Software Constraint Models – A Means to Improve Maintainability and Consistency

Dag I.K. Sjøberg
Department of Informatics, University of Oslo, Norway
dagsj@ifi.uio.no

Abstract

As application systems live longer and grow in size and complexity, there is an ever increasing need for methodologies, models and tools that can aid software builders in developing maintainable, correct and consistent systems. Imposing constraints, representing architectures, conventions, guidelines, etc., on the software is one step in that direction. One may distinguish between constraints within programs, constraints between programs, and constraints between programs and secondary storage. A coherent set of constraints are collected in a *software constraint model*. Automatic verification tools are crucial to the usefulness of such models. The paper describes a constraint model and a corresponding verification tool that have been developed in a persistent programming environment.

Keywords: programming environments, maintenance, consistency, software constraints

1 Introduction

Large, long-lived and data-intensive application systems that satisfy a complete area of information-processing requirements, such as management information systems, health management systems, CAD/CAM systems, CASE environments, etc. must continuously undergo change in order to reflect change in their environments [18, 23, 27, 28]. To satisfy new requirements, code must be modified, which in turn may cause its structure to deteriorate [21] and introduce inconsistencies. Making consistent changes to software is difficult; it is easy to cause a mutation, but very hard to generate a viable one, particularly if multiple copies have been shipped, etc. A change in one place may have unintended effects elsewhere; even minor local changes can have global impact [7]. Included in the consequences are new errors (the ripple effect).

As a means to manage maintenance and correctness, this paper introduces the notion of a *software constraint model*, denoting a coherent set of application independent constraints defined over all the software used in an application system.[1] The consistency of a system is evaluated relative to such a model. The constraints may support program architectures, naming conventions and other aspects of a programming methodology.

---

[1] The constraints of a software constraint model can be compared to what Date [13] refers to as "general integrity rules" or "metarules" in databases. "Specific integrity rules" express constraints in the real-world application; general integrity rules are independent of a specific application but may depend on the type of data model being used (e.g. the relational data model).

The compiler of a programming language already performs many forms of consistency checks such as type checking, ensuring declaration and unique naming (within a scope) of identifiers, etc. The constraint model is concerned with complementary checks, such as those between programs and those between programs and data on a secondary storage. The model supports standardisation and disciplined software construction and maintenance; commonly agreed rules and conventions that should be adhered to in a given programming environment are made explicit by the constraint model. Well-structured software is a requirement for easy maintenance in the future [17, 21]. The following three issues are crucial to the success of a constraint model.

i) *The actual constraints of the model* The constraints will depend on the programming language(s), the programming environment, the programming methodology being used, etc.

ii) *Supporting tools* Automatic system analysis and constraint verification are desirable, particularly for large and complex systems.

iii) *A constraint specification language* In practice, a general constraint model should be modified and extended according to increased knowledge about the development process and changed working practices. Local adaptations will typically be necessary. Achieving flexibility calls for the provision of a general constraint specification language.

The paper is organised as follows. Section 2 presents examples of generally applicable constraints belonging to three different categories. A particular software constraint model proposed in a persistent programming environment is discussed in Section 3. An environment supporting this model is the issue of Section 4. Section 5 describes related work. Section 6 concludes.

2 Constraint Categories

This section discusses categories of constraints and gives examples of constraints applicable to most programming environments, e.g. constraints that aim to prevent redundant or duplicated code. In a particular programming environment, where possibly a particular methodology is adhered to, software builders and maintainers would benefit from a tailored constraint model. An example is the constraint model developed in a persistent programming environment described in Section 3.

A violation of a constraint could be a logical error or could just indicate a situation that might eventually cause problems. For example, redundant type and value declarations do not affect the functionality of a program but should be avoided since they may cause confusion when someone tries to understand the program. The programs become unnecessarily large and complex, which may also impair performance and maintainability.

A constraint verification tool (Section 4) should give warnings when violations are detected, similarly to the way modern grammar checkers work. The tool should feature optional selection of the constraints; programmers should be able to “switch off” the check of individual constraints. For example, a programmer may know that certain constraints will not be adhered to during periods of development (typically during initial construction) and may wish to avoid the noise of unnecessary inconsistency messages.

A constraint model may operate at several levels, as indicated by the following three categories:

i) constraints within programs[2]

ii) constraints between programs

iii) constraints between programs and a persistent storage

The following are examples of constraints within programs: “a declared identifier should be used”, “a variable should be updated” (otherwise the identifier should be a constant), “a variable should be accessed after update”, etc.

Constraints between programs operate at the application level. One example is constraints involving type definitions.[3] For example, “all type definitions should be used”, in particular, “some code should create instances of the type”, and “all components of a type definition should be used.” Moreover, “a type name should be declared only once within an application system.”[4] The reason for this constraint is that multiple declarations of type names are confusing, require unnecessary compilation and are a potential problem concerning change. Maintaining consistency requires that all declarations describing the same concept (e.g. *Person*) must be changed if the intention is to modify the implementation of the concept (e.g. add a new attribute). It is difficult to arrange that when several programmers (responsible for several components) who require use of a common type, each write out equivalent type definitions (particularly if they are complex). It is even harder to ensure that when the type is amended, the same amendments are applied in every usage context. One concept should therefore be represented by only one type definition.

Other constraints between programs may concern code involving data files, relations in a database or other *storing constructs* for persistent data. For example, “a storing construct created in one program should also be written to and read from (unless intended for export only) by at least one program”, “a storing construct should be created in exactly one program”, “a storing construct should be deleted in one program only”, “for each storing construct used by a program there should be a corresponding program that creates the construct”, etc.

The paragraphs above discussed constraints that can be checked by static analysis of source code. In contrast, the constraints in the third group concern relationships between source code and storing constructs present on a persistent storage at the time of analysis. For example, “each storing construct present on a persistent storage should have exactly one corresponding program that creates the construct (except for storing constructs imported from an external application such as a library)”, “at least one program should use the storing construct”, “a storing construct used in a program should be present on a persistent storage (unless something else is indicated by the programmer)”, etc. A check of the last constraint may prevent run-time errors. For example, a removal or renaming of a file will often not be detected before a program attempts to access the file at run-time.

Names are central to system builders’ thinking and thus influence the way software is organised. Meaningful names are important for problem solving, understanding of

---

[2] A *program* in this context is a unit of compilation, typically contained in a single file, but may be represented by several files (e.g. assembled first by a pre-processor, held in a source code control system like RCS, etc.) or may be extracted from one file. The term *module* is often used in the literature synonymously with our definition of program.

[3] The term *type definition* used here corresponds to *class* in the object-oriented model, to *relation type* in the relational model, etc.

[4] This is a particular problem in systems where types can be defined in different scopes, but in relational systems, for example, a relation name must always be unique due to flat name space.

semantic structure and memorisation [8, 33]. Within an application people should use names with a consistent intended meaning. The choice of names for identifiers is crucial to the readability of programs and is particularly important when trying to administer and manage change. Orthogonal to the constraint categories described above are conventions for naming files, programs, directories, type definitions, procedures, variables, etc., for example, "all type definitions should start with an upper case letter." Various naming guidelines have been proposed in the literature [1, 15]. The important point is that there is a naming scheme, not its exact form.

Two criteria must be present before a constraint should be introduced. First, an argument should justify why a constraint should be adhered to. For example, regarding the constraint that a declared identifier should be used, a study of FORTRAN programs found a correlation between the proportion of unused variables and fault rate [10]. A motivation for many of the constraints discussed in this paper can be found in [29].

Second, it is necessary to ensure that the constraints are of practical value. Therefore, we need to investigate current practice to identify constraints that are violated (otherwise there is no problem). For example, in a study of production PL/1 programs 28% of all identifiers were reported unused [16]. In a comprehensive study of 20 applications built in a persistent programming language, the adherence to ten potential constraints (see Section 3) were investigated [31]. The study showed that more than one third of all variables were never updated and could therefore have been declared constants, 29% of all types were repeatedly declared, etc. The proportion of violation varied between 4% and 35%, which confirmed the usefulness of the constraints.

The measurements reported in the previous paragraph show the proportion of violations. An evaluation of the practical value of a constraint should also take into account the use frequency of the language constructs that the constraint relate to. Studies of conventional languages show that only a small subset of the languages is used in 90 per cent of all statements [16, 19, 32]. Constraints relating to this subset are generally more important than other constraints.

3 SPASM – A Software Constraint Model in a Persistent Programming Environment

The Structured Persistent Application System Model (SPASM) is an example of a tailored constraint model defined in a persistent programming environment. The concept of persistence tackles the mismatch between database systems and programming languages [2, 12]; a uniform model for representations and operations on persistent and transient data is provided. Tools, programs and data may reside in the same store. Many of the benefits of persistent language technology have been described in the literature [3, 5, 6].

SPASM is couched in terms of the orthogonally[5] persistent programming language Napier88 [26]. (A similar model may operate in an object-oriented database context [4].) A persistent store is organised in *environments*, which are extensible collections of *bindings*. Each binding is a quadruple: a unique identifier, a type, a value, and a constancy. The Napier88 system provides a mechanism for storing pre-compiled type definitions in a database analogously to the meta-database used with conventional databases to hold schemata. Programs can be compiled against such type databases.

Programmers who share a common view of how to develop applications in their environment form a particular programming culture. Such cultures may differ considerably

[5] All data values, whatever their type (including procedures), are allowed the full range of persistence.

from group to group even though the programming language is the same. Some of the SPASM constraints are explicit formulations of rules and conventions in a programming culture already adhered to by experienced persistent programmers. Other constraints have been defined as a result of the inconsistencies detected in the study reported in [31]. All the SPASM constraints are believed to improve the maintainability of application systems and can be both supported and exploited by change management tools.

As a means to improve the way applications are organised around the persistent store, SPASM restricts each program to perform only one kind of operation on the store. Any program should belong to exactly one of the following categories:

- *Insert-program* – inserts at least one binding into an environment in the persistent store but neither updates a persistent location nor deletes any binding.
- *Update-program* – updates at least one persistent location but neither inserts nor deletes any binding.
- *Drop-program* – deletes at least one binding but neither updates a persistent location nor inserts any binding.[6]
- *Startup-program* – uses at least one binding but neither changes the binding to a persistent location, nor inserts or deletes any binding. A startup-program's distinguishing feature is that it does not change any of the bindings in any persistent environment; it typically invokes an interactive menu or any persistent procedure.
- *Type-program* – its contents are exclusively type definitions.

Several constraints help ensure adherence to an incremental construction methodology based on updatable persistent locations [14]. Using the methodology, insert-programs create stub locations in environments, one for each component of the application. For each component, an update-program finds bindings to locations of components required by the component under construction. The update-program creates the new component with bindings to these locations, and updates the component's location with the newly constructed version.

The programming methodology describes criteria of the construction and maintenance process of the product (the application system); SPASM describes criteria of that product. SPASM and the methodology mutually support each other (Figure 1). Obtaining an application system compliant with SPASM is simpler (but is still not guaranteed) if the methodology is adhered to during construction and maintenance.

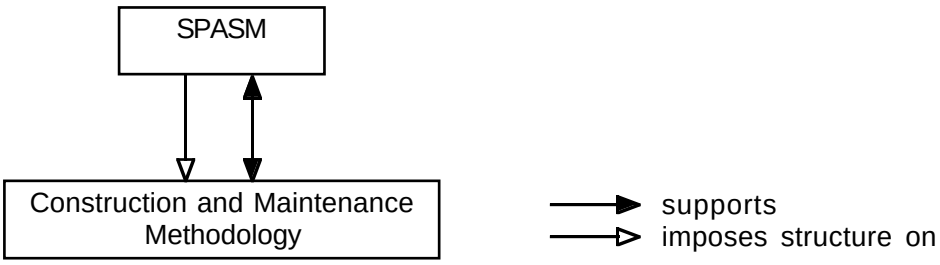

Figure 1: Relationship between SPASM and the methodology

6 "Delete" is called "drop" in Napier88 terminology. These terms are used synonymously in this paper.

The following are examples of SPASM constraints: “programs and data in the persistent store should be used in at least one application program”, “a persistent variable binding should be updated in at least one application program”, “an update-program should update only one persistent location (typically containing a procedure) or a coherent group of persistent locations contained in the same environment”, etc.

Some of the constraints can be expressed in an Entity-Relationship diagram[7] (Figure 2) describing relationships between the type definitions, program categories, persistent locations, environments and other kinds of binding. The arrow texts should be read from the entity on the left of the relationship to the entity on the right. The diagram shows, for example, that a persistent location is associated with exactly one update-program, but one update-program can update several persistent locations.

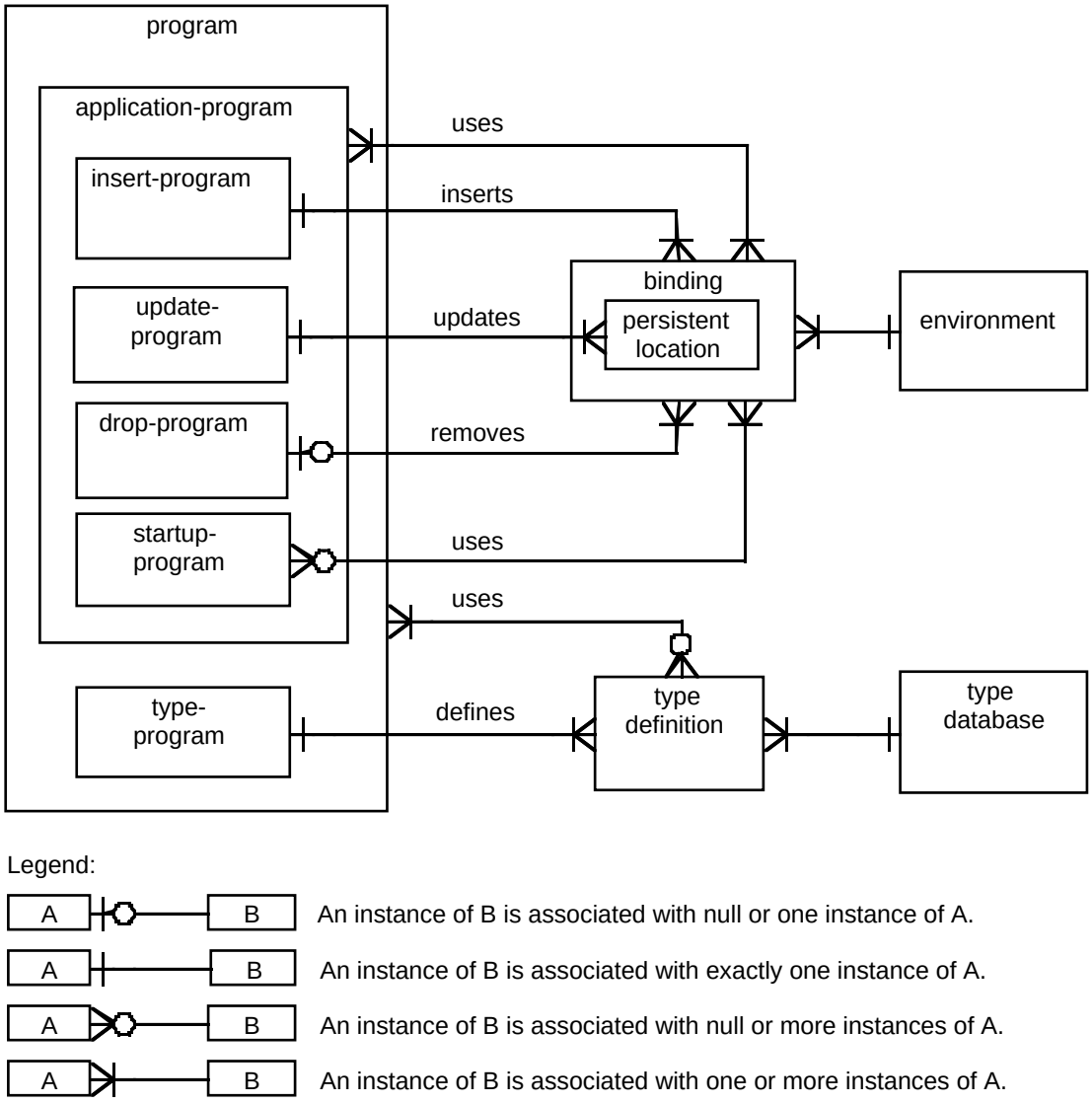

Figure 2: ER diagram of programs, bindings and type definitions

[7] This kind of Entity-Relationship diagram is one of many variants of the original definition [11].

Another category of SPASM constraints concerns ordering. First, “there should be a partial order [20] among the type-programs.” This is a requirement for compilation. Determining a partial order may be a non-trivial task if there are several type-programs with dependencies between them. Second, “there should be a partial order among the insert-programs.” To enable system installation in a persistent programming environment, the bindings used by one insert-program must already have been inserted into the persistent store by another insert-program before the former can be executed. This is always possible if a partial order exists among the insert-programs.

SPASM was tailored to a persistent programming paradigm. Application development according to other programming paradigms will have other particular problems, and a constraint model would have to be developed accordingly, but many of the SPASM constraints represent ideas that are generally applicable.

4 Constraint Verification

The success of a constraint model depends heavily on a supporting environment that automatically checks constraint adherence and provides relevant information in the case of violation. Figure 3 shows the components of such an environment.

EnvMake is a persistent programming tool that, among other things, verifies programs against the constraints of SPASM. In the case of constraint violation, EnvMake informs the programmer about the kind and source of violation. As an example, Figure 4 shows EnvMake's output after a check of the constraint that a program should belong to one category only. The two programs `newPerson.N` and `personInfo.N` perform more than one kind of operation on the persistent store.

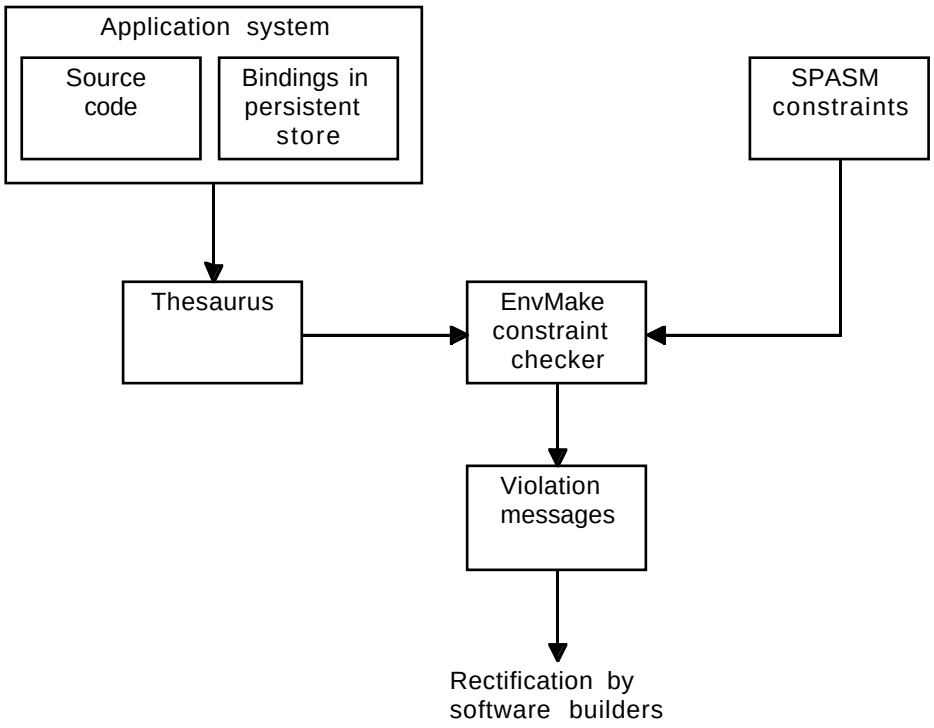

Figure 3: A constraint support environment

```
CONSTRAINT VIOLATION: MORE THAN ONE STORE OPERATION WITHIN A PROGRAM
-------------------------------------------------------------------------------
PROGRAM                         BINDING                     OPERATION
-------------------------------------------------------------------------------
Personnel/newPerson.N           newPerson                   insert
Personnel/newPerson.N           newPerson                   drop
Personnel/personInfo.N          personInfo                  insert
Personnel/personInfo.N          personInfo                  drop
Personnel/personInfo.N          personInfo                  update
-------------------------------------------------------------------------------
Number of programs: 2
-------------------------------------------------------------------------------
```

Figure 4: A constraint violation table produced by EnvMake

EnvMake does not analyse the applications directly but uses the information stored in the *thesaurus* – a meta-database that integrates the notions of data dictionary in the database area and cross-referencer in the programming language area [30]. The thesaurus holds fine-grained, enhanced cross-reference information about all user-introduced names occurring in the source programs of an application and the names of the bindings to code and other data in the associated persistent stores. Information relevant to EnvMake is: *name, type, constancy* of an identifier and *usage* and *context* of identifier occurrences. Usage indicates how the identifier is being used, e.g. declaration or use of a type identifier, or declaration, left context or right context of a value identifier. Context indicates whether the identifier occurs in an operation on the persistent store or as a declaration of a type parameter, procedure parameter, structure field, variant tag, etc. or as a dereferenced structure field, projected variant, etc.
All the contents of the thesaurus are automatically maintained. The whole application system is analysed, and the thesaurus updated, regularly at times specified by the user, for example daily at 02:00. A full analysis and update can also be initiated at any time.

Both EnvMake and the thesaurus tool are implemented in Napier88. The provision of persistence enables the thesauri, as well as the tools, to be contained and integrated in the persistent store like any other values. It should be emphasised that because the thesauri are in the same store, the thesaurus can be automatically constructed and updated with guarantees of consistency with the data that they describe. Moreover, constraints concerning the whole processing environment can easily be verified, e.g. constraints between programs and other objects (bindings) in a persistent store.

There are generally many ways of violating the SPASM constraints, and for each violation there are generally several ways of rectification. For example, if a type definition is never used, the programmer could modify or create a new program that will use the type definition, or she could delete the type definition. One may envisage a tool that automates the latter but not the former. In general, since it is a semantic problem how to rectify inconsistent states, fully automatic supporting tools seem infeasible, but future research should investigate the possibility of semi-automatic tools that interact with the programmer.

5 Related Work

Many modern compilers, e.g. [9], can be instructed to give warnings if constraints within programs such as those mentioned in Section 2 are violated. However, it might be easier to use a tailored constraint analysis tool that is decoupled from the compiler, and which would typically be invoked after the program has been verified as syntactically correct.

The *Law of Demeter* [22] intends to improve the style and structure of object-oriented programs. The law imposes constraints on how member functions are allowed to call each other, which may reduce coupling among classes. Two transformations have been defined that modify any program to become consistent according to the law in the case of violation.

To support Ada software builders in incremental development, the AdaPIC tool set [34] provides consistency analyses on interfaces within and among modules. Consistency violations are divided into *errors* and *anomalies*. The latter indicate situations that might eventually cause problems.

Meyers and Lejter [25] have identified several conditions in C++ programs that indicate programming errors, but that are not language errors detected by a compiler (even though there are compilers that give warnings for some of the conditions). To help prevent such errors, Meyers *et al.* [24] developed the C++ Constraint Expression Language (CCEL) – a meta-language for C++ that enables software builders to specify a whole range of constraints on programs. Violations are automatically detected. In CCEL one can also specify parts of a system (files, functions or classes) where the constraints should (or should not) apply. CCEL is a *language* for constraint specification and verification. Hence, the support for extensibility is more sophisticated than that of EnvMake, where the verification is hard-wired into the tool.

Common to the work described above is that the constraints concern source code only. In contrast, SPASM and EnvMake also include constraints that involve components on a secondary storage; the persistent language technology and the thesaurus information enable formulation and verification of constraints concerning the whole processing environment.

6 Conclusions and Future Work

Constructing and maintaining large and long-lived application systems, with possibly many people involved, are complex tasks. To achieve maintainability and correctness, it is crucial software builders adhere to practices and conventions that have proved useful. Standardisation may eliminate peculiar programming styles and may simplify collaboration, software reuse, etc. The notion of a software constraint model introduced in this paper is a means to achieve such standardisation. The constraints should be formal enough to be automatically verified by a supporting tool.

The exact form of a constraint model will depend on the actual culture and environment, even though the paper describes some constraints that are generally applicable. An example of a constraint model is SPASM, which was defined in a persistent programming environment and which encourages compliance with a certain persistent programming methodology. As is often the case for guidelines, adhering to SPASM seems awkward for small applications, and some developers may feel that their personal programming style is unnecessarily constrained. However, it is an investment that will pay off in the long run, particularly in a community of software builders constructing and maintaining large and complex application systems.

A tool called EnvMake has been tailored to support the SPASM model. For each violation of a constraint, EnvMake gives a warning and indicates the source of the violation. It is then the responsibility of the programmer to rectify the inconsistent state. (An enhanced version of EnvMake could often offer a solution.)

If a programmer complies with SPASM, EnvMake provides more assistance than just checking the SPASM constraints. One example (not discussed in this paper) is automatic build management, including smart recompilation [29].

The reported research has been conducted in the context of a persistent programming language (Napier88). Since the same language is used to represent and manipulate transient and persistent data (including code), it is easy to formulate and verify constraints between programs, and constraints between programs and secondary storage. It is more complicated, but still possible, to implement similar constraints in the context of conventional language technology.

The current implementation of the SPASM verification is hard-wired into EnvMake. Of course, the default SPASM constraints will not be adequate or sufficient in all cases or in all cultures. Hence, a major issue for future work is how to design and implement a constraint specification language (similar to CCEL [24]) that is tailored for and exploits the software engineering features of persistent language technology.

The constraint models discussed in this paper focus on system implementations. Future constraint models may be extended to also operate on design structures, data model specifications, etc., enabling other phases of the life cycle to be supported as well.

Acknowledgements

Numerous stimulating discussions with Malcolm Atkinson and Ray Welland were essential for the ideas and work reported in this paper. The St Andrews persistent programming team provided the underlying language technology and have made several useful comments on the SPASM model. The author was supported by the Research Council of Norway.

References

[1] Anand, N. “Clarify Function!”. *ACM SIGPLAN Notices*, Vol. 23, No. 6, pp. 69-79, 1988.

[2] Atkinson, M.P. “Programming Languages and Databases”. In: Fourth International Conference on Very Large Data Bases (Berlin, West Germany, 13th–15th September, 1978), Yao, S.B. (editor), pp. 408–419, IEEE and ACM, 1978.

[3] Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. and Morrison, R. “An Approach to Persistent Programming”. *Computer Journal*, Vol. 26, No. 4, pp. 360–365, November 1983.

[4] Atkinson, M.P., Bancilhon, F. ., DeWitt, D., Dittrich, K., Maier, D. and Zdonik, S. “The Object-Oriented Database System Manifesto”. In: *First International Conference on Deductive and Object-Oriented Databases (Kyoto, Japan, 4th–6th December, 1989)*, pp. 40–57, Elsevier Science Publishers B.V., 1989.

[5] Atkinson, M.P. and Buneman, O.P. “Types and Persistence in Database Programming Languages”. *ACM Computing Surveys*, Vol. 19, No. 2, pp. 105–190, 1987.

[6] Atkinson, M.P., Chisholm, K.J. and Cockshott, W.P. “PS-algol: An Algol with a Persistent Heap”. *ACM SIGPLAN Notices*, Vol. 17, No. 7, pp. 24–31, July 1982.

[7] Atkinson, M.P., Sjøberg, D.I.K. and Morrison, R. “Managing Change in Persistent Object Systems”. In: *First JSSST International Symposium on Object Technologies for Advanced Software (Kanazawa, Japan, 4th – 6th November, 1993)*, Nishio, S. and Yonezawa, A. (editors), pp. 315-338, Lecture Notes in Computer Science 742, Springer-Verlag, 1993.

[8] Barnard, P., Hammond, N.V., MacLean, A. and Morton, J. “Learning and Remembering Interactive Commands in a Text-Editing Task”. *Behaviour and Information Technology*, Vol. 1, pp. 347–358, 1982.

[9] Borland C++, User's Guide. Version 3.1, Borland International Inc., 1992.

[10] Card, D.N., Church, V.E. and Agresti, W.W. “An Empirical Study of Software Design Practices”. *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 2, pp. 264-270, 1986.

[11] Chen, P.P. “The Entity-Relationship Model – Toward a Unified View of Data”. *ACM Transactions on Database Systems*, Vol. 1, No. 1, pp. 9–36, 1976.

[12] Copeland, G. and Maier, D. “Making Smalltalk a Database System”. *Proceedings of the ACM SIGMOD 1984 Conference on the Management of Data, ACM SIGMOD Record*, Vol. 14, No. 2, pp. 316–325, June 1984.

[13] Date, C.J. *An Introduction To Database Systems*. The Systems Programming Series, Addison Wesley, 1990.

[14] Dearle, A., Cutts, Q. and Connor, R. “Using Persistence to Support Incremental System Construction”. *Microprocessors and Microsystems*, Vol. 17, No. 3, pp. 161-171, April 1993.

[15] Einbu, J.M. “An Architectural Approach to Improved Program Maintainability”. *Software—Practice and Experiences*, Vol. 18, No. 1, pp. 51-62, January 1988.

[16] Elshoff, J.L. “An Analysis of some Commercial PL/1 Programs”. *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 2, pp. 113–120, June 1976.

[17] Gibson, V.R. and Senn, J.A. “System Structure and Software Maintenance Performance”. *Communications of the ACM*, Vol. 32, No. 3, pp. 347–358, March 1989.

[18] Jørgensen, M. “An Empirical Study of Software Maintenance Tasks”. *Journal of Software Maintenance* (to appear).

[19] Knuth, D.E. “An Empirical Study of FORTRAN Programs”. *Software – Practice and Experience*, Vol. 1, No. 2, pp. 105–133, April–June 1971.

[20] Knuth, D.E. *Fundamental Algorithms*. The Art of Computer Programming, Addison-Wesley, Vol. 1, January 1973.

[21] Lehman, M.M. and Belady, L. *Program Evolution, Processes of Software Change*. A.P.I.C. Studies in Data Processing No. 27, Academic Press, London, 1985.

[22] Lieberherr, K.J. and Holland, I.M. “Assuring Good Style for Object-Oriented Programs”. *IEEE Software*, pp. 38-48, September 1989.

[23] Lientz, B.P. and Swanson, E.B. *Software Maintenance Management: A Study of the Maintenance in 487 Data Processing Organizations*. Addison-Wesley Publishing Company, Reading, Mass., 1980.

[24] Meyers, S., Duby, C.K. and Reiss, S.P. “Constraining the Structure and Style of Object-Oriented Programs”. In: *First Workshop on Principles and Practice of Constraint Programming (PPCP93) (Newport, RI, USA, 28th–30th April, 1993)*, 1993.

[25] Meyers, S. and Lejter, M. “Automatic Detection of C++ Programming Errors: Initial Thoughts on a lint++”. In: *USENIX C++ Conference Proceedings 1991)*, pp. 29-40, April 1991.

[26] Morrison, R., Brown, F., Connor, R. and Dearle, A. The Napier88 Reference Manual. Technical Report PPRR-77-89, Universities of Glasgow and St Andrews, 1989.

[27] Nosek, J.T. and Palvia, P. “Software Maintenance Management: Changes in the last Decade”. *Journal of Software Maintenance*, Vol. 2, No. 3, pp. 157–174, 1990.

[28] Sjøberg, D.I.K. “Quantifying Schema Evolution”. *Information and Software Technology*, Vol. 35, No. 1, pp. 35–44, January 1993.

[29] Sjøberg, D.I.K. Thesaurus-Based Methodologies and Tools for Maintaining Persistent Application Systems. Ph.D. Thesis, Department of Computing Science, University of Glasgow, July 1993.

[30] Sjøberg, D.I.K., Atkinson, M.P. and Welland, R. “Thesaurus-Based Software Environments”. In: *Workshop on Software Engineering and Databases in conjunction with the 16th International Conference on Software Engineering (Sorrento, Italy, 16th – 17th May, 1994)*, 1994.

[31] Sjøberg, D.I.K., Cutts, Q., Welland, R. and Atkinson, M.P. “Analysing Persistent Language Applications”. In: *Sixth International Workshop on Persistent Object Systems (Tarascon, Provence, France, 5th – 9th September, 1994)*, Atkinson, M.P., Benzaken, V. and Maier, D. (editors), Springer-Verlag and British Computer Society, 1994.

[32] Saal, H.J. and Weiss, Z. “An Empirical Study of APL Programs”. *Computer Languages*, Vol. 2, No. 3, pp. 47–59, 1977.

[33] Weiser, M. and Shneiderman, B. “Human Factors of Computer Programming”. In: *Handbook of Human Factors*. Salvendy, G. (editor), pp. 1398–1415, John Wiley & Sons, 1987.

[34] Wolf, A.L., Clarke, L.A. and Wileden, J.C. “The AdaPIC Tool Set: Supporting Interface Control and Analysis Throughout the Software Development Process”. *IEEE Transactions on Software Engineering*, Vol. 15, No. 3, pp. 250–263, March 1989.