a component.

To ensure reliable substitution, it was also necessary to put requirements on components which are to be *reliable refinements* of reliable specifications, eg, requirements on component implementations. These requirements are expressed as formal *reliability requirements*. All the formal reliability requirements defined for Omicron configurations have practical consequences. These consequences can be formulated as advice on how to make implementations of components so that both the components and the component's collaborators can be reliably substituted. In many cases this advice corresponds with component developers' rules for making reusable components. It may therefore be assumed that the presented formalism captures some important aspects of object component systems. The most surprising new reliability requirement is that if-sentences comparing names of objects in other components give unreliable refinements.

The definition of reliable specifications and refinements can be used to create and/or change existing system description languages, and development methods. This will improve the development and use of OCS components and thereby better exploit the benefits of component based software. The definition of reliable substitution allows separate verification of the behaviour of parts of a system in such a way that the behaviour of the total system can be proved from the verified behaviour of the parts. Distribution of design work among

separate teams and the maintenance of extensible systems can exploit this property. Both can take advantage of the property that the behaviour of the total system can be proven from the separately verified behaviour of the parts. In this way the reliability properties of specifications and refinements can help co-ordinate work on different parts of a system and also simplify co-ordination and integration problems.

This thesis presents a formal model of the object-oriented properties described above. As documented, these are commonly found in object component design and implementation languages, and in development methods. However, variations of these properties exist and several are discussed throughout the thesis. As the discussions show, the requirements on reliable refinements and specifications are applicable to many variants of object component models. In addition, some requirements can translate to corresponding requirements to related models. For example if classes are present in the model, the requirements can be reformulated for classes. However, all variations in object component models have not been studied, such as models with objects which can move between components, systems with multiple dispatch of methods and the meta object protocol. Also, variations in the definition of similar actions exist. A common example is allowing a sequence of messages to be viewed as similar to a single message representing an abstraction of the sequence. A further study is necessary to identify other such variations which diverge from the presented alternatives. Then, further study is necessary to find out to what degree the reliability requirements are applicable when these diverging properties are present.

Reliable substitution of components becomes increasingly important as an increasing number of systems are designed, implemented and maintained by composing OCS components. It is a hope that the presented formalism and results will contribute to a deeper understanding of possibilities and pitfalls related to the development and maintenance of large extensible object component systems.

Bibliography

Abadi, Martín and Cardelli, Luca, (1994) *A theory of primitive objects*, Digital Equipment Corporation, DEC-SRC, Shorter versions in European Symposium om Programming, and in Theoretical Aspects of Computer Science 1994

Abadi, Martín and Lamport, Leslie, (1993) *Composing Specifications*, in ACM Transactions on Programming Languages and Systems January 1993, 15:1, pp. 73-132

Abadi, Martín and Lamport, Leslie, (1995) *Cojoining Specifications*, in ACM Transactions on Programming Languages and Systems May 1995, 17:3, pp. 507-534

Ågesen, Ole, Palsberg, Jens and Schwartzbach, Michael I., (1993) *Type Inference in SELF. Analysis of Objects with Dynamic and Multiple Inheritance*, European Conference on Object-Oriented Programming 1993, published in LNCS 707, pp. 247-267.

Agha, Gul, (1986) *A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, Mass.

Agha, Gul, Mason, Ian A., Smith, Scott and Talcott, Carolyn, (1993) *A Foundation for Actor Computation*, FTP: ftp://biobio.cs.uiuc.edu/pub/papers/actor-semantics.ps.Z

Allen, Robert and Garland, David, (1994) *Beyond Definition/Use: Architectural Interconnection*, Workshop on Interface Definition Languages 1994, published in ACM SIGPLAN Notices 29:8, pp. 35-45.

America, Pierre, (1987) *Inheritance and Subtyping in a Parallel Object-Oriented Language*, European Conference on Object-Oriented Systems 1987, published in LNCS 276, pp. 234-242.

America, Pierre and Rutten, Jan, (1990) *A Layered Semantics for a Parallel Object-Oriented Language*, Foundations of Object-Oriented Languages, REX School/Workshop 1990, Noordwijkerhout, The Netherlands, published in LNCS 489, pp. 91-123.

Andersen, Egil P., (1997) *Conceptual Modeling of Objects, A Role Modeling Approach*, Dr. Scient thesis 4, 1997, University of Oslo, Informatics Department

Andersen, Egil P. and Reenskaug, Trygve, (1992) *System Design by Composing Structures of Interacting Objects*, ECOOP 1992, published in LNCS 615, pp. 133-152.

Arapis, Costas, (1992) *Object Behaviour Composition: A temporal Logic Based Approach*, in Object Frameworks, ed: D. Tsichritzis, Centre Universitaire d'Informatique, University of Geneva, pp. 79-107

Barbier, Frank, (1992) *Object-oriented analysis of systems through their dynamical aspects*, in JOOP May 1992, 5:2, pp. 45-51

Binder, R.V., (1994) *Design for Testability with Object-Oriented Systems*, in Communications of the ACM 1994, 37:9, pp. 87-101

Birtwistle, Graham M., Dahl, Ole-Johan, Myhrhaug, Bjørn and Nygaard, Kristen, (1973) *Simula BEGIN*, Potrocelli/Charter, New York, ISBN : 0-88405-340-7

Blair, Gordon, Gallagher, John, Hutchison, David and Shephard, Dough, (1991) *Object-Oriented Languages, Systems and Applications*, Pitman Publishing, London, ISBN : 0-273-03132-5

Bollay, Denison, (1992a) *Code reuse: how to reduce maintenance costs by a factor of 10*, in JOOP July/August, 5:4, pp. 64-67

Bollay, Denison, (1992b) *Dylan (dynamic language) or CLOS +-*, in JOOP November-December 1992, 5:7, pp. 75-77

Bolognesi, Thommaso and Brinksma, Ed, (1987) *Introduction to the ISO Specification Language LOTOS*, in 1987, pp. 25-59

Briggs, Ted L. and Werth, John, (1994) *A Specification Language for Object-Oriented Analysis and Design*, ECOOP 1994, published in LNCS 821, pp. 365-385.

Broy, Manfred, (1996) *Toward a Mathematical Concept of a Component and Its Use*, the Components' Users Conference CUC'96 1996, Munich

Broy, Manfred, (1997) *A Functional Rephrasing of the Assumption/Commitment Specification Style*, in Formal Methods in System Design, ed: Kluwer

Bruce, Kim B., Cardelli, Luca and Pierce, Benjamin C., (1997) *Comparing Object Encodings*, in

Cardelli, Luca and Wegner, Peter, (1985) *On Understanding Types, Data Abstraction and Polymorphism*, in Computing Surveys December 1985, 17:4, pp. 471-522

Chambers, Craig, Ungar, David, Chang, Bay-Wei and Hölzle, Urs, (1991) *Parent and Shared Parts of Objects: Inheritance and Encapsulation in SELF*, in Lisp and Symbolic Computation 1991, 4:3

Cheon, Yoonsik, (1991) *Larch/Smalltalk: A Specification Language for Smalltalk*, Dept. C.S., Iowa State University, Ames, IA, 91-15

Cheon, Yoonsik and Leavens, Gary T., (1994) *A quick overview of Larch/C++*, in JOOP October 1994, 7:6, pp. 39-49

Cook, Steve and Daniels, John, (1994) *Designing Object Systems*, Prentice Hall, Hemel Hempstead, UK, ISBN : 0-13-203860-9

Cook, William R., Hill, Walter L. and Canning, Peter S., (1990) *Inheritance Is Not Subtyping*, in ACM 089791-343-4, 90:1, pp. 125-135

Cox, Brad J. and Novobilski, Andrew J., (1991) *Object-Oriented Programming: An Evolutionary Approach*, Addison-Wesley, 270, ISBN : 0-201-54834-8

D'Souza, Desmond and Wills, Alan, (1995) *CATALYSIS - Practical Rigor and Refinement*, ICON Computing, Inc., Available at http://www.iconcomp.com

D'Souza, Desmond and Wills, Alan Cameron, (1997) *Types, Behaviours, Collaborations, Refinement, and Frameworks - Input for OMG OOA&D Submission*, ICON Computing, www.icon.com,

Dahl, Ole-Johan and Owe, Olaf, (1991) *Formal Development with ABEL*, in VDM'91, LNCS, Vol. 552, ed: S. Prehn and W. J. Toetenel, Springer-Verlag, pp. 320-362

Dahl, Ole-Johan and Owe, Olaf, (1998) *Formal Methods and the RM-ODP*, Research Report 261, Informatics Department, University of Oslo, May 1998

Dahl, Ole-Johan, (1992) *Verifyable Programming*, Prentice Hall, ISBN : 0-13-951062-1

Dahl, O.J., Dijkstra, E. W. and Hoare, C.A.R., (1972) *Structured Programming*, Academic Press, ISBN : 0-12-200550-3

Dahl, Ole-Johann, Myhrhaug, Bjørn and Nygaard, Kristen, (1968) *SIMULA 67 Common Base Language*, Norwegian Computing Center,

Dami, Laurent, (1993) *The HOP Calculus*, in Visual Objects, ed: D. Tsichritzis, Centre Universitaire d'Informatique, Université de Genève, 24 rue Général-Dufour, CH-1211 Genève 4, Switzerland, Genève, pp. 149-210

Danforth, Scott and Tomlinson, Chris, (1988) *Type Theories and Object-Oriented Programming*, in ACM Computing Surveys March 1988, 20:1, pp. 29-72

Deutsch, L. Peter, (1989) *Design Reuse and Frameworks in the Smalltalk-80 System*, in Software Reusability, Vol. II, ed: T. J. Biggerstaff and A. J. Perlis, ACM-Press, New York, New York, pp. 55-71, ISBN : 0-201-50018-3

Ehrich, H.-D., Goguen, J.A. and Sernadas, A., (1990) *A Categorical Theory of Objects as Observed Processes*, Foundations of Object-Oriented Languages, REX School/Workshop 1990, Noordwijkerhout, The Netherlands, published in LNCS 489, pp. 203-228.

Eliëns, Anton, (1994) *Principles of Object-Oriented Software Development,* Addison-Wesley, 513, ISBN : 0-201-62444-3

Fiadeiro, J. and Maibaum, T., (1990) *Describing, Structuring and Implementing Objects*, Foundations of Object-Oriented Languages, REX School/Workshop 1990, Noordwijkerhout, The Netherlands, published in LNCS 489, pp. 274-310.

Gamma, Erich, Helm, Richard, Johnson, Ralph and Vlissides, John, (1994) *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison-Wesley, 416, ISBN : 0-201-63361-2

Gangopadhyay, Dipayan and Mitra, Subrata, (1993) *ObjChart: Tangible Specification of Reactive Object Behaviour*, European Conference on Object-Oriented Programming 1993, published in LNCS 707, pp. 432-457.

Goldberg, Adele, (1990) *Information Models, Views and Controllers - The key to reusability in Smalltalk-80 lies within MVC,* in Dr. Dobb's Journal July 1990, pp. 54-61

Goldberg, Adele and Robson, Dave, (1983) *Smalltalk-80: The Language and Its Implementation,* Addison-Wesley, Reading, Mass.

Graver, Justin O., (1990) *A Type System for Smalltalk,* in ACM 089791-343-4, 90:1, pp. 136-150

Gunter, Carl A. and Mitchell, John C., (1994) *Theoretical Aspect of Object-Oriented Programming,* The MIT Press, ISBN : ISBN 0-262-07155-X

Guttag, J., (1980) *Notes on type abstraction,* in IEEE Transactions on Software Engineering Jan. 1980, SE-6:1, pp. 13-23

Guttag, J.V., (1993) *Larch: Languages and Tools for Formal Specification,* Springer Verlag, New York

Helm, Richard, Holland, Ian M. and Fangopadyay, Dipayan, (1990) *Contracts: Specifying Behavioural Composition in Object-Oriented Systems*, OOPSLA/ECOOP 1990, Ottawa, Canada, published in ACM SIGPLAN Notices 25:10, pp. 169-180.

Henderson-Sellers, B., (1993) *The economy of reusing library classes,* in JOOP July-August 1993, 6:4, pp. 43-50

Hewitt, C., (1977) *Viewing control structures as patterns of passing messages,* in Journal of Artificial Intelligence June 1977, 8:3, pp. 323-364

Hoare, C.A.R., (1978) *Communicating Sequential Processes,* in Communications of the ACM August/1978, 21:8, pp. 666-677

Hoare, C.A.R., (1985) *Communicating Sequential Processes,* Prentice-Hall

Hogg, John, (1991) *Islands: Aliasing Protection In Object-Oriented Languages*, OOPSLA 1991, published in SIGPLAN Notices 26:11, pp. 359-411.

Hogg, John, Lea, Doug, Wills, Alan, deChampeaux, Dennis and Holt, Richard, (1992) *The Geneva Convention On The Treatment of Object Aliasing,* in OOPS Messenger from ACM April 1992, 3:2, pp. 11-16

Hohan, James Michael, (1988) *A New Design for the Smalltalk-80 Text Subsystem*, Master of Science, University of Illinois at Urbana-Champaign

Holbækk-Hansen, Erik, Håndlykken, Petter and Nygaard, Kristen, (1975) *System Description and the Delta Language*, Norwegian Computing Centre, 523

Holland, Ian M., (1992) *Specifying reusable components using Contracts*, European Conference on Object-Oriented Programming 1992, published in LNCS:615, pp. 287-307.

Honda, Kohei and Tokoro, Mario, (1991) *An Object Calculus for Asynchronous Communication*, ECOOP 1991, published in LNCS 512, pp. 133-147.

Hölzle, Urs, (1993) *Integrating Independently-Developed Components in Object-Oriented Languages*, European Conference on Object-Oriented Programming 1993, published in LNCS 707, pp. 36-56.

Jacobson, Ivar, Bylund, Stefan, Jonsson, Patric and Ehneboom, Staffan, (1995) *Using contracts and use cases to build pluggable architectures,* in JOOP May 1995, 8:2, pp. 18-76

Jacobson, Ivar, Jonsson, Patric and Övergaard, Gunnar, (1992) *Object-Oriented Software Engineering*, Addison-Wisley, ISBN : 0-201-54435-0

Johnson, Ralph E., (1992) *Documenting Frameworks with Patterns*, Object-Oriented Programming, Languages and Applications 1992, Vancouver BC, published in SIGPLAN Notices, pp. 63-76.

Johnson, Ralph E. and Foot, Brian, (1988) *Designing Reusable Classes,* in JOOP 1988, 1:2, pp. 22-25

Jones, Cliff B., (1983) *Specification and design of (parallel) programs*, Information Processing 83: IFIP 9th World Congress 1983, pp. 321-332.

Jones, C. B., (1995) *Accommodating Interference in the Formal Design of Concurrent Object-Based Programs,* in FMiSD, Kluwer Academic Publishers, Boston. Manufactured in the Netherlands, 1:19

Keene, Sonya E., (1989) *Object-oriented Programming in Common LISP,* Addison-Wesley, Reading, Mass., ISBN : 0-201-17589-4

Khoshafian, Setang N. and Copeland, George P., (1986) *Object Identity*, Object-Oriented Programming, Systems, Languages and Applications 1986, published in ACM SIGPLAN Notices, pp. 406-416.

Kiczales, Gregor, (1993) *Traces (A Cut at the "Make Isn't Generic" Problem),* ISOTAS 1993

Kiczales, Gregor, Rivieres, Jim des and Bobrow, Daniel G., (1991) *The Art of the Metaobject Protocol,* MIT Press

Krasner, Glenn E. and Pope, Stephen T., (1988) *A Cookbook for using the Model-View-Controller User Interface Paradigm in Smalltalk-80,* in JOOP Aug./Sep. 1988, pp. 26-49

Lamport, Leslie, (1994) *The Temporal Logic of Actions,* in ACM Transactions on Programming Languages and Systems May 1994, 16:3, pp. 872-923

Lamport, Leslie and Lynch, Nancy, (1990) *Distributed Computing: Models and Methods,* in Handbook of Theoretical Computer Science, ed: J. v. Leeuwen, Elsevier Science Publishers, pp. 1155-1199

Lano, Kevin and Haughton, Howard, (1994) *Object-oriented specification case studies,* Prentice Hall, 236, ISBN : 0-13-097015-8

Leavens, Gary T. and Weihl, William E., (1990) *Reasoning about Object-Oriented Programs that use Subtypes*, ECOOP/OOPSLA 1990, Ottawa, Canada, published in ACM SIGPLAN Notices 25:10, pp. 212-223.

Lieberherr, K., Holland, I. and Riel, A., (1988) *Object-Oriented Programming: An Objective Sense of Style*, OPPSLA 1988, San Diego, California, published in SIGPLAN Notices (ACM) 23:11, pp. 323-334.

Liskov, B. and Guttag, J., (1986) *Abstraction and Specification in Program Development,* MIT-Press

Liskov, Barbara and Wing, Jeannette M., (1993) *A New Definition of the Subtype Relation*, European Conference on Object-Oriented Programming 1993, published in LNCS 707, pp. 119-141.

Meseguer, José, (1993) *A Logical Theory of Concurrent Objects and Its Realisation in the Maude Language*, in Research Directions in Concurrent Object-Oriented Programming, ed: G. Aga, P. Wegner and A. Yonezawa, The MIT Press, Cambridge, Massachusetts, pp. 314-390, ISBN : 0-262-01139-5

Meyer, Bertrand, (1988) *Object-Oriented Software Construction,* Prentice-Hall

Meyer, Bertrand, (1989) *The New Culture of Software Development: Reflection on the Practice of Object-Oriented Design*, TOOLS 1989, Paris, France, pp. 13-23.

Meyer, Bertrand, (1994) *Reusable Software,* Prentice Hall, ISBN : 0-13-245499-8

Milner, Robin, (1989) *Communication and Concurrency*, Prentice Hall, 260, ISBN : 0-13-115007-3

Milner, Robin, Parrow, Joachim and Walker, David, (1989a) *A Calculus of Mobile Processes. Part I*, University of Edinburgh

Milner, Robin, Parrow, Joachim and Walker, David, (1989b) *A Calculus of Mobile Processes. Part II*, University of Edinburgh

Milner, R., Tofte, M. and Harper, R., (1990) *The definition of Standard ML*, The MIT Press, ISBN : 0-262-12355-9

Moreira, Ana M.D. and Clark, Robert G., (1994) *Combining Object-Oriented Analysis and Formal Description Techniques*, ECOOP 1994, published in LNCS 821, pp. 344-364.

Morzenti, Angelo and Pientro, Pierluigi San, (1991) *An Object-Oriented Logic Language for Modular System Specification*, ECOOP 1991, published in LNCS 512, pp. 131-147.

Nierstrasz, Oscar, (1993) *Composing Active Objects*, in Research Directions in Concurrent Object-Oriented Programming, ed: G. Aga, P. Wegner and A. Yonezawa, The MIT Press, Cambridge, Massachusetts, pp. 151-171, ISBN : 0-262-01139-5

Nierstrasz, Oscar and Papathomas, Michael, (1990a) *Toward a Type Theory for Active Objects - Working Paper*, in Object Management, Vol. July 1990, ed: D.C.Tsichritzis, University of Geneva, pp. 295-304

Nierstrasz, Oscar and Papathomas, Michael, (1990b) *Viewing Objects as Patterns of Communicating Agents*, ECOOP/OOPSLA 1990b, published in SIGPLAN NOTICES 25:10

Nordhagen, Else, (1987) *A New Text Editor Implementation for Smalltalk-80*, Center for Industrial Research (SI, now SINTEF), Oslo, Norway, EKI-N-42

Nordhagen, Else, (1989) *Generic Object-Oriented Systems*, TOOLS 1989, Paris, France, published in TOOLS'89 Proceedings, pp. 131-140.

Nordhagen, Else K., (1992) *π-Calculus semantics for a Smalltalk like language*, NIK 1992, Tromsø, Norway, pp. 43-54.

Nordhagen, Else K., (1994) *Objects in Four Dimensions - the COIR-archticture*, in To be published, copy available from the author (lc@ifi.uio.no)

Owe, Olaf, (1988) *An Alias-Free Object-Oriented Language Concept*, Department of Informatics, University of Oslo, 141, ISBN 82-7368-047-9

Palsberg, Jens and Schwartzback, Michael I., (1994) *Object-Oriented Type Systems*, John Wiley & Sons Ltd., 180, ISBN : ISBN 0-471-94128-X

Papathomas, Michael, (1991) *A Unifying Framework for Process Calculus Semantics of Concurrent Object-Oriented Languages*, in LNCS 612, Vol. 612, ed: M. Tokoro, O. Nierstrasz and P. Wegner, Springer-Verlag, pp. 53-80, ISBN : 0-540-55613-3

Papathomas, Michael, (1992) *Behaviour Compatibility and Specification for Active Objects, Working Paper*, in Object Frameworks, ed: D. Tsichritzis, University of Geneve, pp. 31-40

Pierce, Benjamin C. and Sangiorgi, Davide, (1996) *Behavioural Equivalence in the Polymorphic Pi-Calculus*, Indiana University Computer Science, TR 468

Plotkin, G., (1981) *A structural approach to operational semantics*, Computer Science Department, Aarhus University, DAIMI FN-19

Pountain, Dick and Szyperski, Clemens, (1994) *Extensible Software Systems*, in BYTE May 1994, pp. 57-62

Reenskaug, Trygve, Andersen, Egil, Berre, Arne Jørgen, Hurlen, Anne, Landmark, Anton, Lehne, Odd Arild, Nordhagen, Else, Næss-Ulseth, Eirik, Oftedal, Gro, Skaar, Anne Lise and Stenslet, Pål, (1992) *OORASS:*

Seamless Support for the Creation and Maintenance of Object-Oriented Systems, in JOOP October 1992, 5:6, pp. 27-41

Reenskaug, Trygve and Nordhagen, Else K., (1989) *The design and description of complex, object-oriented systems*, SI - Senter for Industriforskning, 89-272-1, ISBN: 82-411-0193-7

Reenskaug, Trygve, Wold, Per and Lehne, Odd Arild, (1995) *Working with Objects, The OOram Software Engineering Method,* Prentice Hall, ISBN : 1-884777-10-4

Reghizzi, S. Crespi and Paratesi, G. Galli de, (1991) *Definition of reusable concurrent software components*, European Conference on Object-Oriented Programming 1991, published in LNCS 512, pp. 148-166.

Sato, Ichiro and Tokoro, Mario, (1992) *A Formalism for Real-Time Concurrent Object-Oriented Computing*, OOSPLA 1992, published in ACM SIGPLA Notices 27:10, pp. 315-326.

Schaffert, Craigh, Cooper, Topher and Wilpolt, Carrie, (1985) *Trellis Object-Based Environment*, Digital Equipment Corporation, DEC-TR-372

Skuce, Douglas and Mili, Ali, (1995) *Behaviour specification in object-oriented programming,* in JOOP January 1995, 7:8, pp. 41-49

Snyder, Alan, (1986) *Encapsulation and Inheritance in Object-Oriented Programming Languages*, Object-Oriented Programming, Languages, Systems and Applications 1986, published in ACM SIGPLAN Notices, pp. 38-45.

Stroustrup, Bjarne, (1986) *The C++ Programming Language,* Addison Wesley

Stølen, Ketil, (1996) *Assumption/Commitment Rules for Dataflow Networks - with an Emphasis on Completeness*, Programming Languages and Systems - ESOP '96 1996, published in LNCS 1058, pp. 356-372.

Thomsen, Bent, (1993) *Plain CHOCS, A second generation calculus for higher order processes,* in Acta Informatica, 30:1, pp. 1-59

Udell, John, (1994) *ComponentWare,* in Byte May 1994, pp. 46-56

Ungar, David, Chambers, Craig, Chang, Bay-Wei and Hölzle, Urs, (1991) *Organising Programs Without Classes,* in Lisp and Symbolic Computation 1991, 4:3

Vasconcelos, Vasco T., (1994) *Typed Concurrent Objects*, ECOOP 1994, published in LNCS 821, pp. 100-117.

Waldén, Kim and Nerson, Jean-Marc, (1995) *Seamless Object-Oriented Software Architecture,* Prentice Hall, ISBN : 0-13-031303-3

Walker, David, (1991) *π-Calculus Semantics of Object-Oriented Programming Languages*, TACS 1991, published in LNCS 526, pp. 532-547.

Walker, David, (1992) *Objects in the π-calculus,* in Personal copy, pp. 1-34

Wegner, Peter, (1987) *Dimensions of Object-Based Language Design*, OOPSLA 1987, published in ACM SIGPLAN Notices, pp. 168-182.

Wegner, Peter, (1993) *Tradeoff between Reasoning and Modeling,* in Research Directions in Concurrent Object-Oriented Programming, ed: G. Aga, P. Wegner and A. Yonezawa, The MIT Press, Cambridge, Massachusetts, pp. 22-41, ISBN : 0-262-01139-5

Wegner, Peter, (1994) *Models and Paradigms of Interaction,* in Object-Based Distributed Programming, ECOOP'93 Workshop, Vol. 791, ed: R. Guerraoui, O. Nierstrasz and M. Riveill, Springer-Verlag, pp. 1-32, ISBN : 3-540-57932-X

Wegner, Peter, (1995) *Object Technology, A Virtual Roundtable, Theoretical Foundations,* in IEEE Computer October 1995, 28:10, pp. 70-72

Wegner, Peter and Zdonick, Stanley B., (1988) *Inheritance as an Incremental Modification Mechanism or What Like Is or Isn't Like*, European Conference on Object-Oriented Programming 1988, Oslo, Norway, published in LNCS 322, pp. 55-77.

Wieringa, Roel, Jonge, Wieben de and Spruit, Paul, (1994) *Roles and Dynamic Subclasses: A Modal Logic Approach*, ECOOP 1994, published in LNCS 821, pp. 32-59.

Wirfs-Brock, Rebecca, Wilkerson, Brian and Wiener, Lauren, (1990) *Designing Object-Oriented Software*, Prentice Hall, ISBN : 0-13-629825-7

Wirfs-Brock, Rebecca J. and Johnson, Ralph E., (1990) *Surveying Current Research in Object-Oriented Design*, in Communications of the ACM September 1990, 33:9, pp. 105-124

Appendixes

A: Basic Definitions

B: Translation between the π-calculus and Omicron

C: References

Appendix A : Basic Definitions

1 BNF

The BNF language used in this document consists of the following symbols:

X ::= Y	X is defined as Y
X | Y	X or Y
X*	zero or more X
X^+	one or more X
$X^{*>2}$	two or more X
{X}	zero or one X
$X^*_;$	zero or more X separated by ';' . ';' may be replaced by any other character.

Bold font is used to denote reserved words and character combinations.

2 Map notation and formal definitions

The notation for Maps used in this document is taken from (Dahl 1992) pages 217-219, Initialized maps. The Map definition has been expanded with the functions ^.Dom and ^+^.

A Map is a representation of a function U -> T defined for a finite set of arguments. The domain and codomain are arbitrary given types. In this document the domains are either object names or slot names. The codomains are correspondingly object definitions or object names.

The map concept is defined as follows:

```
type Map{U,T} ==
module
func init() :                    ⟶ Map          Empty Map
func ^[^→^] :    Map × U × T     ⟶ Map          Update Map
func ^(^) :      Map × U         ⟶ T            Apply map
        M(x) ==
                 case M of
                 init            :    ⊥
                 M'[y→z]         :    if x=y then z else M'(x)

func ^.Dom :     M               ⟶ Set{U}       The domain of M
        M.Dom ==
                 case M of
                 init            :    {}
                 M'[y→z]         :    M'.Dom ∪ {y}

func ^-^ :       Map × U         ⟶ Map
        M-x ==
                 case M of
                 init            :    init
                 M'[y→z]         :    if x=y then M'x else (M'-x)[y→z]

func ^+^ :       Map × Map       ⟶ Map
        M+N ==
                 case N of
                 init            :    M
                 N'[y→z]         :    (M+N')[y→z]
```

For configurations of parallel objects the notation C||D is used instead of C+D.

Slots maps with the special object name **this**:

C(o)(s) == let value = C(owner(C,o,s))(s) in
if value = **this** then return(o)
else return(value)

Special functions eliminating the need to use ☆ and : in slotNames in sentences, ie, when looking up or updating a slot:

C(o)(s) == if @C(o)(s) then return(C(o)(s))
else
if @C(o)(:s) then return(C(o)(:s))
else
if @C(o)(:s☆) then return(C(o)(:s☆))
else
if @C(o)(s☆) then return(C(o)(s☆))

Correspondingly for C(o)[s→j]

3 Object name substitutions

This section describes substitutions for object names in relation to Omicron configurations and Omicron actions. The below is obvious to those who are familiar with substitutions and is therefore meant as a support to those who are not. It also gives a precise definition of the interpretation of substitutions used in this document.

To not confuse the domains and ranges of substitutions and configurations, the terms keys and values are used for substitution domains and ranges respectively. Also the syntax for defining mappings for substitutions is different from that of Slots and Configurations to make the distinction clear.

Definition: An object name substitution

A substitution σ is an operation mapping object names to object names.
The operation is written post fix, eg, Cσ, and binds stronger than other operations
so that, eg, A||Cσ means A || (Cσ)
The set of all object name substitutions is denoted $\mathcal{Su}$.

A substitution has the form $\{a_1/b_1\}\ldots\{a_n/b_n\}$ where {a/b} means 'b' is replaced by 'a'. For b-names not listed in the {a/b}-sequence the substitution mapping is equal to the identity function. For the substitution to be well-defined, each b_i in the sequence must be unique. {a/b}* denotes a sequence of {a/b}.

Definition: Keys and values of a substitution

The keys of a substitution is the set of names which map to other names than themselves:
keys(σ) = {b | bσ ≠ b }
The values of a substitution is the set of names which some other name than themselves map to:
values(σ) = {bσ | bσ ≠ b }

Definition: Cσ

Oσ denotes an object obtained from O by simultaneously substituting oσ for each occurrence of the object name o in O.

$C = o_1 : O_1 \,\|\ldots\|\, o_n : O_n \quad \Rightarrow \quad C\sigma = o_1\sigma : O_1\sigma \,\|\ldots\|\, o_n\sigma : O_n\sigma$
$O = M, S \quad \Rightarrow \quad O\sigma = M\sigma, S$
$M = [\, s_1 \rightarrow o_1, \ldots s_n \rightarrow o_n] \quad \Rightarrow \quad M\sigma = [\, s_1 \rightarrow o_1\sigma, \ldots s_n \rightarrow o_n\sigma]$

Definition: ασ

ασ denotes an action obtained from α by simultaneously substituting oσ for each occurrence of an object name o in α.

$(e\text{->}(o_1.s_1)\ldots(o_n.s_n) := i)\sigma = e\sigma\text{->}(o_1\sigma.s_1)\ldots(o_n\sigma.s_n) := i\sigma$
$(e\text{->}(o.s) := k/j)\sigma = e\sigma\text{->}(o\sigma.s) := k\sigma / j\sigma$
$(e\text{->}o!m(p_1\ldots p_n)/k)\sigma = e\sigma\text{->}o\sigma!m(p_1\sigma\ldots p_n\sigma)/k\sigma$
$(e\text{->error})\sigma = e\sigma\text{->error}$

Definition: Combining substitutions

Given two substitutions $\sigma = \{s_i/t_i\}^*$ and $\rho = \{q_i/r_i\}^*$.

The result of applying the combined substitution $\sigma\rho = \delta$, is the same as first applying σ and then ρ.

δ will then look like:

if some j so that $t_i=q_j$ for some i then

A) if $r_j = s_i$ then neither s_i/t_i or q_j/r_j is found in δ (as they give identity)

B) if not then s_i/r_j is found in δ (the two substitutions are combined)

if there is no j so that $t_i=q_j$ then

C) if for some k $s_i=q_k$ then s_i/t_i is found in δ and q_k/r_k is not found (as q_k is not found after σ is applied)

D) if there is no such k then s_i/t_i and q_k/r_k is both found in δ

4 New form for Case statements

The following notation allows casing on more than one item, eg:

```
case x, y of
        x = 0, y = 0 : expression1
        x > 0, y> 0 : expression2
        otherwise: expression3
```

which is an alternative from for:

```
case x of
        x = 0 :  case y of :
                        y = 0 : expression 1
                        otherwise : expression3
        x > 0 : case y of :
                        y > 0 : expression 2
                        otherwise: expression3
```

Using the alternative with casing on more than one item, usually gives a more readable definition as is the case in this small example.

Appendix B:
Translations between
the π-calculus and Omicron

1 The π-calculus and Omicron

Process calculi does not reflect object-oriented concepts. However it is rather simple to describe the semantics of an object-oriented language using the π-calculus (Milner et al. 1989a), (Milner et al. 1989b), (Thomsen 1993) as, eg, done in (Walker 1991), (Walker 1992) and (Nordhagen 1992). Translating parallel Omicron to π is also rather simple and is done below. It is also rather simple to translate from π to Omicron something which is done further below.

First the π-calculus language is presented and then the translations.

π-calculus syntax:
Below is given a short summary of the π-calculus expressions used in the semantic definitions. For a more detailed description see (Milner et al. 1989a) and (Milner et al. 1989b).

The π-calculus is a process calculus in which processes with changing communication structure may be expressed. The processes share communication channels and such channels can be passed on to other processes. Each channel has a name and the only "values" which may be communicated are such names. This means that in the π-calculus "variables", "communication channels" and "values" are all *names*. An infinite set N of names is presupposed, and in the below description single letter such as x,y,v (possibly with subscripts) range over names. Below the informal semantics of π will be expressed using words such as channel, parameter, etc. to reflect the role the name has in the expression.

The basic building blocks of π-expressions are process expressions. P,Q range over such expressions that are built from the following expressions:

P ::=

$\underline{x}y.P$	output action: send name y on channel x and behave like P[20]
x(y).P	input action: receive an unknown name (say v) on channel x, and then behave like P{v/y) (P with v for y, v must be a new name not occurring in P)
(νy)P	y is a private name for P, making it unique within the total system. y may be passed to other processes so that they can communicate on this channel.
[x=y]P	behave like P if name x is equal to name y else terminate
P | Q	behave as if P and Q act independently in parallel. P and Q may share channels and communicate on these.
P + Q	behave either like P of like Q
0	terminate (usually left out at end of expressions)

The expression α * P is defined as:
α * P == α.(P | α * P) where α is one of $\underline{x}y$ or x(y).

From Omicron to π-calculus

In the below $[[\text{Omicron expression}]]_\pi$ denote a function translating the Omicron expression into the π language.

A Configuration of Omicron objects is translated into π-processes as follows:

$[[\text{Object}_1 \parallel \ldots \parallel \text{Object}_n]]_\pi == [[\text{ Object}_1]]_\pi \ \ldots \ [[\text{ Object}_n]]_\pi$

A simple translation of Omicron objects into π processes can be done as shown below. This translation do not take into account inheritance between objects. Inheritance is shown further below.

$\underline{x}(y)$ is short for (νy) $\underline{x}y$, that is send a new local name out on x.

[20] In π-calculus "overscore" is used, instead of "underscore".

$[[o : (Slots, Body)]]_\pi$ ==

$\underline{co}(o) * [[Slots]]_\pi$ |

$o(u) * u(m).($

$([m=m_i]\ (z)\ [[m_i]]_\pi(z) \mid z(i).\underline{u}i +$

$[m=\#clone]\ u(v).co(k).\underline{v}k +$

$[m=\#exe]\ u(v).co(k).\underline{k}(l).\underline{l}\#exe2.\underline{v}l +$

$[m=\#exe2]\ u(p_1).\underline{w}_{p1}p_1 ... u(p_m).\underline{w}_{pm}p_m.(\underline{o}'o \mid [[Body]]_\pi\ (o')))$

where $p_1 \ldots p_n$ = Slots.inputs

The channel 'co' is used to create a new object clone by sending a name on this channel. A new object clone is created as the operator * is defined as follows:

$\alpha * P == \alpha.(P \mid \alpha * P)$

and $\alpha = \underline{co}(o)$ making o is a new local channel for each instantiation. To clone an object, one must send the message #clone to it. The #clone-code above first receive a return-cannel and then send on the co-cannel before it returns the name of the new object.

When an object receives a message it first sends out a local name u in o(u) on the channel o and then spawns a new process (*). This new process models the new instance of the method to be contacted when a new call is done. Then the process receives the message on the new channel (u(m)).

An object is turned into an executing process by sending it the message #exe. It then clones itself and sends out the channel name to the new clone. Then it sends the message #exe2 to the new clone which then receives a set of parameters and starts executing the body of the object.

To get the body to refer to the correct in-channel for the object of which it is the body to, the name o is sent on the local channel o' (in $\underline{o}'o$). The name o must be sent to the process modelling the body since the object may be cloned and then o will and should be different from the original object's in-channel.

Slots definition

$[[\ [\ s_1 \rightarrow i_1, \ldots, s_n \rightarrow i_n\ ]\ ]]_\pi == (s_1, \ldots, s_n)\ Reg_{s_1}(i_1) \mid ..... \mid Reg_{s_n}(i_n)$

where

$Reg_S(i) == \underline{r}_S i.Reg_S(i) + w_S(j).\ Reg_S(j)$

Slot read:

$[[\ s\ ]]_\pi\ (v) == r_s(z).\ \underline{v}z$

Including inheritance:

To model slot lookup (and update) in the inheritance hierarchy three functions/macros are defined to control the sequence of lookups in the π-program:

P ifContinue Q == (stop continue)(P | stop(x) + continue(x).Q) where x is not in fn(Q)

Stop == $\underline{stop}$ stop.0

Cont == $\underline{continue}$ continue.0

The slot lookup algorithm defined here is similar to the algorithm defined earlier in the section on inheritance slots for Omicron: left to right slot priority and closest object first.

Object without inheritance slots:

$[[o : (Slots, Body)]]_\pi$ ==

$\underline{co}(o) * [[Slots]]_\pi$ |

$o(u) * u(m).($

$([m=\#clone]\ u(v).co(k).\underline{v}k +$

$[m=\#exe]\ u(v).co(k).\underline{k}(l).\underline{l}\#exe2.\underline{v}l +$

$[m=\#exe2]\ u(p_1).\underline{w}_{p1}p_1 ... u(p_m).\underline{w}_{pm}p_m.(\underline{o}'o \mid [[Body]]_\pi\ (o')) +$

$[m=\#lookup]\ u(m).u(v).$

$([m=mi]\ [[mi]]_\pi(z) \mid z(i).\underline{v}i.Stop) +$

$[m \neq mi]\ Cont\ ) +$

$[m=\#store]\ u(m).u(k)$

$([m=mi]\ \underline{w}_{mi}k.Stop) +$

$[m \neq mi]\ Cont\ ) +$

$[m=OTHER]\ \ o(v).\underline{v}\#lookup.\underline{v}m.\underline{v}(u)\ ifContinue\ ))$

Object with inheritance slots $s_1 \ldots s_n$:

$[[o : (\text{Slots, Body})]]_\pi ==$

$\underline{co}(o) * [[\text{Slots}]]_\pi \mid$

$o(u) * u(m).($

$([m=\#clone]\ u(v).co(k).\underline{v}k +$

$[m=\#exe]\ u(v).co(k).\underline{k}(l).\underline{l}\#exe2.\underline{v}l +$

$[m=\#exe2]\ u(p_1).\underline{w}_{p1}p_1 \ldots u(p_m).\underline{w}_{pm}p_m.(o')\ (\underline{o}'o \mid [[\text{Body}]]_\pi\ (o)) +$

$[m=\#lookup]\ u(m).u(v).$

$([m=mi]\ [[mi]]_\pi(z) \mid z(i).\underline{v}i.\text{Stop} +$

$[m \neq mi]\ (z_1)\ (\ [[s_1]]_\pi(z_1) \mid z_1(i_1).\underline{i}_1(w).\underline{w}\#lookup.\underline{w}m.\underline{w}(v)\ \text{ifContinue}$

$\ldots$

$(z_n)\ (\ [[s_n]]_\pi(z_n) \mid z_n(i_n).\underline{i}_n(w).\underline{w}\#lookup.\underline{w}m.\underline{w}(v)\ \text{ifContinue}$

$\text{Cont}\ ))))\ +$

$[m=\#store]\ u(m).u(k)$

$([m=mi]\underline{w}_{mi}k.\text{Stop} +$

$[m \neq mi]\ (z_1)\ (\ [[s_1]]_\pi(z_1) \mid z_1(i_1).\underline{i}_1(w).\underline{w}\#store.\underline{w}m.\underline{w}(k)\ \text{ifContinue}$

$\ldots$

$(z_n)\ (\ [[s_n]]_\pi(z_n) \mid z_n(i_n).\underline{i}_n(v).\underline{w}\#store.\underline{w}m.\underline{w}(k)\ \text{ifContinue}$

$\text{Cont}\ ))))\ +$

$[m=\text{OTHER}]\ \ o(v).\underline{v}\#lookup.\underline{v}m.\underline{v}u\ \text{ifContinue}\ ))$

Whenever a slot name is sent over the channel u (in u(m)) then the OTHER-branch is used. In this branch the object is sent a #lookup message and the name of the slot to lookup is sent in $\underline{v}$m. u is the name of the channel to return the slot value on. The name of this return channel is then sent in $\underline{v}$u.

In the lookup branch the slot name is received and then the channel to return the slot value on is received. The first branch-alternative ([m=mi]) describes what happens when the slot is found in this object. Then the slot is read and the value of the slot is returned in $\underline{v}$i and the lookup is stopped by the macro Stop. The second alternative ([m≠mi]) describe what to do when the slot is not found in this object. Then a lookup is done in the object referred to in the first inheritance slot by sending it a lookup message. If the slot is found somewhere in this branch of the inheritance hierarchy a Stop macro will be called and then ifContinue will lead to no further execution of sentences in this process. If no slot is found in the branch, the next inheritance slot is searched etc. If no slot is found then no return is given to the sender of the slot name because there are no further expressions behind the ifContinue in the OTHER branch.

The #store-branch is similar except that instead of reading a slot value and returning it, the slot is updated with a new value.

Translating the bodies of objects:

The macros used for slot lookup is also used for ordering the sequence of sentences in the body of an object. ifContinue will always be true unless there is a slot lookup error, in which case no Cont is done. This means that if there is a slot lookup error the process just stops by itself in that no further sentences are executed.

The slot lookup in an inheritance tree below is done as follows:

$[[\ s\ ]]_\pi\ (o,v) == \underline{o}(u).\underline{u}s.u(i).\underline{v}i$

$[[\ \text{this}\ ]]_\pi\ (o,v) == \underline{v}o$

To model storing into a slot in the inheritance hierarchy the message #store is used.

$[[\langle\text{empty body}\rangle]]_\pi\ (o') == o'(o)$ <nothing>

$[[\ E1. \ldots En\ ]]_\pi\ (o') ==\ \ o'(o).\ [[E1]]_\pi(o)\ \text{ifContinue} \ldots\ldots\ \text{ifContinue}\ [[En]]_\pi(o)$

$[[\ s := t\ \text{clone}\ ]]\ (o) == (z)\ (\ [[t]](o,z) \mid z(i).\underline{i}(c).\underline{c}\#clone.\underline{c}(z) \mid (u)\ z(k).\underline{o}u.\underline{u}\#store.\underline{u}s.\underline{u}k.\text{Cont})$

First the value of the slot t is found and the message #clone sent to the object to be cloned. Then the name of the new object is stored into the slot s in the inheritance hierarchy of the object which input channel is o: first get a channel to the object by $\underline{o}$u. Then send #store, the slot name s and then the value k to the object.

$[[\ s_1 \ldots s_n := (\ x = y\ t\ f\ )\ ]]_\pi\ (o) == (z\ i\ j)\ (\ [[x]]_\pi(o,z) \mid (v)\ (z(i).[[y]](o,v) \mid v(j).$

$([i=j]\ (w)\ ([[t]]_\pi(o,w) \mid (u)\ (w(k).\underline{o}u.\underline{u}\#store.\underline{u}s_1.\underline{u}k \ldots \underline{o}u.\underline{u}\#store.\underline{u}s_n.\underline{u}k)) +$

$[i \neq j]\ (w)\ ([[f]]_\pi(o,w) \mid (u)\ (w(k).\underline{o}u.\underline{u}\#store.\underline{u}s_1.\underline{u}k \ldots \underline{o}u.\underline{u}\#store.\underline{u}s_n.\underline{u}k)))))$

$\text{Cont}\ )$

First the x and y slots are read and then compared. If they are similar (i=j) then the value of t is read and stored into the s-slot. If they are different the value of f is stored into the s-slot.

$$
\begin{aligned}
[[\ s\ !\ m\ (a_1 \ldots a_n)\ ]]_\pi\ (o) == \ (w)\ ([[s]]_\pi(o,w)\ \mid\ & (v_1)\ (w(r).[[a_1]]_\pi(o,v_1) \\
& \mid (v_2)\ (v_1(p_1).[[a_2]]_\pi(o,v_2) \\
& \ldots. \\
& \mid (v_n)(v_{n-2}(p_{n-1}).[[a_n]]_\pi(o,v_n) \\
& \mid (v_m)(v_n(p_n).[[m]]_\pi(o,v_m) \\
& \mid v_m(p_m)\ (z)\ (\underline{r}z.\underline{z}p_m \mid (l)\ (z(k).\underline{k}l.\underline{l}\#exe.\underline{l}(z)) \mid z(x).\underline{x}p_1 \ldots \underline{x}p_n\ \ Cont)))))\ldots)
\end{aligned}
$$

First the value of s is read into the local name r. Then the values of the parameters are read into the local names p_1 to p_n. After that the message selector is read from the slot named m into the local name p_m. Then a channel is gotten from the receiver by $\underline{r}z$. Then the message selector m is sent on this channel. Next the name of the method for the message is received in z(k), where k is the channel to the process representing the method-object. A channel is then gotten from this object by $\underline{k}l$ and across this channel is sent #exe and a channel to get a return on ($\underline{l}$(w)). The channel to the new method-copy is received by w(x), and the parameters are sent across the x-channel.

From π to Omicron

This section defines functions which translate from π-process expressions to Omicron object configurations. In the following some short hand notation is used:

> #s is short for defining a variable with the name s and value s.
> It is assumed that all object have a slot defined as follows "self → this"

In the implementation of π's synchronous message passing by asynchronous message passing in Omicron, slots are used as semaphores. A slot with value 0 is an open semaphore while slots with value ≠ 0 is a closed one.

The following Omicron objects implement π communication in Omicron when the Omicron objects are used as described further below. The strategy is to use semaphores to control the matching of one send and one receive. It is expected that the selection of which Omicron sentence to execute next is fair. If this expectation hold, then it is believed that the fairness in π message sending is modelled by the below Omicron message sending. A formal proof is necessary to show that the belief is actually true. A brief sketch of such a proof is found at the end of this appendix.

```
 top : ( [G→g], #first!#do(#top)]
|| g : ( [reg → regM, send →sendM, waitingCnl → nil, waitingName → nil, Wsema →0, Rsema →1], )
|| regM : ( [super*→g, :obj → nil, :msg →nil, next → x, x→ nil],
                Rsema, x := (Rsema = #0 #contReg #waitReg); self!next();)
|| contReg : ( [super* → regM, next →x, x→nil],
                x := (msg=waitingCnl #okReg #notOkReg); self!next();)
|| waitReg : ( [super* →regM], #g!#reg(obj, msg); )
|| okReg : ( [super*→regM], obj!#receive(obj, waitingName); Wsema := 0; )
|| notOkReg : ( [super* →regM], #g!#reg(obj, msg); Rsema := 0; )
|| sendM : [super*→g, :msg →nil, :par →nil, next→x, x→nil],
                Wsema, x := (Wsema = #0, #contS, #waitS); self!next(); )
|| contS : [super*→sendM] waitingCnl := msg; waitingName := par; Rsema := 0; )
|| waitS : [super*→sendM], #g!send(msg, par); )
```

All Omicron objects in the sequel will inherit from the object names top and in this way G will be comparable to a global variable. g is the name of an object which controls communication. The object can control communication because it holds semaphores and variables used for synchronisation and these are used in the translation. Semaphores are not part of Omicron, but can be implemented using Omicron constructs. In this case variables are used as semaphores where the slots Rsema and Wsema are read and write semaphores respectively, controlling the access to the variables waitingCnl and waitingName.

When a name is to be sent on a channel, the Omicron object modelling this synchronous sending will send the message *send* to the object names g. If Wsema is open, then it gets closed and then the variable waitingCnl is updated to hold the name of the channel. Also, waitingName is updated to hold the name to be sent on the channel. In addition the Rsema semaphore is opened. If the semaphore is closed then a new send-message is issued to g. This is an active wait-loop.

When a name may be received on a channel the Omicron object modelling this synchronous receive sends the message *reg* to the object named g. If the Rsema semaphore is open then it gets closed and it is checked if the waitingCnl matches the cannel name for reading. If the channel names match then the message receive is sent the Omicron object waiting for a π-receive-communication action. The parameter to this message is the name passed over the channel. Finally the Wsema is opened. If the channel names do not match, then the Rsema semaphore is opened and a new reg-message is send. Also, if the Rsema is closed, a new reg-message is sent to g. This also gives and active wait-loop.

Based on this synchronous communication mechanism, π-processes are translated into Omicron by functions with the following signature:

[[P]] (o) where P is a π-expression,
o is the name of the main object representing the first π sentence in P

The initial call to the translation function will be:
[[P]](first) where P is the π-expressions and first is an arbitrary object name

The different π sentences are translated as follows (When "-k" is used as part of an object name it indicates that the object name with -k in it will be unique for each translated π-sentence.):

[[0]] (k) ==
```
k : ( [do→doM], ) ||
doM : ( [:p*→nil], )
```

[[(νx).P]] (k) ==
```
k : ( [do→doM], ) ||
doM : ( [:p*→nil, x→x], #Cont-k!#do(self); ) || [[ P ]] (Cont-k)
```

[[xy.P]] (k) ==
```
k : ( [do → doM], ) ||
doM : ( [:p*→nil], G!#send(#x, #y); #Cont-k!#do(p); ) || [[ P ]] (Cont-k)
```

[[x(y).P]] (k) ==
```
k : ( [do → doM], ) ||
doM : ( [:p*→nil, receive→recM], G!#reg(self, #x); ) ||
recM : ( [:p*→nil, :y→nil], #Cont-k!#do(self); ) || [[ P ]] (Cont-k)
```

[[P | Q]] (k) ==
```
k : ( [do→doM], ) ||
doM : ( [:p*→nil], #pCont-k!#do(p); #qCont-k!#do(p); ) ||
[[ P ]] (pCont-k) || [[ Q ]] (qCont-k)
```

[[P + Q]] (k) ==
```
k : ( [do → doM, sema→0, doP→doPM, doQ→doQM], ) ||
doM : ( [:p*→nil, owner*→k], #k!#doP(self); #k!#doQ(self) ) ||
doPM : ( [:p*→nil, owner*→k, next→x, x→nil],
        sema,x:=(sema = 0 #Pcont-k, #Pnot-k); self!next(self); ) ||
Pcont-k : ( [:p*→nil], #pCont-k!#do(p); ) ||
Pnot-k : ( [:p*→nil], ) ||
doQM : ( [:p*→nil, owner*→-k, next→x, x→nil],
        sema,x:=(sema = 0 #Qcont-k, #Qnot-k); self!next(self); ) ||
Qcont-k : ( [:p*→nil], #qCont-k!#do(p); ) ||
Qnot-k : ( [:p*→nil], ) ||
[[ P ]] (pCont-k) || [[ Q ]] (qCont-k)
```

[[[x=y]P]] (k) ==
```
k : ( [do→doM], ) ||
doM : ( [:p*→nil, next→x, x→nil] x:=(x=y #next-k #stop-k); self!next(p);) ||
next-k : ( [:p*→nil], #Cont-k!#do(p); ) ||
stop-k : ( [:p*→nil], ) ||
[[ P ]] (Cont-k)
```

[[xy.*P]] (k) ==
```
k : ( [do→doM ], ) ||
doM : ( [:p*→nil, copy → nil], G!#send(#x, #y);
        copy := #k clone; #Cont-k!#do(p); copy!#do(p); ) ||
[[ P ]] (Cont-k)
```

[[x(y).*P]] (k) ==
```
k : ( [do→doM ], ) ||
doM : ( [:p*→nil, copy → nil, receive→recM], G!#reg(self, #x);
        copy := #k clone; copy!#do(p); ) ||
recM : ( [:p*→nil, :y→nil], #Cont-k!#do(self); ) ||
[[ P ]] (Cont-k)
```

Sketching a proof of correct translations

(Walker 1991) translates an object-oriented language to π-calculus. The translation is done much along the same lines as the above translation of Omicron into π. (Walker 1992) gives the operational semantics of an object-oriented language and also gives the semantics by a translation of the language to π. The paper finally shows a close correspondence between the two semantics. Showing the correspondence between the Omicron operational semantics and the π semantics can be done in a similar way. The correspondence will then be shown as follows:

Let C denote Omicron system expressions and P denote π agents. Let $\rightarrow$ be the transition relation between Omicron expressions and let $\rightarrow^*$ denote zero or more such transitions Let $\Rightarrow$ denote the reflexive and transitive closure of silent π transitions τ, and ~ the relation of strong bisimilarity on π agents from (Milner et al. 1989a). Let C_0 be the initial Omicron system. Then:

1) Any computation $C_0 \rightarrow C_1 \rightarrow \ldots \rightarrow C_n$ is directly mirrored by a derivation
$[[C_0]] \Rightarrow \sim [[C_1]] \Rightarrow \sim \ldots \Rightarrow \sim [[C_n]]$ which, because of the finer grain of action, typically involves more transitions.

2) If $([[C_0]] \Rightarrow P)$ then for some C, $C_0 \rightarrow^* C$ and $(P \Rightarrow \sim [[C]])$

A similar exercise must be done to show that the translation from π to Omicron gives the semantics of π corresponding to the operational semantics given in (Milner et al. 1989a).

These proofs are left for further study.

Appendix C: References

1 Object-oriented languages

Beta	B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen and K. Nygaard, (1987) *The BETA Programming Language* in Research Directions in Object-Oriented Programming, ed: B. Shriver and P. Wegner, MIT Press
C++	Bjarne Stroustrup, (1986) *The C++ Programming Language,* Addison Wesley
CLOS	Sonya E. Keene, (1989) *Object-oriented Programming in Common LISP,* Addison-Wesley, Reading, Mass., ISBN : 0-201-17589-4
Dylan	Apple, (1992) *Dylan, an object-oriented dynamic language,* Apple Computer, 1 Main Street, Cambridge, MA 02142, To order: email: dylan-manual-request@cambridge.apple.com
Eiffel	Bertrand Meyer, (1988) *Object-Oriented Software Construction,* Prentice-Hall
Java	See URL http://www.javasoft.com for updated lists of books, publications and online documentation.
SELF	David Ungar and Randall B. Smith, (1987) *Self: The Power of Simplicity*, OOPSLA 1987, published in ACM SIGPLAN Notices 22:12, pp. 227-241.
Simula	Graham M. Birtwistle, Ole-Johan Dahl, Bjørn Myhrhaug and Kristen Nygaard, (1973) *Simula BEGIN,* Potrocelli/Charter, New York, ISBN : 0-88405-340-7
Smalltalk-80	Adele Goldberg and Dave Robson, (1983) *Smalltalk-80: The Language and Its Implementation,* Addison-Wesley, Reading, Mass.

2 Object-oriented methods

BON	Kim Waldén and Jean-Marc Nerson, (1995) *Seamless Object-Oriented Software Architecture,* Prentice Hall, ISBN : 0-13-031303-3
Catalysis	Book under development by Desmond D'Souza, see URL http://www.iconcomp.com/catalysis for reports and updated information on publications
Foundation	James Martin and James J. Odell, (1995) *Object-Oriented Methods, a Foundation,* Prentice Hall, ISBN : 0-13-630856-2
Fusion	Derek Coleman, Patric Arnold, Stephanie Bodoff, Chris Dollin, Helena Gilchrist, Fiona Hayes and Paul Jeremaes, (1994) *Object-Oriented Development. The Fusion Method,* Prentice Hall, 313, ISBN : 0-13-338823-9
Objectory	Ivar Jacobson, Patric Jonsson and Gunnar Övergaard, (1992) *Object-Oriented Software Engineering,* Addison-Wisley, ISBN : 0-201-54435-0 Later papers on Objectory referred to in this thesis: Ivar Jacobson, Stefan Bylund, Patric Jonsson and Staffan Ehneboom, (1995) *Using contracts and use cases to build pluggable architectures* in Journal of Object-Oriented Programming May 1995, 8:2, pp. 18-76 Ivar Jacobson, (1995) *Use Cases in Large-Scale Systems* in Report on Object Analysis and Design (ROAD) March-April 1995, 1:6, pp. 9-12
OMT	James Rumbaugh, Michael Blaha, William Premerlani, Fredrick Eddy and William Lorensen, (1991) *Object-Oriented Modelling and Design,* Prentice Hall, Englewood Cliffs, New Jersey 07632, ISBN : 0-13-629841-9

OOram

Trygve Reenskaug, Per Wold and Odd Arild Lehne, (1995) *Working with Objects, The OOram Software Engineering Method,* Prentice Hall, ISBN : 1-884777-10-4

First report on OOram:
Trygve Reenskaug and Else K. Nordhagen, (1989) *The design and description of complex, object-oriented systems,* SI - Senter for Industriforskning, 89- 272-1, ISBN: 82-411-0193-7

First paper on OOram:
Trygve Reenskaug, Egil Andersen, Arne Jørgen Berre, Anne Hurlen, Anton Landmark, Odd Arild Lehne, Else Nordhagen, Eirik Næss-Ulseth, Gro Oftedal, Anne Lise Skaar and Pål Stenslet, (1992) *OORASS: Seamless Support for the Creation and Maintenance of Object-Oriented Systems* in Journal of Object-Oriented Programming October 1992, 5:6, pp. 27-41

Paper on aspects of OOram referred to in this thesis:
Egil P. Andersen and Trygve Reenskaug, (1992) *System Design by Composing Structures of Interacting Objects*, ECOOP 1992, published in LNCS 615, pp. 133-152.

RDD: Responsibility Driven Design

Rebecka Wirfs-Brock, Brian Wilkerson and Lauren Wiener, (1990) *Designing Object-Oriented Software,* Prentice Hall, ISBN : 0-13-629825-7

Syntropy

Steve Cook and John Daniels, (1994) *Designing Object Systems,* Prentice Hall, Hemel Hempstead, UK,
ISBN : 0-13-203860-9

UML

See URL http://www.rational.com for reports and lists of books and publications

Not a method, but a way to describe designs:
Design Patterns or just Patterns

Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, (1994) *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison-Wesley, 416, ISBN : 0-201-63361-2

Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, (1993) *Design patterns: Abstraction and reuse in object-oriented design*, European Conference on Object-Oriented Programming 1993, Kaiserslautern, Germany, published in LNCS 707, pp. 406-431.

3 Related publications

The following list contains publications by the author which are related to the topics of this thesis.

Example of object component design:

> Else Nordhagen, (1987) *A New Text Editor Implementation for Smalltalk-80*, Center for Industrial Research (SI, now SINTEF), Oslo, Norway, EKI-N-42

General architecture for software composition:

> Else Nordhagen, (1989) *Generic Object-Oriented Systems*, TOOLS 1989, Paris, France, published in TOOLS'89 Proceedings, pp. 131-140.

Translation of an object-oriented language to the π-language:

> Else K. Nordhagen, (1992) *π-Calculus semantics for a Smalltalk like language*, NIK 1992, Tromsø, Norway, pp. 43-54.

Architecture for sharing objects, designing programs and reusing code:

> Else K. Nordhagen, (1993) *Four types of types for objects*, NIK 1993, Halden, Norway, pp. 1-12.
>
> Latest version:
>
> Else K. Nordhagen, (1995) *The COIR Architecture for Flexible Software Components and Systems*, Department of Informatics, University of Oslo, Norway, Research Report 197, ISBN: 82-7368-108-4,

The OOram design method for object component systems (latest reference (Reenskaug et al. 1995)):

> Trygve Reenskaug and Else K. Nordhagen, (1989) *The design and description of complex, object-oriented systems*, SI - Senter for Industriforskning, 89-272-1, ISBN: 82-411-0193-7
>
> Trygve Reenskaug, Egil Andersen, Arne Jørgen Berre, Anne Hurlen, Anton Landmark, Odd Arild Lehne, Else Nordhagen, Eirik Næss-Ulseth, Gro Oftedal, Anne Lise Skaar and Pål Stenslet, (1992) *OORASS: Seamless Support for the Creation and Maintenance of Object-Oriented Systems* in Journal of Object-Oriented Programming October 1992, 5:6, pp. 27-41

Index: