UNIVERSITY OF OSLO
Department of Informatics

GMSim: A generalized semi-Markov simulation environment

Frode B. Nilsen

Research Report 258

ISBN 82-7368-183-1
ISSN 0806-3036

April 1998

GMSim: A generalized semi-Markov simulation environment

Frode B. Nilsen

Research Report 258
ISBN 82-7368-183-1, ISSN 0806-3036
Department of Informatics, University of Oslo.

April 1998

Abstract

A discrete-event simulation environment, called GMSim, based on the generalized semi-Markov process (GSMP) framework is described. The tool is completely generic and extendible by Tcl script programming. Application specific components are developed in an objected-oriented setting by C++ programming in combination with M4 macro processing. Components are conveniently integrated by run-time linking.

The strong links to the underlying mathematical GSMP description is favorable in two respects. First, qualitative results from a body of theory is readily available. Next, the structured view leads to an efficient implementation.

Contents

Preface

This report was written as a part of my work with a doctoral dissertation at the Department of Informatics, University of Oslo. The work is supported by grant no. 100722/410 from the Norwegian Research Council.

The report documents version 1.0 of GMSim. The full source distribution for the simulation tool is available at

`<http://www.ifi.uio.no/˜froden/gmsim>`

Otherwise, the contact address of the author is

Frode B. Nilsen
`<froden@ifi.uio.no>`
`<http://www.ifi.uio.no/˜froden>`
Department of Informatics, University of Oslo
P.O. Box 1080, Blindern
N-0316 Oslo, Norway

Chapter 1

Introduction

The generalized semi-Markov process (GSMP) description is a versatile stochastic formulation of the dynamics found in discrete-event systems [10]. It is at the same time both a precise mathematical setting for analysis and a discrete-event simulation algorithm. In some sense the GSMP view unifies analytical methods with simulation, hence we use the term *simulytic* to characterize the approach.

Closed-form solutions can *not* be derived from the GSMP formulation. The framework is targeted at simulation but benefits greatly from the inherent mathematical structure. Quantitative results must be obtained by simulation but a flavour of qualitative theory *can* be established [10]. The latter is the reason for applying the GSMP framework in the first place. The key point is that theoretically sound and computationally efficient methods for carrying out the simulation are readily available. This includes experimental design, sampling strategies and output analysis [9, 12].

A GSMP model can really be implemented in any programming or simulation language. However, we argue that it is preferable to keep the implementation of a simulation model in in close resemblance with the underlying view. This document describes the development of an appropriate simulation environment called GMSim.

The starting point is the basic GSMP formalism. Figure 1.1. suggests that the idea is to convert this into an object-oriented implementation. The intermediate step is a compositional GSMP view. The contribution of this work is related to the latter two steps and the novelty is the compositional formulation. To our knowledge there is no available implementation of a simulation tool that directly reflects the GSMP view.

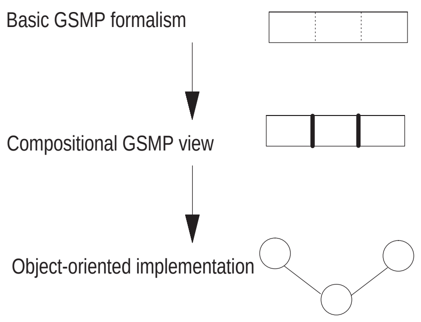


Figure 1.1: *The idea behind the GMSim development*

The preciseness of the underlying description is also beneficial when it comes to implementation. Due to the systematic and well-structured view we conjecture that an direct implementation of the GSMP formulation leads to a consistent and efficient tool. This is the motivation for the GMSim development.

The objectives of GMSim are execution speed and flexibility. Speed is gained by building binary code for the basic parts of a model. This is based on the C++ programming language [38] in combination with M4 macro pre-processing [34]. The concept of various hook functions is central to the discussion. Together, the set of recognized hooks forms the programming interface to the GMSim core.

The GMSim tool itself is completely generic. Application specific components are conveniently incorporated by run-time linking. Flexibility is further enhanced by using the Tcl/Tk script environment [27] for simulation setup, configuration, control and reporting. The tool can be extended in arbitrary ways by script programming. Scheduling in GMSim is based on two-level strategy where a pairing-heap [8] is used for the global queue.

In this document it is generally assumed that the reader has knowledge of the C++ programming language, the Tcl/Tk environment and the M4 macro processor. The GMSim tool is currently available [1] for the GNU gcc compiler and the Linux and Solaris operating systems. It has successfully been used for studying the performance of wormhole-switched communication systems [26].

Note that we only consider the core of the GMSim tool. Any application specific extensions are beyond the scope of this document.

1.1 Organization

The emphasis of this document is the *implementation* of GMSim more than its theoretical foundation. Hence, a discussion of the basic GSMP formalism is left to appendix B. Likewise, the compositional GSMP view is formally discussed in appendix C. For the casual reader it is sufficient to know that the object-oriented implementation is based on how a GSMP model can be represented as an ensemble of interacting sub-models. For readers unfamiliar with the Tcl/Tk environment, a brief introduction is given in appendix A.

The rest of this document is organized as follows. Chapter 2 provides some background information on modeling of systems with discrete-event dynamics. The GSMP is one possible approach which builds on the successful heritage from Markov chain analysis. This is followed by an overview of the GMSim environment and its features in chapter 3. Central to the discussion is how the Tcl/Tk scripting environment work together with code written in C++. Chapter 4 elaborates on certain basic concepts like packages, classes, simulation mode, verboseness and configuration. The C++ programming interface is described in chapter 5. This is based on the concept of hook functions. In chapter 6 various issues related to model development are discussed. This includes debugging, deadlock detection and facilities for extending the tool itself. Finally, the important topic of scheduling is discussed in 7. The report is summarized in chapter 8.

[1]See `<http://www.ifi.uio.no/~froden/gmsim>`

Chapter 2

Background

Modern technology has increasingly created man-made systems with discrete-event dynamics. The class of queuing networks serves as an example. Such systems, termed discrete-event dynamic systems (DEDS) are in contrast to continuous variable dynamics systems (CVDS) often found in nature [14,15]. The distinction between DEDS and CVDS is illustrated by figures 2.1 and 2.2.

Differential equations are *the* most important tool for modeling and analysis of CVDS. The success of the approach is due to a functional view with well-defined relations between the constituent variables. A DEDS is different as its workings are naturally defined in terms of a set of operational rules. This corresponds to an algorithmic view. The definition of state is also different. DEDS use a pure physical state description whereas CVDS entail a proper mathematical state. The latter includes the former, but not vice versa. Another feature is that DEDS are often so complex as too appear as random even though the constituent parts all behave deterministically.

There seems to be no convenient way to capture the behavior of DEDS mathematically with the same degree of efficiency as the case of CVDS with differential equations [14, 15]. The available theory is in general less developed than for CVDS. The most important achievement is the theory of Markov chains [4]. Associated scientific disciplines are operations research [24] and queuing theory [21].

Much of the theory in both domains is qualitative in nature. A major computational

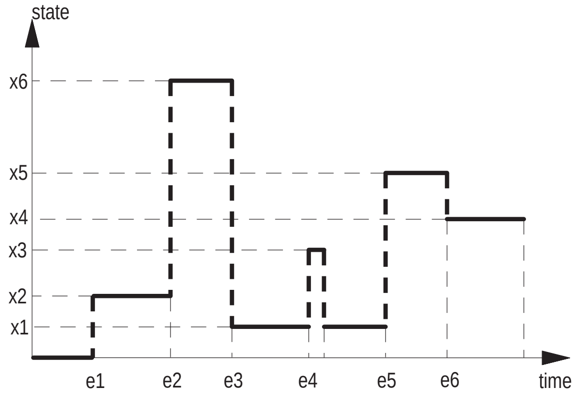


Figure 2.1: *Illustration of a discrete-event dynamic system (DEDS).*

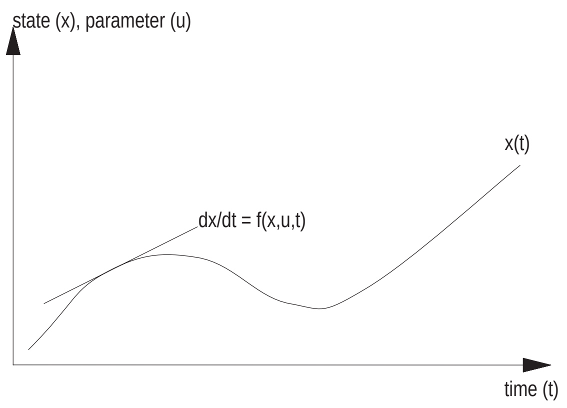


Figure 2.2: *Illustration of a continuous variable dynamic system (CVDS).*

difference is that CVDS are more readily available for numerical analysis. DEDS typically suffer from a combinatorial explosion of the underlying state space [12, 15]. For a completely general approach to modeling of DEDS we are left with the alternative of simulations [1, 2, 6, 22, 28].

The generalized semi-Markov process (GSMP) formalism [9–13, 35, 36] is one approach to modeling and analysis of DEDS. It is an attempt to build on the successful heritage from Markov chains. Hence, it provides a precise mathematical framework for analysis. At the same time it is a succinct description of a discrete-event simulation algorithm. To emphasize the duality we use the term *simulytic* to characterize the approach. Quantitative results can be found from simulations. The utility of the inherent mathematical structure is to simplify output analysis. A flavor of qualitative theory and numerical algorithms can be obtained as a result of viewing discrete-event systems as GSMP. In particular, methods for output analysis and statistical inference are readily available.

The basic idea of the GSMP formalism is to add supplementary variables [5] to the state description so as to maintain the Markov property. Then the process becomes memoryless is the sense that the future evolution depends only on the current state. As always, this simplifies analysis. It should also be noted that the GSMP framework is a behavioral formulation rather than a logic or algebraic description. Therefore, it is well-suited for performance studies but has less relevance when it comes to formal verification and validation [2, 29] of system designs.

Chapter 3

Overview

GMSim is structured as shown in figure 3.1 and consists of a number of packages to be loaded into a Tcl interpreter. The interpreter must exist in the realm of a running program. Each package comprises binary code which is dynamically linked[1] with the executing program at load-time. One of the standard Tcl/Tk shells `tclsh` or `wish` are normally used to host the interpreter. Readers unfamiliar with Tcl/Tk should consult appendix A before reading on.

A package named `core` must be loaded to set up the basic simulation environment. This results in an enriched set of commands and global variables that enable simulation. Names added to the interpreter have the format `sim_xxx`. Appendix F and appendix G document the commands and variables pertinent to the user.

The new script commands are used to build and manage simulation models. A model comprises items instantiated from prototype classes of various kinds.

- alive classes
- dead classes
- parameter classes
- statistics classes.

[1]Dynamical linking is a feature of the Tcl interpreter.

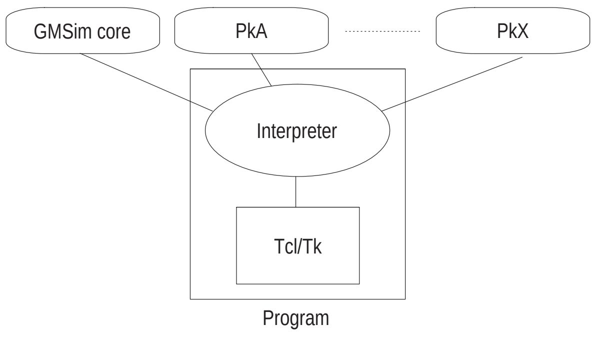


Figure 3.1: *GMSim is structured as a number of packages to be loaded into a running Tcl interpreter. Loading the `core` package sets up the basic simulation environment.*

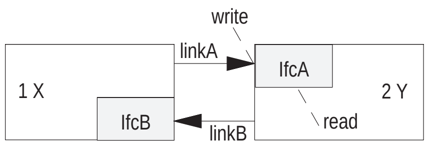


Figure 3.2: *Inter-object communication takes place in terms of links and interfaces. Links are typed and can only connect to conforming objects.*

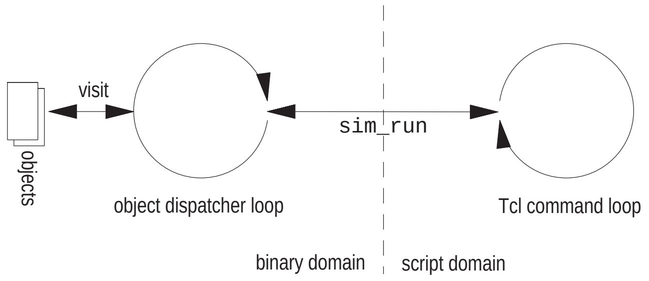


Figure 3.3: *GMSim alternates between the script domain and the binary domain during operation. The command loop parses script commands. Objects are visited from the object dispatcher loop.*

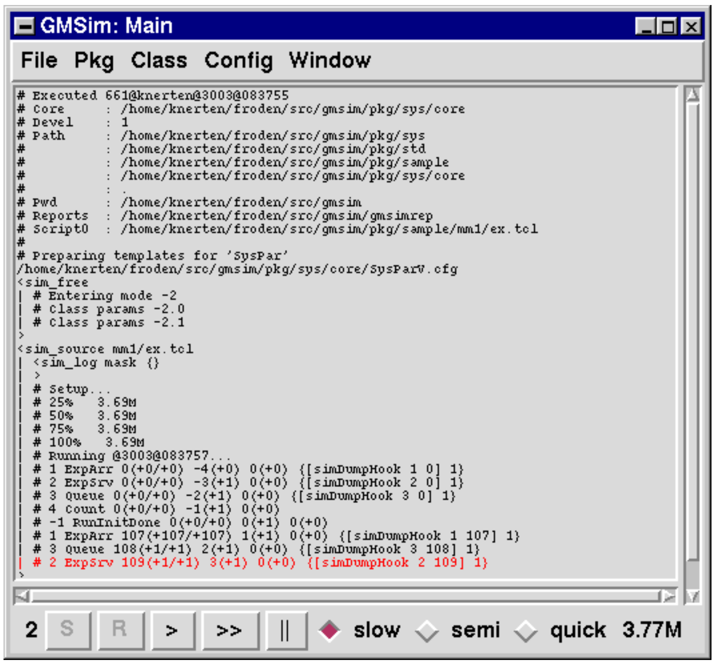
An item is either an object or a statistics. Objects are instantiated from the first two types of classes. Statistics are created from statistics classes. In section 4.4 we return to the properties of the different kinds of classes. Before instantiation can start, space must be allocated by the `sim_alloc` command. The `sim_new` command is responsible for instantiation.

An alive object corresponds to a component in the compositional GSMP formulation discussed in appendix C. Connections between objects are formed by links and interfaces. This is illustrated in figure 3.2. Two objects are instantiated from classes `X` and `Y` and connected by a pair of links. Interaction takes place in term the connecting links. In this case each object posses one interface (shaded) and the links are typed according to the interfaces they conform to. Hence, only compatible objects can be linked. This issue is further discussed in section 4.7. Links are created by the `sim_link` command.

The GMSim core provides *no* classes. This is left to additional packages. Each package is expected to implement pertinent classes according to the compositional GSMP view. The recognized classes depends on which packages are loaded. New packages are loaded in response to the `sim_pkg` command. Hence, the modeling capabilities of GMSim can be extended in arbitrary ways without recompiling the core. In section 4.3 we return to the issue of package development.

The operation of GMSim is illustrated in figure 3.3. We distinguish between the binary domain and the script domain. The command loop is responsible for parsing script commands. If the `sim_run` command is invoked, simulation proceeds by transferring control to the binary domain and the object dispatcher loop. In turn, the instantiated (alive) objects gain control according to their ordering in a scheduler queue. Scheduling is discussed in more detail in section 7.1.

For each alive object a piece of behavioral code is executed before control is relinquished. This is called a visit to the object. The number of objects visited before



GMSim: Main
File Pkg Class Config Window
2 S R > >> || slow semi quick 3.77M

```
# Executed 661@knerten@3003@083755
# Core      : /home/knerten/froden/src/gmsim/pkg/sys/core
# Devel     : 1
# Path      : /home/knerten/froden/src/gmsim/pkg/sys
#           : /home/knerten/froden/src/gmsim/pkg/std
#           : /home/knerten/froden/src/gmsim/pkg/sample
#           : /home/knerten/froden/src/gmsim/pkg/sys/core
#           : .
# Pwd       : /home/knerten/froden/src/gmsim
# Reports   : /home/knerten/froden/src/gmsim/gmsimrep
# Script0   : /home/knerten/froden/src/gmsim/pkg/sample/mm1/ex.tcl
#
# Preparing templates for 'SysPar'
/home/knerten/froden/src/gmsim/pkg/sys/core/SysParV.cfg
<sim_free
| # Entering mode -2
| # Class params -2.0
| # Class params -2.1
>
<sim_source mm1/ex.tcl
| <sim_log mask {}
| >
| # Setup...
| # 25%   3.69M
| # 50%   3.69M
| # 75%   3.69M
| # 100%   3.69M
| # Running @3003@083757...
| # 1 ExpArr 0(+0/+0) -4(+0) 0(+0) {[simDumpHook 1 0] 1}
| # 2 ExpSrv 0(+0/+0) -3(+1) 0(+0) {[simDumpHook 2 0] 1}
| # 3 Queue 0(+0/+0) -2(+1) 0(+0) {[simDumpHook 3 0] 1}
| # 4 Count 0(+0/+0) -1(+1) 0(+0)
| # -1 RunInitDone 0(+0/+0) 0(+1) 0(+0)
| # 1 ExpArr 107(+107/+107) 1(+1) 0(+0) {[simDumpHook 1 107] 1}
| # 3 Queue 108(+1/+1) 2(+1) 0(+0) {[simDumpHook 3 108] 1}
| # 2 ExpSrv 109(+1/+1) 3(+1) 0(+0) {[simDumpHook 2 109] 1}
>
```

Figure 3.4: *The main window of GMSim showing the log for a specific example. The current response line is highlighted.*

control returns to the script domain depends on the designated simulation speed. This is further discussed in section 4.2. The most recently visited object is referred to as the current object. Information about the current status is provided by the `sim_curr` command.

GMSim provides a graphical user interface if the hosting program supports Tk. Figure 3.4 shows the status of the main window if the script file `mm1.tcl` listed in appendix E.4 is used. Except from menus and buttons, the main window shows a log of responses as script commands are evaluated. The last line marked with a hash

```
2 ExpSrv 109(+1/+1) 3(+1) 0(+0) {[simDumpHook 2 109] 1}
```

is the response after three simulation steps. Whenever control returns to the script domain a line like this is printed. The first field is an identification of the current object. Objects are given numerical identifiers as the are instantiated. The second field shows the object class. The third field is the current simulation time. The incremental value in parenthesis is the elapsed simulation time since the last invocation of the run command. The fourth field is the accumulated number of visits and the fifth field is associated with deadlocking as discussed in section 6.3. The sixth field has to do with script hooks as explained in section 6.1

The log is an example of a report in GMSim. Reports can be opened to write information to ordinary text files during simulation. Associated windows can optionally be displayed on the screen.

The log is an example of a report in GMSim. A Report, which is associated with an underlying text file and optionally a window, can be managed entirely from the script domain by the `sim_rep` command. A particular kind of report, called a dump report, can be requested for any alive object. This makes the object verbose about its inner workings.

GMSim includes a system where configuration parameters can be set and read from the script domain. This is accopplished by the `sim_par` command and is subject to discussion in section 4.6. A generic feature for recording measurements in the binary domain is also provided . This takes place in terms of assigned statistics. Statistics are in turn accessible from the script domain. We return to this issue in section 4.8. The `sim_stats` command is used to manage statistics.

General system information is provided by the `sim_info` command. This includes the names of recognized and loaded packages. The same command gives specific information about packages, classes or objects. Package information includes dependencies and implemented classes. Class information includes linking features. Object information includes actual links and scheduled status.

Finally, note that on-line help is available by the `sim_h` command.

Chapter 4

Basic concepts

4.1 Startup

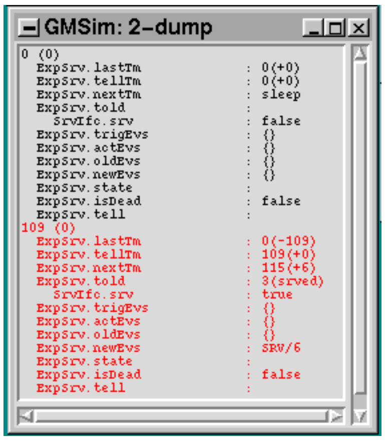
The detailed procedure for starting GMSim is left to a README file in the source distribution. Without going into details, it includes a convenient shell script wrapper. In this section we discuss general concepts related to startup. Keep in mind that the tool is launched by loading the `core` package in a running Tcl interpreter.

More specifically, there are two variants of the package referred to as the development and production versions. The former is equipped with debugging and verboseness features. The production version is targeted at execution speed and minimizes such features. Each additional packages also come in two version. The appropriate type is loaded according to the version of the `core`.

GMSim is started with the graphical user interface if the hosting program supports Tk. Otherwise a pure Tcl version is started. In any case, a command-line interface is available if the executing program provides interactive terminal capabilities. The graphical interface always include the main window. Other windows are initially displayed depending on the setting of the `sim_wdisp` variable. E.g. if a window showing data about the current object is of interest, `sim_wdisp(curr)` should be set prior to loading the `core`.

An important step in the startup sequence is recognition of installed packages. This is based on the contents of the `sim_path` variable. For each directory `yyy` in the path list, the immediate sub-directories `yyy/xxx` are searched for packages. If a sub-directory `xxx` contains a file `simpkgIndex.tcl`, it is assumed to be the directory location of a package named `xxx`. Note that packages are *not* loaded as they are recognized. The user can control which packages are recognized by setting the path variable prior to startup.

The final step of startup is to source the script files named by the `sim_script` variable. In this way application specific scripts can be evaluated immediately. This is required in batch mode. Scripts can also be evaluated on demand by the `sim_source` command. Note that the `sim_path` variable is also used when searching for the location of script files.



```
0 (0)
  ExpSrv.lastTm                    : 0(+0)
  ExpSrv.tellTm                    : 0(+0)
  ExpSrv.nextTm                    : sleep
  ExpSrv.told                      :
    SrvIfc.srv                     : false
  ExpSrv.trigEvs                   : {}
  ExpSrv.actEvs                    : {}
  ExpSrv.oldEvs                    : {}
  ExpSrv.newEvs                    : {}
  ExpSrv.state                     :
  ExpSrv.isDead                    : false
  ExpSrv.tell                      :
109 (0)
  ExpSrv.lastTm                    : 0(-109)
  ExpSrv.tellTm                    : 109(+0)
  ExpSrv.nextTm                    : 115(+6)
  ExpSrv.told                      : 3(srved)
    SrvIfc.srv                     : true
  ExpSrv.trigEvs                   : {}
  ExpSrv.actEvs                    : {}
  ExpSrv.oldEvs                    : {}
  ExpSrv.newEvs                    : SRV/6
  ExpSrv.state                     :
  ExpSrv.isDead                    : false
  ExpSrv.tell                      :
```

Figure 4.1: *Example of a dump report window in GMSim. The current dump is highlighted.*

4.2 Verboseness and speed

GMSim is verbose at every return to the script domain since a response line is always printed. Verboseness can be further enhanced in two ways

- Script commands can be logged as they are evaluated.
- Objects can be made verbose as they are visited.

Logging of script commands is controlled by supplying an appropriate mask to the `sim_log` command. Each loggable command is associated with a specific level designation. Logging takes place if this level number appears in the mask when the command is evaluated. This feature is very helpful to trace bugs in scripts.

The `sim_dump` command can be used to reveal the inner workings of an object and simplify debugging. A dump of the internal state for any object is available at request. This can be used to make an object verbose in the sense that a dump is automatically prepared at every visit. The information is in turn written to an associated dump report when control returns to the script domain. An example of a dump report is shown in figure 4.1 for an object instantiated from the class `ExpSrv` discussed in appendix E. In section 6.2 we discuss how to generate the information contained in such a dump.

Control should return to the script domain for each verbose object. Hence, verboseness works together with simulation speed. The speed is controlled by the `sim_speed` command and takes one of the values `slow`, `semi` or `quick`. Slow mode corresponds to single-step behavior where control returns to the command loop at every pass through the object dispatcher loop. In semi mode control returns only for objects set to be verbose. In quick mode verboseness is bypassed all together and simulation proceeds as fast as possible without returning to the script domain.

Note that exceptional system conditions will also always force a return to the script domain. This is controlled by setting system parameters as documented in appendix D. In addition, the evaluation of object script hooks as described in section 6.1 may lead to a forced return, regardless of the designated speed.

In any case, when control eventually returns to the command loop, the subsequent behavior depends on if > or >> was given as an argument to `sim_run`. The former

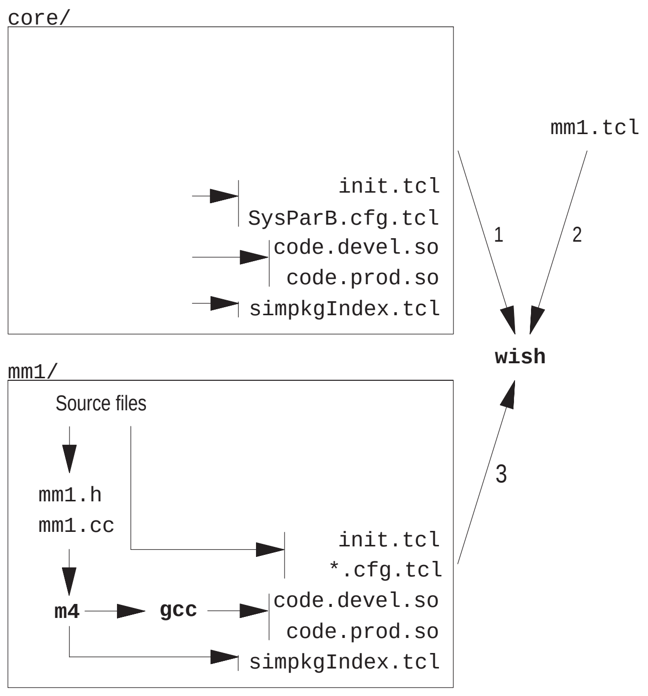


Figure 4.2: *GMSim is launched (1) by loading the* `core` *package into a running interpreter. Additional packages (3) are loaded in response to application (2) needs. The* `mm1` *package illustrates package building and which files are involved.*

corresponds to breaked operation. Then a new run command must be invoked explicitly in order to continue. Otherwise the simulation proceeds immediately as the latest command is re-issued automatically. This is called continuous run.

4.3 Packages

The user is responsible for loading packages according to which classes are needed in a particular simulation model. Loading is accomplished by the `sim_pkg` command. It starts with resolving all dependencies so that any subsequently required packages are loaded first.

The process of package development and use is outlined by an example in figure 4.2. A package `mm1` is considered and it is assumed that all associated files are installed in a directory `mm1/` relative to the search path. File listings for this example are found in appendix E.

The package is built from a single source file `mm1.cc` which in turn is compiled into a pair of binary files `code.devel.so` and `code.prod.so`. The binary files correspond to development and production versions of the package, respectively. An accompanying index file `simpkgIndex.tcl` contains information about implemented classes and any package dependencies. The `*.cfg.tcl` files have to do with configuration as explained in section 4.6.

The source file is pre-processed by the M4 macro processor before actual compilation. Using a set of macros which is related to the GSMP formalism is mandatory.

As these macros are expanded code is automatically generated to takes care of package initialization and all interfacing to the Tcl environment. The last step of the initialization code is to read a script file `init.tcl`. This provides for packet initialization from the script domain. E.g. the `core` package has a long script sequence setting up the basic GMSim environment.

The GMSim source distribution includes several makefiles which assist in building packages.

4.4 Classes

As already suggested there are different kinds of prototype classes in GMSim. Most objects are instantiated from alive classes. An alive class corresponds to part of a GSMP description and comprises:

1. A piece of behavioral code which is written according to the GSMP view. This is based on a state description and a set of events as discussed in section 5.5.
2. Declarations of links which can be used to form outgoing connections. We return to this issue first in section 4.7 and then in sections 5.4 and 5.5.
3. One or more interfaces to accept in-going connections. This is described in section 5.3.
4. Recognized configuration parameters enhancing the versatility of the class. The corresponding programming interface is discussed in section 5.4.
5. Declaration of measures to record observations. We return to this topic first in section 4.8 and then in section 5.5.

A simulation model can also comprise dead objects. Since a dead class lacks characteristics 1 and 5, such objects do not really participate in the simulation. Their role is simply to act as link centers.

Statistics are instantiated from statistics classes. They only possess property 4 but are related to alive classes by way of measures. We return to this issue in section 4.8.

A parameter class is also restricted to property 4 but is otherwise self-contained. This is how global variables are handled in GMSim. Parameter classes cannot be used for instantiation in the ordinary way by `sim_new`. They are automatically instantiated once and given an identifier corresponding to the class name in lower case. E.g. the `core` package defines the `SysPar` class with a corresponding instance named `syspar`. The system parameters are documented in appendix D and is related to generic simulation control. The normal use of parameter classes is to define one such class for each package. In this way global parameters associated with the package can be set conveniently.

Note finally that the class concept in GMSim build directly on C++ classes. Hence, the full power of inheritance and polymorphism characteristic for object-oriented programming is available.

4.5 Modes

The GMSim tool is in different modes during operation. The mode designation is numeric in the range $[-2, 2]$ where -2 is the default after launching the tool. The mode

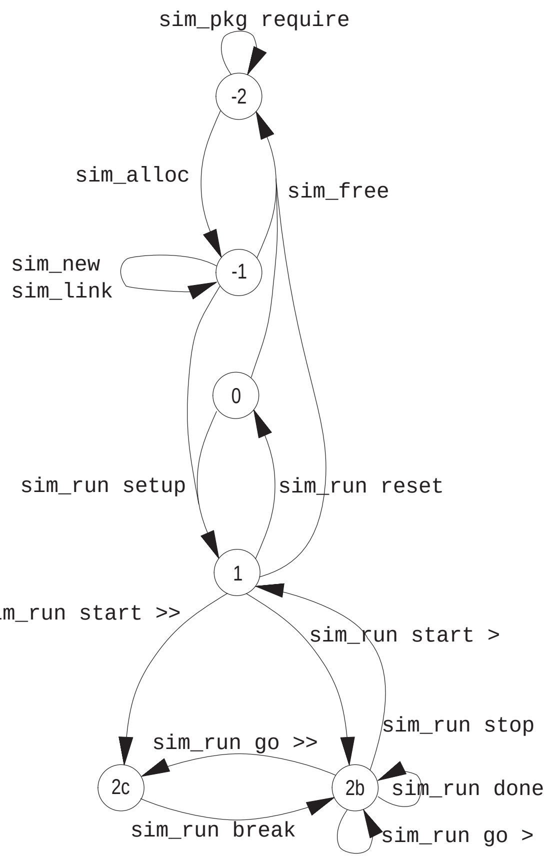


Figure 4.3: *GMSim operates according to a mode designation. The diagram shows the mode specific script commands and how they lead to transitions.*

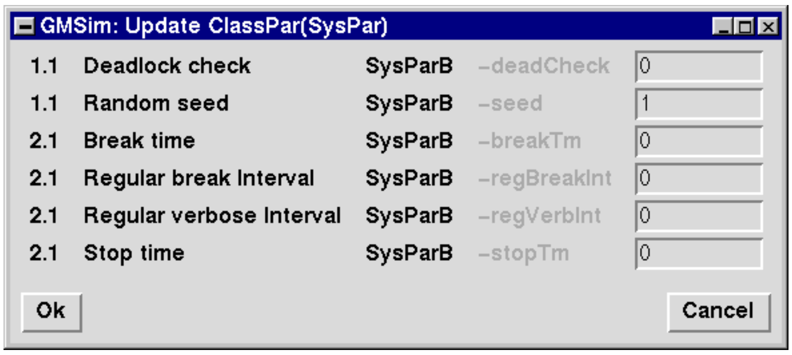
is important in two respects. First, certain script commands are restricted to specific modes. Second, the significance of configuration parameters are mode dependent as discussed in the next section.

Figure 4.3 is a mode transition diagram displaying the mode specific commands. Every command not included in this figure is general and can be used in any mode. Mode -1 is where objects can be instantiated and linked. Topology specific configuration parameters are also set in this mode. Other configuration parameters get their initial value in mode 1. Mode 2 is the run mode and it is split into two sub-modes corresponding to continuous or breaked run. Note that all required packages must be loaded in mode -2 before the topology can be setup in mode -1.

4.6 Configuration

The final step of package loading is to prepare a configuration template for each class introduced by the actual package. The templates are build from configuration files having names `xxx.cfg.tcl` where `xxx` corresponds to a class name. These files have one line of the following format for each recognized parameter.

```
x.y "text" -flag defval {opt1 opt2}
```


GMSim: Update ClassPar(SysPar)

1.1	Deadlock check	SysParB	-deadCheck	0
1.1	Random seed	SysParB	-seed	1
2.1	Break time	SysParB	-breakTm	0
2.1	Regular break Interval	SysParB	-regBreakInt	0
2.1	Regular verbose Interval	SysParB	-regVerbInt	0
2.1	Stop time	SysParB	-stopTm	0

Figure 4.4: *The configuration dialog box for the `SysPar` parameter class.*

There are five fields separated by white spaces. The third field is a unique flag used for identification. The second field is an explanatory text used in configuration menus. The fourth field holds the default value for the parameter. The fifth field is optional and is a list of legal values. If this field is omitted any value is accepted.

The leading dot-formated number $x.y$ is used to determine the when the parameter is significant. The number $y \in \{0, 1\}$ tells whether it is a class or object parameter. A class parameter is common to all instances whereas object parameters can be set individually. The number $x \in \{-1, 1, 2\}$ gives the highest mode m_h in which the parameter is setable. E.g. a topology specific parameter should have a value $m_h = -1$. In addition, the second column of the following table is used to determine the lowest mode m_l in which the parameter can be set.

x	Low(m_l)	High(m_h)
-1	-1	-1
1	1	1
2	1	2

Hence, $[m_l, m_h]$ gives the range of modes in which a specific parameter is setable.

At entry to a new mode m all parameters with a corresponding low designation m_l get their initial value from the default entry held by the template. For $m < m_l$ the parameter is without significance and the user is free to modify the default template value. For $m \geq m_l$ the template value is considered read-only. For $m > m_h$ the parameter can not be set but is still of significance.

The default parameter value in template entries can be set or read by the `sim_par` command using the class name as the second argument. Object parameters can be set and read by the same command with the object identifier as the second argument. Note that class parameters are only setable by way of the default template value.

If Tk is available parameters can also be set and browsed in terms of dialog boxes. This happens in response to the commands `sim_xpar` and `sim_xvpar`. Figure 4.4 shows the dialog box that will appear if the template entry for the `SysPar` parameter class is inspected.

Configuration parameters for a class are implemented in terms of the general programming interface discussed in section 5.4. In addition, some parameters concerning links and measures are recognized automatically. This is discussed in more detail by section 4.7 and section 4.8.

4.7 Links and interfaces

Connections between model objects are formed by links and interfaces as illustrated in figure 3.2. A link acts as a plug which can be inserted into a socket in terms of an interface. Together they form a one-way communication channel. An interface is considered to be written from outside and read from inside. Links are typed according to the interface(s) they conform to so that only compatible objects can be linked.

We will see in section 5.4 that a link declaration is complete except that the number of emerging links of a particular name is left unspecified. Rather, this is configurable by a parameter. If `lnknam` is the name of a link, then `-#lnknam` is a parameter specifying the number of links. Note that these parameters are topology specific and common to all instances of a class. Hence, the configuration template entry has a significance field of -1.0.

4.8 Measures and statistics

In section 5.5 it will become apparent that a measure is a function which can be called whenever an object is visited. A measure is concerned about how to make an observation. The observation is in turn recorded in terms of a statistics.

For a measure to be in effect for a particular object, one or more statistics must be assigned. This is accomplished by setting a configuration parameter which is automatically recognized. I.e. if `mesnam` is the name of a measure, then `-mesnam` is a parameter which takes a list of statistic identifiers. Any non-zero value indicates that observations should be collected and recorded in the designated statistics. A zero value breaks any association for the measure.

The preferred way to establish an association between a measure and a statistics is to use the `sim_stats` command. This command also performs other management functions related to statistics.

Statistics can be assigned individually for objects during run. Accordingly, the configuration template entry for a measure parameter has a significance field of 2.1. Note also that the same statistics can be assigned for different objects, if required.

Chapter 5

Programming

This chapter describes how object-oriented programming according to the GSMP view takes place. Macro expansion and the concept of hook functions are central to the discussion.

5.1 Macro expansion

Code development for GMSim depends on using a number of M4 macros in a C++ environment. The idea is to extend the concept of a class declaration by using block constructs like

```
sim_xxx(...) sim_use(...) {
  sim_yyy(...);
  ...
sim_data:
  ...
sim_hooks:
  ...
sim_body:
  ...
};
```

Here `sim_xxx`, `sim_use`, `sim_yyy`, `sim_data`, `sim_hooks` and `sim_body` corresponds to M4 macro calls. As these macros are expanded code is automatically generated that takes care of all interactions with the GMSim core.

The following kinds of blocks are recognized with the associated opening macro call in parenthesis:

- Interface declaration (`sim_ifc`)
- Virtual alive class declaration (`sim_vaclass`)
- Real alive class declaration (`sim_raclass`)
- Virtual dead class declaration (`sim_vdclass`)
- Real dead class declaration (`sim_rdclass`)
- Virtual statistics class declaration (`sim_vsclass`)
- Real statistics class declaration (`sim_rsclass`)

- Virtual parameter class declaration (`sim_vpclass`)
- Real parameter class declaration (`sim_rpclass`)

The `sim_use` macro following a block-opening macro is always used to specify inheritance. An alive class can only inherit alive base classes. Likewise, an dead class can only be derived from dead base classes. The same applies for parameter classes and statistics classes. The distinction between virtual and real class declarations is important. The latter corresponds to a complete prototype which can be instantiated from the script domain. Virtual classes are abstract [38] and cannot be instantiated. They represents base classes which must be further inherited. On the other hand, a real class is final and cannot be inherited.

Anything between the delimiting `sim_data` and `sim_hooks` statements is expected to be ordinary C++ variable declarations pertinent to the class. The trailing part of the block, i.e. after the `sim_body` statement, is expected to be ordinary C++ member function declarations.

For each of the block constructs there is a set of C++ member functions, referred to as hooks. The hook functions are called from within the `core` and represents a convenient way to let the user implement a particular behavior for a class. Each hook has a default implementation which is used unless overridden by the user. Overridden hooks should be declared between the `sim_hooks` and the `sim_body` statements. Note that there will be several hooks of the same type in a multi-level class hierarchy. Similar hooks are invoked in sequence from top to bottom and left to right in the inheritance graph. Hence, *all* instances are called in response to a hook invocation.

The following subsections describes each of the block constructs in turn. For simplicity we restrict the discussion to real classes, noting that the virtual class constructs have similar features. The reader is urged to also take a look at the example in appendix E. For more detailed information the source distribution should be considered.

5.2 Standard hooks

In GMSim there is a number of standard hooks common to all of the block constructs.

```
  ...
sim_data:
  ...
sim_hooks:
  void objParSet (Sim_Opt &opt);
  void objParGet (EString &lst);
  bool objCheck (int mode, EString &msg);
  void objInit(int mode);
  void objClean(int mode);
  void objInfo(EString &lst);
  ...
sim_body:
  ...
```

The `objParSet` and `objParGet` hooks represent the programming interface for setting and reading configuration parameters. The utility class `Sim_Opt` is used for convenient parsing of options.

The `objParSet` hook is called for the first time right after instantiation of a model object or statistics. Subsequently, it is invoked when GMSim makes a transition into a higher-numbered simulation mode. Each time the configuration parameters with a matching significance designation get their default values. However, the `objCheck` hook is called before leaving the current mode. It provides an opportunity to check that the configuration is valid before completing the transition.

Otherwise, the `objParSet` hook is called in response to the `sim_par set` command. The `objParGet` hook is invoked from the `sim_par get` command.

The `objInit` hook is also called for every object and statistics at a transition into a higher-numbered simulation mode. It is responsible for initialization to known state according to the new mode. Since this happens after the corresponding invocation of `objParSet`, initialization may depend on configuration parameters. The hook is initially called at instantiation.

In contrast, the `objClean` hook is invoked for every object and statistics as GMSim makes a transition into a lower-numbered simulation mode. As the name implies, it it responsible for cleaning up according to the current mode.

The `objInfo` hook can be used to provide specific information about an object or statistics. This hook is called whenever the `sim_info` command is invoked. For interfaces, alive and statistics classes additional hooks are defined as explained in section 5.3, section 5.5 and section 5.6

5.3 Interface

An interface is the abstraction used for inter-object communication as discussed in section 4.7. Interfaces are declared according to

```
sim_ifc(name) sim_use(...) {
sim_data:
  int x;
  ...
sim_hooks:
  <std. hooks>
  void ifcClear (void);
sim_body:
  name &operator= (const name &ifc);
  void xWrite(int wx) {
    x = wx;
  };
  ...
};
```

and the standard hooks are recognized. We will see in section 5.5 that interfaces become part of alive classes. A standard interface hook is invoked whenever the corresponding hook of the incorporating class is called.

For an interface declaration to be useful ordinary C++ variables should be included. The variable `x` provides an example. Variables are supposed to represent features of the interface and they are modified when the interface is written from an external object. Writing should always be accomplished in terms of a corresponding member function.

Internally, GMSim operates with dual interfaces. This protects from mutual overwriting when objects are scheduled at the same time. An assignment operator function should *always* be provided in order to make a temporary copy of any pertinent variables.

The `ifcClear` hook is invoked right after the copy operation. This should leave the interface in a clean state so that any newly written variables are recognized at the next visit. As will be apparent from the discussion in section 7.1, an object will always get the opportunity to read the copied variables before they are overwritten.

Note that interfaces can be inherited like classes. A derived interface will be polymorphic in the sense that it also conforms to its base interfaces, if any.

5.4 Dead class

A dead class is declared according to

```
sim_rdclass(name) sim_use(...) {
  sim_links(...);
sim_data:
  ...
sim_hooks:
  <std. hooks>
sim_body:
  ...
};
```

and inheritance is allowed. Emerging links are declared by the `sim_links` macro using an argument list of the following format

```
  name1 Type1, name2 Type2 OPT, ...
```

Each argument has two or three white-space separated fields and comprises a link definition. The leading field is an identifier. The second field defines the type of the link. For dead classes this should be the name of another dead class. Hence, dead objects can only connect to dead objects. Further, since a link is typed it can only connect to conforming objects.

The declaration of a link identifier corresponds an array of pointers to objects of the actual type. The size of this array is run-time configurable as explained in section 4.7. If a third field `OPT` is present in the link definition, a value of zero is accepted for the corresponding parameter. Otherwise at least one link of the given name must exists.

To refer to a particular link within the class scope the notation `name1[n]` should be used. The actual number of links is available by calling an automatically generated member function `name1LCnt()`. There are also two public member functions `name1LGet(...)` facilitating external access to the link-pointer array.

5.5 Alive class

Alive classes are declared by the construct

```
sim_vaclass(name) sim_use(...) {
  sim_events(...);
  sim_links(...);
  sim_measures(...);
sim_data:
  ...
sim_hooks:
  <std. hooks>
  Sim_UsrTime initOccur (int e);
  void nextState (void);
  Sim_UsrTime nextOccur (int ev);
sim_body:
  ...
};
```

and have the same features as dead classes. However, linking is more distinguished for alive classes. The point is that alive objects can connect to *both* dead and alive objects. In the latter case an interface specification must be used to specify the type of the link. Further, writing to a remote interface *must* take place in terms of using the `sim_tell` macro. E.g. if `lnk` is the name of a link and `membf` in the name of a member function of the associated interface, then

```
sim_tell(lnk[..], membf(...));
```

is the proper way to express inter-object communication.

The `sim_use` macro is used to indicate that an alive class conforms to a particular interface. If `MyIfc` is the name of an incorporated interface, then an identifier `theMyIfc` is introduced in the scope of the class. This is a reference which makes is possible to read the variables of the appropriate interface.

The careful reader may have noticed that the `sim_use` macro has an extended use in the context of alive classes since both inheritance and conforming interfaces can be specified. The type of an argument is automatically recognized.

The notion of state is significant for alive classes. A state description is provided by declaring ordinary C++ variables in the `sim_data` section. The state variables should be updated whenever the `nextState` hook is invoked. We return to this issue in a moment.

In accordance with the GSMP formulation a number of events is associated with an alive class. The events are declared as a list of symbolic names given to the `sim_events` macro. If `EV` is an event[1] defined for a class `MyClass`, then `MyClass_EV` is a corresponding identifier for the event. Events are represented as integer values internally, but the identifier should always be used when referring to an event.

The following event-sets are also introduced in the scope of the class

```
Sim_EvSet trigEvs;
Sim_EvSet actEvs;
Sim_EvSet oldEvs;
Sim_EvSet newEvs;
Sim_EvSet frozenEvs;
Sim_EvSet freezeEvs;
```

Here `Sim_EvSet` is a utility class which efficiently implements various set operations on integer valued members. The sets are used to express the behavior at a state transition according to the GSMP formulation. The significance of the first four sets should be obvious from the discussion in appendix B. The latter two sets are related to the rate at which event clocks run. In GMSim event clocks are restricted to run normally (rate 1.0) or being frozen (rate 0.0). The `frozenEvs` is the set of events which are currently frozen. The `freezeEvs` is used to mark which events to freeze after the state transition.

To indicate which events are initially scheduled for an object, the `newEvs` set should be set as a part of the `objInit` hook. For each of the initial events the `initOccur` hook is called in order to get the corresponding scheduling time.

At each subsequent state transition the `nextState` hook is called. This is *the* most important hook as it implements the behavioral model for a class. Specifically, `oldEvs`, `newEvs`, `freezeEvs` and the state variables should be modified according to the contents of `trigEvs`, `actEvs`, `frozenEvs` and the current value of the state variables.

By default, the assignments `oldEvs = actEvs` and `freezeEvs = frozenEvs` are made just before visiting the object. Immediately after visit the `nextOccur` hook is called for each event in `newEvs`. This determines when the actual events is to be scheduled. Scheduled events may also be canceled at this point.

The final feature of alive classes is that measures can be declared. A measure is concerned about how to make an observation of the present state of an object. The observation is in turn redcorded in terms of an assigned statistics as explained below. Whereas measures operate in the binary domain, statistics are accessible from the script domain. The assignment of statistics was discussed in section 4.8.

[1]By convention we use uppercase letters do denote events.

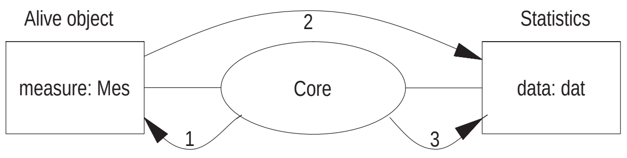


Figure 5.1: *Illustration of how the `core` maintains an association between a measure and a statistics.*

For a measure named `Mes`, its name must first be supplied as an argument to the `sim_measures` macro of the alive class construct. Next, a measure hook

```
bool Mes (double &dat) {
  ...
};
```

must be defined in terms of a member function of the same name. The measure is activated for an object by assigning a statistics as depicted in figure 5.1 The measure is considered by calling (1) the hook function right after each visit to the object. The `dat` parameter is a reference to a data area of the associated statistics. If the hook decides to take an observation, the referenced data area should be updated (2) and a true value returned. The latter informs the core that the statistics need to be invoked (3) in response to the new observation. Then the appropriate actions can be taken according to the nature of the statistics.

Note that a measure hook will only be called if at least one statistics is actually assigned . If more than one statistics are assigned, the hook will be called once for each statistics.

5.6 Statistics class

A statistics class is declared by the following construct.

```
sim_rsclass(name) sim_use(...) {
sim_data:
  ...
sim_hooks:
  <std. hooks>
  void reset(void);
sim_body:
  ...
};
```

Normally, the `sim_data` section declares variables according to the kind of statistics being implemented. The function of the `reset` hook is simply to reset these variables to an initial state.

It is important to note that every statistics class is actually inherited from a common base class

```
class Sim_StatsBase {
protected:
  double sdat;
public:
  virtual bool handleDat (void) = 0;
  virtual EString report(void) = 0;
};
```

When a statistics is assigned to a measure as explained in section 5.5, it is a reference to the variable `sdata` that is passed to the measure hook. If the measure hook returns true, it is assumed that a new value is written. Then the `core` invokes the virtual (abstract) handleDat function so that the modified value can be handled in any particular way by the actual statistics class. The usual action is to update some variable in response to the new data.

The handleDat function is also responsible for signaling whenever a new sample is formed by the actual statistics. This is accomplished in terms of returning a true value. If there is a one-to-one correspondence between measurement data and samples, handleDat should always return true. Otherwise, this function should only return true only if the newly written data area means that a new sample is formed. What is meant by a sample depends on the nature of the actual statistics, of course.

Note that the handleDat function is *not* classified as a hook. It does not make sense to invoke more that one instance of this function in a class hierarchy. The same applies for the virtual report function. The latter function is expected to report the content of the actual statistics in textual form.

5.7 Parameter class

The concept of parameter classes was introduced in section 4.4. To declare such a class the following construct is used

```
sim_rpclass(name) sim_use(...) {
sim_data:
  ...
sim_hooks:
  <std. hooks>
sim_body:
  ...
};
```

Keep in mind that this is how to handle global configuration parameters and that these classes are instantiated only once. If a parameter class ParClass is declared, then a corresponding global identifier

```
extern ParClass *Sim_ParClass;
```

is made available. This is a reference to the instantiated class. In this way global configuration parameter can be accessed from the binary domain.

Chapter 6

Development issues

6.1 Script hooks

The package system represents a way to extend the modeling capabilities of GMSim. In this section we discuss how the behavior of the tool itself can be extended by script programming. The idea is to permit the user to specify Tcl procedures that will be evaluated by GMSim at specific points during operation. We refer to such procedures as script hooks. This is completely different from the C++ hook functions discussed in chapter 5.

Figure 6.1 shows an elaborated view of how GMSim operates It suggests that there are three kinds of script hooks available. System hooks are invoked at every return to the Tcl command loop. Object hooks and statistics hooks are referred to as a callback hooks for obvious reasons. Note that object hooks only make sense for alive objects. Hooks are added and deleted by the `sim_hook` command, and more than one procedure can be specified for any kind of hook.

Object hooks are invoked from the `core` at *every* visit to an alive object. In contrast, statistics hooks are invoked only if certain conditions are met. This is related to the discussion in section 5.6. First, a statistics must be affected by the state transition in the sense that the `sdata` member is actually updated by the associated measure function. Next, the `handleDat` function must return true indicating that a new sample is now ready. Finally, the current sample number must match a next sample designation specified by the user. We return to the latter issue in a moment.

An object script hook is assumed to take at least two arguments

```
proc objHook {id time args} {
  ...
  return <msg>
}
```

The first argument will be the numeric identifier of the object from which the callback takes place. The second argument corresponds to the current simulation time. The return value from an object hook is also of importance. If a non-empty string is returned, control returns to the command loop immediately after the visit is completed. The returned value is in turn indicated in the status line together with the hook name. The trailing field of the status lines is used for this purpose. E.g. the example taken from figure 3.4

```
... {{simDumpHook 1}}
```

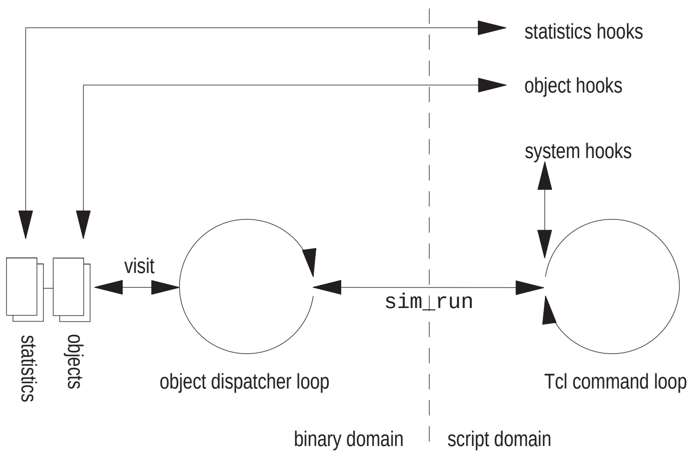


Figure 6.1: *Elaborated view of the operation of GMSim. The tool is extendible by script programming in terms of systems hooks, object hooks and statistics hooks.*

tells that a hook with name `simDumpHook` was invoked and yielded 1 as its return value.

The key point is that a non-empty textual value returned from an object hook *always* forces a return to the script domain. The opposite case is if all hooks return an empty string. Then the object dispatched loop proceeds with the next visit, unless the speed designation is slow in which case control return to the script domain in any case.

The verboseness feature outlined in section 4.2 is implemented in terms of an object hook script. Unless the speed designation is quick, in which case verboseness is bypassed, a dump is prepared when this hook is invoked. The hook then returns a non-empty string so that control returns to the script domain. At this point the dump is ready and can be written to an associated report.

A statistics hook is expected to be defined as

```
proc statsHook {id num args} {
  ...
  return [list <nextSample> <msg>]
}
```

where the first parameter is the numeric identifier of the statistics from which the call-back takes place. The second argument is the current sample number. The return value from a statistics hook is expected to be a two-element list. The significance of the second element is similar to the return value for object hooks. If one of the invoked statistics hooks return a non-empty second element, a forced return to the script domain takes place.

The first element in the return list is what was called the next sample designation. This is used to specify the sample number at which the hook should be invoked next. E.g. the hook is invoked at each new sample if `[expr $num + 1]` is returned. If a value less than the current sample number is returned, the statistics hook will never gain control again. The careful reader may ask what will be the first visit to a statistics hook. The point is that a statistics hook is always invoked with `num` equal to zero whenever the statistics is reset. In this way the hook is itself responsible to tell what should be the first invokation.

Note finally that a system hook takes no arguments but information about the current object is always available in term of the `sim_curr` command.

6.2 Debugging

The `sim_dump` command and the verboseness feature discussed in section 4.2 is a helpful debugging aid. The standard information included in a dump is the time since last visit, the contents of the various event sets and also the scheduled time, if any. In addition, interfaces and alive classes are all equipped with a C++ hook

```
void verbose(ostream &os);
```

which can be used to extend the verboseness. It is a good habit to always supply such a hook since it is often of great help in tracking programming errors.

In the development version of the `core` some run-time overhead is incurred. This is basically due to tracking of inter-object communication. In turn, this information is available from the dumps being prepared. It includes which links are activated by an object and also any incomming interface data. In the production version of the `core`, such features are turned off.

The GMSim environment also allows for low-level debugging by embedding assertion statements

```
Sim_assert(...);
```

in the code. This works together with the Tcl environment so that any errors are reported in the script domain. For efficient reasons, checking of assertions is turned off if the production version.

6.3 Deadlock detection

The GMSim tool implements a generic interface for detection of deadlocks in simulation models. Loosely speeking a deadlock arise in a system if no progress is really made even if the simulation is scheduled to continue. Routing deadlock in packet-switched networks is the typical example of this.

Deadlock checking takes place in terms of a specific C++ hook function

```
bool isDead (void);
```

which exists for alive classes. The default implementation simply returns false. In deadlock sensitive models this hook should be overridden and return a status according to the current state. The idea is that this hook is called at every visit to a object in order to see if the status has changed. If going from false to true a global counter is incremented. The counter is decremented when the status changes from true to false. A global deadlock is said to occur if this counter reaches a certain limit. The limit is settable by the `-deadCheck` system parameter documented in appendix D.

Note that the value returned by the `isDead` hook is available in the dump prepared for an object.

6.4 Granularity

An important development issue is how much a simulation model should be decomposed in terms of objects. We use the term granularity to reflect this. A highly granular model has many interacting objects but the behavior of objects themselves are simple. If the granularity is low there are fewer objects but the actions performed by each object is more complex.

The granularity involves a tradeoff between modeling simplicity and execution efficiency. There are mainly two advantages of high granularity:

- The modeling task is simplified as a combinatorially large state space is decomposed into managable parts.
- It is reasonable to expect that the behavioral code executed at each visit to an (small) object has few conditional tests. This is efficient with respect to execution time.

The disadvantages of high granularity is mostly related to efficiency:

- The administration cost for scheduling queue grows with the number of objects.
- It is reasonable to anticipate heavy interactions between object. Since some overhead is associated with inter-object communication, the execution time grows.

In general it is impossible to say what is the appropriate level of decomposition for a simulation model. However, quite often the granularity follows from the natural structure of the system being modeled.

Chapter 7

Scheduling

7.1 Two-level strategy

Scheduling of events is an intrinsic part of the GSMP formulation. Since objects in GMSim are essentialy sub-models, each object is responsible for proper scheduling of its own events. This is referred to as local scheduling. As the number of events per object is expected to be small a linear-search strategy is used for this.

The objects are in turn arranged by a global priority queue [33] as illustrated in figure 7.1. Information about the global scheduler queue is available by the `sim_sched` command. This includes both the actual contents and also various statistics.

Each entry in the global queue comprises an object identifier n and a time designation t. If there are more entries with the same time value they appear in arbitrary order. The time t corresponds to when the triggering event(s) for the associated object is due. Consequently, the same object identified n will *never* appear in two different entries.

The two-level scheduling strategy restricts the number of entries N in the global queue. This is advantageous since the cost of a queueing operation typically depends on N. Here cost refers to the execution time of a *complete* queueing operation. This includes both an insertion and a removal of an entry.

For every iteration through the object dispatcher loop the frontmost record is re-

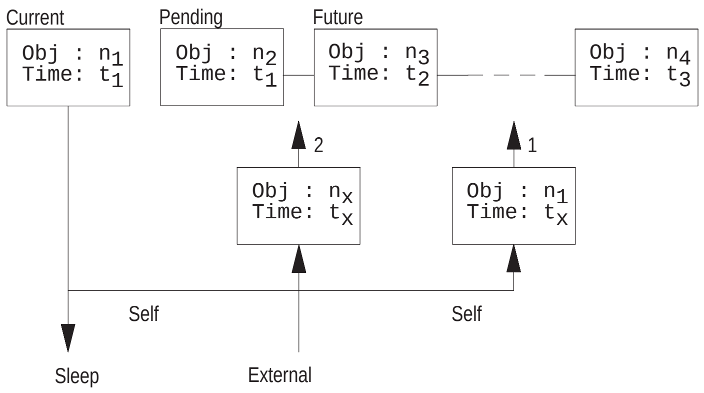


Figure 7.1: *Illustration of how global scheduling takes place.*

moved from the global queue and becomes the current entry. The corresponding current time is denoted t_c. If there are more records scheduled for time $t = t_c$ they are referred to as pending entries.

Control is relinquished to the current object in terms of calling its `nextState` C++ hook. If the object has any locally scheduled events[1] on exit from this hook, a new entry will be inserted into the global queue according the next triggering event(s). This is called self-scheduling and corresponds to the case labeled 1 in the figure 7.1. If there are no scheduled events, the object goes asleep at this point. Thus, an object is said to be sleeping if it does not have an associated entry in the global queue.

An entry will also be inserted in the global queue when inter-object communication takes place. A receiving object needs to be scheduled so that the appropriate actions can be taken in response to the written interface. This is called external scheduling and corresponds to the case labeled 2 in the figure.

Compared to the compositional formulation developed in appendix C, we use a restricted approach for external scheduling. The point is that an entry is always prepended to the future list at the *particular* time $t_n = t_c + \Delta t$ where Δt defines the time resolution. We call t_n the next time since this is the most immediate time value being recognized. The concept of resolution effectively discretize time and helps to ensure proper synchronization of interacting objects. Using any other time than t_n for external scheduling is troublesome for two reasons.

- Scheduling at the current time[2] $t = t_c < t_n$ is prohibited since the object may already have been visited. An inherent assumption of the GSMP formulation is that there should be a unique state-transition at a particular time. Hence, multiple visits to the same object is not permitted.

- Using $t > t_n$ is destructive due to the fact that the interfaces of the object can possibly be overwritten in the interval $(t_n, t]$.

External scheduling takes place as just described if the written object is currently sleeping. The inserted entry then leads to awakening of the object. In the case that the object is already due at time t_n there is no need to proceed. The final case is that the object is scheduled for some future time $t > t_n$. External scheduling is then completed but only after the future entry is removed from the queue. Removal is a safe operation since the associated future event is also stored locally as the triggering event of the object. The entry will eventually be reinserted as a result of a subsequent self-scheduling operation. This happens unless the event is canceled at the next visit.q

7.2 Implementation

Dynamic allocation and release of scheduler queue records become costly if a standard C++ storage allocator is used. For efficiency reasons GMSim implements its own system for management of a pool of queue entries.

Apart from this there are several ways to implement the priority queue used for global scheduling [16–18, 23, 25, 39]. The methods can be classified depending on whether they use time-mapping [3, 20] or maintain a balanced tree structure [19, 37].

The former class employs the principle of hashing [33]. It has the potential of performing a queueing operation in $O(1)$ time, thus being independent of queue size N.

[1]I.e. if either the set of old events or the set of new events is nonempty.

[2]This is often called now-scheduling.

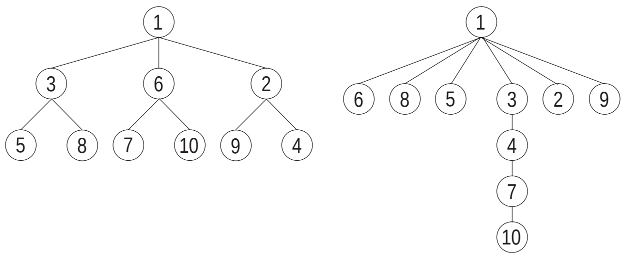


Figure 7.2: *Illustration of a well-balanced (left) and a badly balanced heap (right).*

Unfortunately, this works well only if N and the scheduling distribution does not vary too much during the course of a simulation. The latter refers to how an arbitrary time designation t_i is distributed [3] at insertion. The point is that an efficient time-mapping strategy depends on an almost uniform hashing into a fixed set of underlying bins. I.e. the number of bins should be kept constant, with each bin holding approximately the same number of entries. The number of entries per bin should also be bounded.

The tree based methods are more robust to dynamic variations in N and the scheduling distribution. The provision is that a proper balancing strategy is used. This next section will elaborate on this issue. The motivation for balancing is to keep a queueing operations bounded by $O(\log N)$. Since N can often be quite large in complex simulations, logarithmic behavior is essential. One way to achive tree balancing is to impose a structural constraint and reorganize the tree accordingly at each access. A less strict approach is to use restructuring heuristic which do not guarantee that the tree is always balanced. However, amortized over a large number of accesses the tree will be sufficiently balanced for the $O(\log N)$ bound to apply.

7.3 Pairing heaps

The heap [33] is a tree structure which is well-suited for priority queues. The heap condition states that any entry in the tree should be smaller than all its subordinate entries. Or to put it another way, that the father of any entry is no greater than the entry itself. The ordering of entries depends on some key, of course. In our case this is the time designation.

Figure 7.2 gives two examples of heap-ordered trees. Both heaps comprise the same entries, but they are very different with respect to balancing. Intuitively we would say that the leftmost heap is well-balanced whereas the balance of the rightmost heap is bad. Specifically, there are two structural requirements to consider regarding balance

- Each node should have approximately the same number of childrens. Further, this number should not be too larger.
- Each leaf of the tree should have almost the same depth.

Linking is used a primitive for combining two heap-ordered trees. This is illustrated in figure 7.3. The smallest root becomes the father of the larger root and also the root of the combined tree. If strict balancing is imposed, the combined tree may need to be further restructured. This is accomplished in terms of subsequent split and link

[3] E.g. it is often assumed that an exponential distribution yields a good aproximation in the general case.

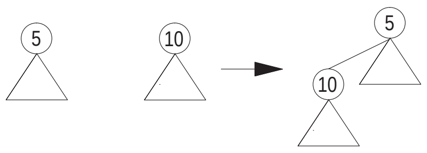


Figure 7.3: *Linking two heap-ordered trees. The triangles deonte trees of arbitrary structure.*

operations. Splitting refers to the oposite operation of linking and should be used at points where bad balance occurs.

In GMSim a pairing heap [8] strategy is used. Rather than keeping the tree strictly balanced, a simple restructuring heuristic is employed. The idea is best explained by comparing figure 7.4 and figure 7.5. Both figures consider the case when four heaps are to be merged. Combination of more than two heaps occurs frequently in practice. The typical case is to combine the subtrees rooted by the children of a removed father.

In figure 7.4 the heaps are combined sequentially in three steps by starting from the left and linking the next sub-heap with the combined tree from the previous step. In figure 7.5 the sub-heaps are linked *pairwise* until a single rooted tree is obtained after the second step. The point is that the pairing strategy results in a heap with is better balanced.

Even if this example is deliberatly choosen to highlight the difference, is can be argued [8] that pairing is generally a better strategy. This is especially true when the resulting heap is *not* further restructued to obtain perfect balance. This is called partial restructuring and is the basis for the pairing heap algorithm. Provided that the insertion order is preserved, it can be shown that a pairing heap is sufficently balanced to obtain a $O(\log N)$ bound in the amortized sense. Hence, the partial restructuring heuristic leads to a self-adjusting data structure.

To explain what is ment by insertion order it is most convenient to use the sibling representation of a heap. This is shown in figure 7.6 and corresponds to the balanced heap in figure 7.2. The point is that the childrens of a node are ordered according to the sequence in which they were attached to their father by linking operations. The first (youngest) child should always be the one most recently attached. Note that this ordering of children is independent of the key order.

The pairing heap algorithm implemented by GMSim uses lazy or late combination of sub-heaps. This means that combine operations are never performed before actually needed. This has the consequence that little effort is required for insertion. Rather, most of the work is associated with retrival of the minimum entry. The entries of the pairing heap will move closer to their proper place each time this operation is performed. To put it another way, the administration cost associated with an entry increases gradually with its lifteime in the queue. This is a nice feature as it keeps the waste at a minimum if entries are exceptionally removed from the queue. In GMSim this happens when a future entry is removed in response to external scheduling. We conjecture that this will happend quite frequently in complex simulations.

External shceduling at the next time t_n can efficiently be incorporated in the pairing heap algorithm as a special case. This is important as scheduling at this particular time is most probably a very frequent operation in any simulation application.

In sum, the paring heap algorithm is efficient in most operational environments. In particular, GMSim is insensitive to dynamic variations in the scheduling distribution and the number of entries in the queue. We also think that a pairing heap strategy fits

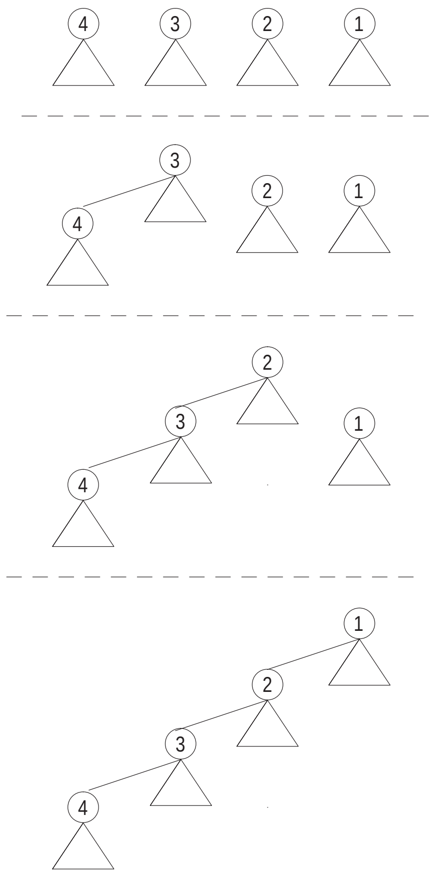


Figure 7.4: *Sequential combination of heaps.*

well with a compositional GSMP view.

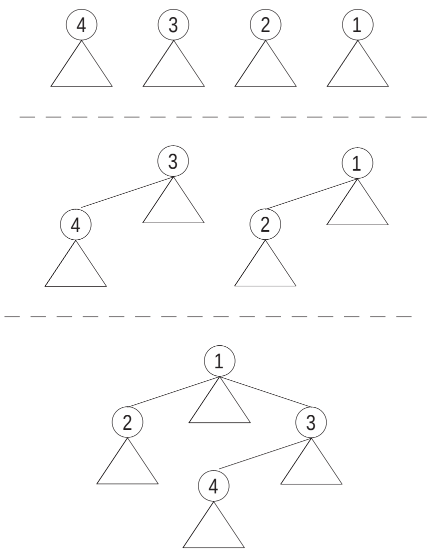


Figure 7.5: *Pairwise combination of heaps.*

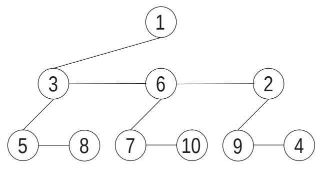


Figure 7.6: *Sibling representation of a heap.*

Chapter 8

Summary

The GSMP framework provides the theoretical foundation for the GMSim development. The framework builds on the successful heritage from Markov chain analysis. Generalized Markov behaviour is obatained by adding suplementary variables in a systematic way.

One immediate consequence of the GSMP view is that theoretically sound and computationally efficient methods for experimental design, sampling and output analysis are readily available. E.g. asynchronous sampling [2, 7], which is generally considered to be efficient, follows easily from the GSMP view [12].

The GMSim development takes the GSMP framework one step further and provides a corresponding simulation environment. The novelity is that this is based on a compositional GSMP view which fits with an object-oriented formulation. The resulting programming environment is in close resemblance with the underlying formalism. We conjecture that this leads to a consistent and efficient implementation.

Model development takes place in terms of C++ programming augmented by a set of M4 macros. As these macros are expanded code is automatically generated to fit with the core. This reduces the programming efforts. Rather than letting objects call the core, the core invokes objects in terms of predefined hook functions. This provides a structured programming interface.

Another utility is that application specific packages are incorporated by run-time linking. Hence, the basic capabilities of GMSim can be extended in arbitrary ways without recompiling the core. The fact that binary code are build for new components contributes to reduced execution time.

Flexibility is further enhanced by an embedded Tcl command interpreter. In fact, GMSim itself comprises an enriched set of script commands enabling simulation. Hence, simulation setup, reporting and control take place entirely in terms script programming. Debugging facilities and a convenient system for setting of configuration parameters are also available from the script domain. The interpretative nature makes GMSim suitable both for interactive use and for batched runs. The GMSim tool itself is also extendable by script programming. A small set of predefined script hooks which are invoked from within the core opens for modified behaviour.

In GMSim each object is responsible for proper scheduling of its own event. In fact, local scheduling is an intrinsic part of the GSMP formulation. The objects are in turn arranged by a global priority queue. Hence, GMSim is characterized by a two-level scheduling strategy. This is efficient since the number of entries N in the global queue is reduced. Nevertheless, N can become large and logarithmic $O(\log N)$ bounds for the basic queueing operations are essential.

In GMSim a tree based pairing heap [8] algorithm is used to maintain the global queue. The algorithm employs a lazy restructuring heuristic for the heap-ordered tree rather than strict balancing. The algorithm is insensitive both to dynamic variantions in N and also in the scheduling distribution. In the amortized sense a queueing operation will be bounded by $O(\log N)$. A salient feature is that the administration cost associated with an entry depends mainly on its *life-time* in the queue and less on the number of entries N at any particular time. Hence, insertion is bounded by $O(1)$ and the waste is kept at a minimum when an entry is exceptionally removed from the queue. We argue that this is a nice feature since exceptional removal will most likely be a frequently occuring case when GSMP modeling is used. In sum, we argue that the paring heap strategy fits well with the compositional GSMP view and that it performs well under various operation conditions.

Finally, the GMSim source code is distributed to the public and depends only on freely available standard components. Thus, it is an open-ended system and should be suitable for experimental research on new simulation methods.

Appendix A

Basics of Tcl/Tk

The Tcl/Tk environment [27] has gained increasing popularity in recent years and is used for a variety of applications. The interested reader should take a look at

`<http://sunscript.sun.com>`

which is an excellent collection on information.

Tcl is a versatile scripting language for controlling and extending applications; its name stands for "tool command language". It is an *interpreted* language and the basic capability of an interpreter is to expand, parse and evaluate sequences of textual commands.

In some respect Tcl is similar to Lisp since a command is always a list. The first element is the name of the command and the remaining elements are arguments. The elements are subject to expansion before the list itself is evaluated as a command. Hence, the structure of Tcl is simple but yet powerful. The standard application for evaluation of Tcl scripts is called `tclsh` which stands for Tcl-shell.

In addition to generic programming facilities like variables and control structures, the standard Tcl interpreter recognizes primitive commands like file based input/output and textual manipulations of strings and list. The built-in commands can be combined in terms of procedures to accomplish more complex tasks. Each procedure defines a new command. Local variables with restricted scoping is supported and procedures can be nested in arbitrary ways.

To give an idea of programming in Tcl, consider the following script which is suitable for evaluation by `tclsh`.

```
#!/usr/bin/tclsh

# Returns reverse of list
proc reverse {list} {
    set res ""
    foreach element $list {
        if {$res == ""} {
            set res $element
        } else {
            set res [concat $element $res]
        }
    }
    return $res
}

# Hello, world example
set hw [list Hello world]
puts "$hw, [reverse $hw]"
```

The script illustrates some of the built-in commands. Variables are expanded if the name is preceded by `$`. Bracketed expressions `[...]` are used to request command expansion. In contrast, anything embraced by `{...}` is protected from expansion.

If the standard Tcl commands are insufficient for a particular application, new commands can also be defined in terms of C programming. In fact, Tcl is completely embeddable. Its interpreter is a library of C functions which can be linked with any application. This opens for extending the basic capabilities of Tcl in arbitrary ways. E.g. the following program sets up an interpreter and defines one new command called `myecho`.

```
#include<tcl.h>

/*-----------------------------------------*/
int myecho_cmdProc(ClientData clientData, Tcl_Interp *interp,
                   int argc, char **argv) {
  Tcl_SetResult(interp, argv[1], TCL_VOLATILE);
  return TCL_OK;
};

/*-----------------------------------------*/
int initProc(Tcl_Interp *interp) {
  Tcl_Init(interp);
  Tcl_CreateCommand (interp, "myecho",
                     myecho_cmdProc,
                     (ClientData) NULL,
                     (Tcl_CmdDeleteProc *) NULL);
  return TCL_OK;
};

/*-----------------------------------------*/
int main(int argc, char **argv) {
  Tcl_Main(argc, argv, initProc);
};
```

The `Tcl_Main` function prepares the interpreter whereas the `initProc` function registers the new command. The command itself is implemented by the function `myecho_cmdProc`. The action of the `myecho` command is simply to return its first argument. If this file is compiled and linked with the appropriate libraries[1], Tcl becombes embedded in the application.

The most useful extension to Tcl is called Tk. It defines a set of primitive commands for building user interfaces in a windowing environment. The utility is that GUI development may take place in terms of script programming rather that writing Ccode. The standard application for evaluation of Tk script is called `wish`. It is a windowing-shell which is linked with the appropriate libraries for the underlying graphics system.

[1]In particular the Tcl C-library `libtcl.a` or `libtcl.so`.

Appendix B

Basic GSMP formalism

A generalized semi-Markov process (GSMP) belongs to the class of discrete-event stochastic processes. It is based on the notion of a state, and makes a state transition when an event associated with the occupied state occurs. Several possible events compete with respect to triggering the next transition and each of these events has its own distribution for determining the next state. At each transition new events may be scheduled. For each of these events, a clock indicating the time until the event is scheduled to occur is set according to an independent mechanism. If a scheduled event does not trigger a transition but is associated with the next state, its clock continues to run. If such an event is not associated with the next state, it ceases to be scheduled and its clock reading abandoned.

The standard definition of a GSMP [9, 10, 12, 30–32, 35] assumes that the set of scheduled events is uniquely determined by the current state. It is also assumed that there is a unique triggering event for each state transition. We use an extended definition where the set of scheduled events is explicitly given and where multiple triggering events is allowed. The former is based on [13] and the latter on [36].

Formal definition of a GSMP is in terms of an embedded Markov chain $\mathbf{X}_k$ that describes a continous-time process $\mathbf{S}(t) \in \mathcal{S}$ at successive epochs of state transition. A one-dimensional illustration is provided in figure B.1. Note that $\mathcal{S}$ signifies an application specific state space which is assumed to be finite or countable. A GSMP process is multi-dimensional in the general case, hence the vector notation.

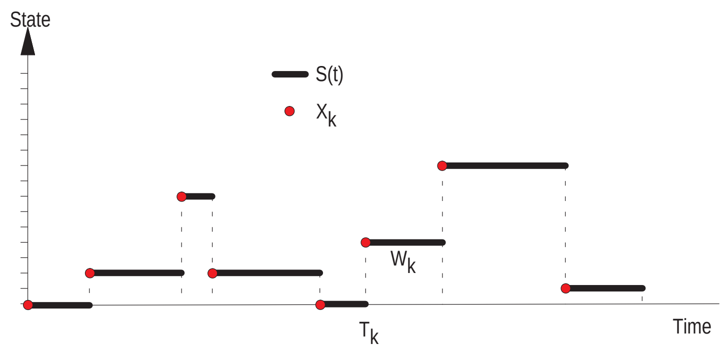


Figure B.1: *One-dimensional illustration of a generalized semi-Markov process.*

At entrance to a new state $\mathbf{X}_k$ at time T_k we associate a set of active (scheduled) events $\mathcal{I}_k \subseteq \mathcal{E}$. Here $\mathcal{E} = \{e_1, e_2, \ldots, e_m\}$ is taken to be a set of events defined specifically for the application. For each active event $e_i \in \mathcal{I}_k$ the product of an associated clock c_i and running speed r_i gives the time until the event is scheduled to occur. The scheduled events compete with respect to triggering the next transition at time T_{k+1}. The winner(s) are the event(s) with the minimum remaining time. This is the set of triggering events denoted $\mathcal{D}^* \subseteq \mathcal{I}_k$. The events $e_j \in (\mathcal{E} - \mathcal{I}_k)$ are classified as inactive. The inter-event time $W_k = T_{k+1} - T_k$ is called the sojourn time in state $\mathbf{X}_k$.

The embedded state description arises from augmenting the natural state vector $\mathbf{S}(T_k)$ with event clocks $\mathbf{C}_k = (c_1, \ldots, c_m)$, hence $\mathbf{X}_k = (\mathbf{S}(T_k), \mathbf{C}_k)$. This approach is related to the supplementary variable technique [5] often used to obtain Markov behavior in stochastic models. As always, Markov behavior simplifies analysis. A Markov-renewal condition [4] is in turn imposed on the compund chain $(\mathbf{X}_k, W_k)$. In addition time-homogenity [4, 10] is assumed. The details are beyond the scope of this document but there are two major implications. First, a time-invariant Markov transition kernel is associated with the embedded chain. Next, the sojourn times are conditionally independent given the embedded chain and with the distribution of W_k depending only on $\mathbf{X}_k$ and $\mathbf{X}_{k+1}$. This ensures semi-Markov behavior of the process $\mathbf{S}(t)$.

Due to the inherent stochastic restrictions of the GSMP formulation, the embedded chain is completely characterized by a time-invariant single-step behavior. For a departing state $\mathbf{x} = (\mathbf{s}, \mathbf{c})$ the probabilistic transition into the next state $\mathbf{x}' = (\mathbf{s}', \mathbf{c}')$ can be expressed [13] by the joint probability distribution function

$$P(\mathbf{x}, \mathcal{A}) = p(\mathbf{s}'; \mathbf{s}, \mathcal{D}^*, \mathbf{c}^*) \prod_{e_i \in \mathcal{N}} F(a_i; \mathbf{s}', e_i, \mathbf{s}, \mathcal{D}^*) \prod_{e_i \in \mathcal{O}} I[0, a_i](c_i^*) \tag{B.1}$$

Here $\mathcal{A}$ is a subspace for $\mathbf{x}'$ corresponding to the case that natural state $\mathbf{s}'$ is entered and the clock reading associated with active event e_i set to a value $c_i' \in [0, a_i]$. Hence,

$$\mathcal{A} = \{\mathbf{s}'\} \times \{\mathbf{c}' \,:\, 0 \leq c_i' \leq a_i \text{ for } e_i \text{ active, } c_i' = 0 \text{ otherwise}\}$$

The set $\mathcal{D}^*$ in equation (B.1) refers to the triggering events and $\mathbf{c}^*$ is the vector of updated clock readings just prior to the transition. Further, $\mathcal{N} = \mathcal{N}(\mathbf{s}'; \mathbf{s}, \mathcal{D}^*, \mathbf{c}^*)$ is a set of new events becoming active due to the transition and $\mathcal{O} = \mathcal{O}(\mathbf{s}'; \mathbf{s}, \mathcal{D}^*, \mathbf{c}^*)$ is the set of old events remaining active. For each old event $e_i \in \mathcal{O}$ we set $c_i' = c_i^*$ keeping the updated clock reading after the transition. New clock readings are generated for each event $e_i \in \mathcal{N}$. A family of probability distribution functions $F(\cdot; \mathbf{s}', e_i, \mathbf{s}, \mathcal{D}^*)$ is defined so that $F(a_i; \mathbf{s}', e_i, \mathbf{s}, \mathcal{D}^*)$ is the conditional probability that event e_i is scheduled with a new clock value $c_i' \in [0, a_i]$.

Each remaining event $e_i \in (\mathcal{E} - \mathcal{N} \cup \mathcal{O})$ is cancelled by setting its clock and speed $c_i' = r_i = 0$. Finally, $p(\cdot; \mathbf{s}, \mathcal{D}^*, \mathbf{c}^*)$ is a family of probability density functions so that $p(\mathbf{s}'; \mathbf{s}, \mathcal{D}^*, \mathbf{c}^*)$ denotes the probability that the next state is $\mathbf{s}'$.

Note the product form of equation (B.1) suggesting that independence is at play. This contributes to the analyticity of a GSMP.

To complete the GSMP formulation, a probability distribution function $\tilde{P}(\cdot)$ concerning the initial state is defined by

$$\tilde{P}(\mathcal{A}) = \tilde{p}(\mathbf{s}) \sum_{\mathcal{I}} \tilde{h}(\mathcal{I}; \mathbf{s}) \prod_{e_i \in \mathcal{I}} \tilde{F}(a_i; e_i, \mathbf{s}, \mathcal{I})$$

where

$$\mathcal{A} = \{(\mathbf{s}, \mathbf{c}) : 0 \leq c_i \leq a_i\}$$

An initial natural state component $\mathbf{s}$ is first picked according to a probability density function $\tilde{p}(\cdot)$. Then an initial set $\mathcal{I}$ of active events is selected according to a conditional probability density function $\tilde{h}(\cdot; \mathbf{s})$. Finally, for each initially active event e_i the corresponding initial clock reading c_i is set (independently) according to a probability distribution function $\tilde{g}(\cdot; e_i, \mathbf{s}, \mathcal{I})$.

Appendix C

Compositional GSMP view

The basic GSMP formulation outlined in appendix B is tractable for simple applications. However, as the number components in the state description and the number of events grow, combinatorial explosion quickly arises. This means that the number of significant state/event combinations to consider become too numerous to handle. This is especially due to the fact that multiple triggering events is allowed. As always, the solution is to decompose the problem.

In this appendic a compositional GSMP view is developed. This is based on establishing a particular *separability* condition. It starts with regarding the state space $\mathcal{S}$ in terms of three components indexed by a, b and c:

$$\mathcal{S} = \mathcal{S}_a \times \mathcal{S}_b \times \mathcal{S}_c$$

Hence, we write the natural state vector as $\mathbf{s} = (\mathbf{s}_a, \mathbf{s}_b, \mathbf{s}_c)$. Accordingly, we partition the set of events $\mathcal{E}$ in three disjunct subsets

$$\mathcal{E} = \mathcal{E}_a \cup \mathcal{E}_b \cup \mathcal{E}_c$$

and write $\mathbf{c} = (\mathbf{c}_a, \mathbf{c}_b, \mathbf{c}_c)$ for the clock vector. Finally, the set of triggering events is decomposed in the same way

$$\mathcal{D}^* = \mathcal{D}^*_a \cup \mathcal{D}^*_b \cup \mathcal{D}^*_c \text{ where } \mathcal{D}^*_j \subseteq \mathcal{E}_j \text{ for } j = a, b, c$$

For convenience we will use double-indexing to refer to two components simultaneously. E.g. $\mathbf{c}_{b,c} = (\mathbf{c}_b, \mathbf{c}_c)$ and $\mathcal{E}_{b,c} = \mathcal{E}_b \cup \mathcal{E}_c$.

In order to arrive at a separable process we impose certain independence restrictions. Our interest is independence in the sense that components a and b are conditioned on a only, whereas component c is conditioned on both b and c. This is illustrated in figure C.1. The underlying idea is to facilitate an object-orient view. Component a takes the role of a sending object whereas component c corresponds to a receiving object. Component b is used to capture one-way inter-object communication. It corresponds to the concept of an interface as discussed in section 4.7. The figure suggests that the interface is considered to be part of the receiving object. This explains why component c is conditioned on both b and c. That component b is conditioned on a only, reflects the fact that an interface has no self-driving capabilities.

Formalistically, the separability condition translates into the following list of requirements

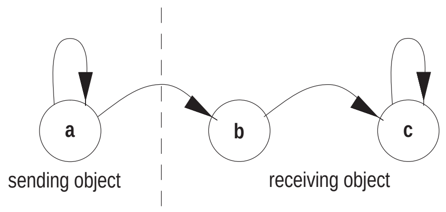


Figure C.1: *The idea of a compositional GSMP view.*

- The probability density functions $p(\cdot;\mathbf{s},\mathcal{D}^*,\mathbf{c}^*)$ are separable in the sense that

$$p(\mathbf{s}';\mathbf{s},\mathcal{D}^*,\mathbf{c}^*) = p(\mathbf{s}'_a;\mathbf{s}_a,\mathcal{D}^*_a,\mathbf{c}^*_a)\cdot p(\mathbf{s}'_b;\mathbf{s}_a,\mathcal{D}^*_a,\mathbf{c}^*_a)\cdot p(\mathbf{s}'_c;\mathbf{s}_{b,c},\mathcal{D}^*_{b,c},\mathbf{c}^*_c)$$

- New events are generated component-wise according to

$$\mathcal{N} = \mathcal{N}_a(\mathbf{s}'_a;\mathbf{s}_a,\mathcal{D}^*_a,\mathbf{c}^*_a)\cup\mathcal{N}_b(\mathbf{s}'_a;\mathbf{s}_a,\mathcal{D}^*_a,\mathbf{c}^*_a)\cup\mathcal{N}_c(\mathbf{s}'_{b,c};\mathbf{s}_{b,c},\mathcal{D}^*_{b,c},\mathbf{c}^*_c)$$

- Decision about which old events to retain are made component-wise according to

$$\mathcal{O} = \mathcal{O}_a(\mathbf{s}'_a;\mathbf{s}_a,\mathcal{D}^*_a,\mathbf{c}^*_a)\cup\mathcal{O}_b(\mathbf{s}'_a;\mathbf{s}_a,\mathcal{D}^*_a,\mathbf{c}^*_a)\cup\mathcal{O}_c(\mathbf{s}'_{b,c};\mathbf{s}_{b,c},\mathcal{D}^*_{b,c},\mathbf{c}^*_c)$$

- New events are scheduled according to component-wise probability distribution functions

$$\begin{gathered}F_a(\cdot;\mathbf{s}'_a,e_i,\mathbf{s}_a,\mathcal{D}^*_a)\\ F_b(\cdot;\mathbf{s}'_b,e_i,\mathbf{s}_a,\mathcal{D}^*_a)\\ F_c(\cdot;\mathbf{s}'_{b,c},e_i,\mathbf{s}_{b,c},\mathcal{D}^*_{b,c})\end{gathered}$$

- Initial state and initial events are set independently for components a and c in the obvious way. For component b no events are allowed to be scheduled initially.

The decomposition strategy just described can be applied repeatedly, of course. In sum, arbitrary complex models can be developed in a compositional setting. This is the theoretical foundation for the object-oriented implementation of GMSim.

Appendix D

System parameters

The following sections document the global configuration options recognized by the `SysPar` parameter class. These parameters are related to generic simulation control.

D.1 `-seed`

Sets the seed used for pseudo-random number generation. Using the same seed ensures reproducibility. Note that this is only valid for distributions implemented as a part of the GMSim tool.

D.2 `-doneTm`

Sets an unconditional stop time for the simulation. When this time is reached control always returns to the script domain with a response `S SysDone`. A simulation cannot proceed from this point. If a zero value is supplied the simulation continues until there are no more scheduled objects.

D.3 `-breakTm`

This is similar to `-doneTm` except that is responds with `S SysBreak` and breaks. A simulation can always proceed from this point.

D.4 `-deadCheck`

As discussed in section 6.3 this parameters sets the threshold defining a global deadlock condition. Deadlock checking is turned off if a zero value is supplied. Whenever deadlock is encountered control returns to the script domain with as response `S SysDeadlock` and breaks. A simulation can always proceed from deadlock.

D.5 `-regVerbInt`

If this parameter is set to a non-zero value, control return to the script domain at the specified regular interval. The response is `S SysRegVerbInt`. This feature is normally used in conjunction with a system hook to do regular reporting. Note that a continuous run will not be breaked when this condition occur.

D.6 `-regBreakInt`

This is similar to `-regVerbInt` except that the response is `S SysRegBreakInt` and the the run is always breaked.

Appendix E

The `mm1` package

To get an idea of package development for GMSim, this appendix contains listings for the `mm1` package considered in chapter 4.3. The package defines three classes: `ExprArr`, `Queue` and `ExpSrv`. Their function should be obvious from the names. Together they form a model of a discrete-time M/M/1 queuing system. For background information on classical queuing theory see [21].

Note that the M/M/1 queuing system could very well have been implemented in terms of a single class. However, when arrivals and serving are handled by separate classes, we get the opportunity to illustrate linking and interfaces. In section E.4 we consider a specific example using the `mm1` package. The involved objects, classes, links and interfaces are as shown in figure E.1 for the example.

E.1 Header file `mm1.h`

The header file is associated with the source file `mm1.cc` listed in the next section. The distinction between real and virtual classes is illustrated by the empty real classes `ExprArr`, `Queue` and `ExpSrv` which are based on the corresponding virtual classes `ExprArrB`, `QueueB` and `ExpSrvB`. The virtual classes contain the body of declarations in terms of hook functions.

Three interfaces are also defined: `QArrIfc`, `QDepIfc` and `SrvIfc`. The first two are used by the queue class. The latter is used by the server class. The queue link of the `ExpArrB` class match the `QArrIfc` interface of the `QueueB` class. Likewise, there are two matching link/interface pairs between the `QueueB` and `ExpSrvB` classes.

The `ExpArrB` class defines one arrival event and also one measure function designed to count arrivals. The `ExpSrvB` class defines one departure event. The `QueueB` class has no events defined. Its action depends on information written to one of its in-

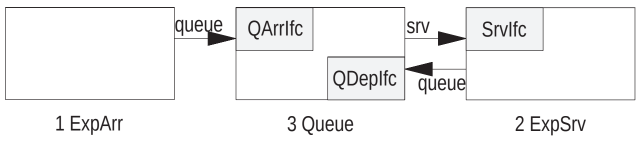


Figure E.1: *The involved objects, classes, links and interfaces for the `ex.tcl` example.*

terfaces. Hence, this class has no self-driving capabilities. Note finally that the variable `num` of the `QueueB` class comprises the state description for the model. It reflects the number of customers in the queue at any point in time.

```
// This may look like C code, but it is really -*- C++ -*-

/*
  GMSim/mm1: M/M/1 queue model
  Copyright (C) 1998  Frode B. Nilsen

  This program is free software; you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation; either version 2 of the License, or
  (at your option) any later version.

  This program is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  A copy of the GNU General Public License is available in the file
  COPYING at the root of the GMSim source distribution tree.
*/

//
// Note: se the README file for information on this sample package
//

#ifndef mm1_mm1_h__
#define mm1_mm1_h__

#pragma interface

#include "core/sim.h"

/*---------------------------*/
sim_ifc(QArrIfc) {
sim_data:
  bool arr;
sim_hooks:
  void verbose (ostream &os) {
    os « Sim_Indent("arr") « toStr(arr) « endl;
  };
  void ifcClear(void) {
    arr=false;
  };
sim_body:
  void arred(void) {
    arr = true;
  };
};

/*---------------------------*/
sim_ifc(QDepIfc) {
sim_data:
  bool dep;
sim_hooks:
  void verbose (ostream &os) {
    os « Sim_Indent("dep") « toStr(dep) « endl;
  };
  void ifcClear(void) {
    dep=false;
  };
sim_body:
  void deped(void) {
    dep = true;
  };
};

/*---------------------------*/
sim_ifc(SrvIfc) {
sim_data:
  bool srv;
sim_hooks:
  void verbose (ostream &os) {
    os « Sim_Indent("srv") « toStr(srv) « endl;
  };
  void ifcClear(void) {
    srv=false;
  };
sim_body:
  void srved(void) {
    srv = true;
  };
};

/*---------------------------*/
sim_vaclass(ExpArrB) {
  sim_events(ARR);
  sim_links(queue QArrIfc);
  sim_measures(ArrCnt);
sim_data:
  int marr;
  Sim_Geometric arr;

sim_hooks:
  void objParSet (Sim_Opt &opt);
  void objParGet (EString &lst);
  void objInit(int mode);
  void nextState (void);
  Sim_UsrTime initOccur (int ev) {
    return nextOccur(ev);
  };
  Sim_UsrTime nextOccur (int ev);
  bool ArrCnt (double &dat);
sim_body:
};

/*---------------------------*/
sim_vaclass(ExpSrvB) sim_use(SrvIfc) {
  sim_events(SRV);
  sim_links(queue QDepIfc);
sim_data:
  int msrv;
  Sim_Geometric srv;
sim_hooks:
  void objParSet (Sim_Opt &opt);
  void objParGet (EString &lst);
  void objInit(int mode);
  void nextState (void);
  Sim_UsrTime initOccur (int ev) {
    return nextOccur(ev);
  };
  Sim_UsrTime nextOccur (int ev);
sim_body:
};

/*---------------------------*/
sim_vaclass(QueueB) sim_use(QArrIfc,QDepIfc) {
  sim_links(srv SrvIfc);
sim_data:
  int num;
  bool insrv;
sim_hooks:
  void verbose (ostream &os) {
    os « Sim_Indent("num") « toStr(num) « endl;
  };
  void objInit(int mode);
  void nextState (void);
sim_body:
};

/*---------------------------*/
sim_raclass(ExpArr) sim_use(ExpArrB) {
sim_body:
};
```

```
/*---------------------------*/
sim_raclass(ExpSrv) sim_use(ExpSrvB) {
sim_body:
};

/*---------------------------*/
sim_raclass(Queue) sim_use(QueueB) {
sim_body:
};

#endif
```

E.2 Source file `mm1.cc`

This file contains the implementation of the hook functions declared in `mm1.h`. Note that the `#siminclude` statement at the beginning is really a M4 macro call indicating that that the included header file should be expanded. Files incorporated by an ordinary include directive are not expanded.

Each of the arrival and server classes recognizes one configuration parameters corresponding to mean inter-arrival time and mean service (departure) time. The parameters are set and read by the `objParSet` and `objParGet` hooks, respectively.

Otherwise, the implementation is to a large extent self-explanatory. A thorough understanding requires that the full source distribution is consulted. It is beyond the scope of this document, however.

```
/*
  GMSim/mm1: M/M/1 queue model
  Copyright (C) 1998  Frode B. Nilsen

  This program is free software; you can redistribute it and/or modify
  it under the terms of the GNU General Public License as published by
  the Free Software Foundation; either version 2 of the License, or
  (at your option) any later version.

  This program is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU General Public License for more details.

  A copy of the GNU General Public License is available in the file
  COPYING at the root of the GMSim source distribution tree.
*/

#pragma implementation

// Plain C Header files

// Sim includes
#siminclude(mm1/mm1.h)

/*---------------------------*/
void ExpArrB::objParSet (Sim_Opt &opt) {
  if (opt.take("-marr")) {
    double d = opt.val;
    if (d < 0)
      opt.errVal();
    else
      arr.mean(d);
  };
};

/*---------------------------*/
void ExpArrB::objParGet (EString &lst) {
  LAPP(lst,"-marr");
  LAPP(lst,toStr(arr.mean()));
};
```

```
/*---------------------------*/
void ExpArrB::objInit(int mode) {
  if (mode == 2)
    newEvs.set(ExpArrB_ARR);
};

/*---------------------------*/
void ExpArrB::nextState (void) {
  // Arrival: send to queue and prepare next
  if (trigEvs[ExpArrB_ARR]) {
    newEvs.set(ExpArrB_ARR);
    sim_tell(queue[0], arred());
  };
};

/*---------------------------*/
Sim_UsrTime ExpArrB::nextOccur (int ev) {
  if (ev == ExpArrB_ARR)
    return Sim_UsrTime(arr.rnd());
  return Sim_noTime;
};

/*---------------------------*/
bool ExpArrB::ArrCnt(double &dat) {
  if (trigEvs[ExpArrB_ARR]) {
    dat++ ;
    return true;
  };
  return false;
};

/*---------------------------*/
void ExpSrvB::objParSet (Sim_Opt &opt) {
  if (opt.take("-msrv")) {
    double d = opt.val;
    if (d < 0)
      opt.errVal();
    else
      srv.mean(d);
  };
};

/*---------------------------*/
void ExpSrvB::objParGet (EString &lst) {
  LAPP(lst,"-msrv");
  LAPP(lst,toStr(srv.mean()));
};

/*---------------------------*/
void ExpSrvB::objInit(int mode) {
  // No specific initialization
};

/*---------------------------*/
void ExpSrvB::nextState (void) {
  // start service
  if (theSrvIfc.srv) {
    newEvs.set(ExpSrvB_SRV);
  };
  // service ended
  if (trigEvs[ExpSrvB_SRV]) {
    sim_tell(queue[0], deped());
  };
};

/*---------------------------*/
Sim_UsrTime ExpSrvB::nextOccur (int ev) {
  if (ev == ExpSrvB_SRV)
    return Sim_UsrTime(srv.rnd());
  return Sim_noTime;
};

/*---------------------------*/
void QueueB::objInit(int mode) {
  num = 0;
  insrv = false;
};

/*---------------------------*/
void QueueB::nextState (void) {
  // Arrival
  if (theQArrIfc.arr) {
    num++;
  };
  // Server no ready
  if (theQDepIfc.dep) {
    insrv=false;
  };

  // Pass to server
  if (num > 0 && !insrv) {
    num-;
    insrv=true;
    sim_tell(srv[0], srved());
  };
};
```

E.3 Configuration file `ExpArrB.cfg`

This file holds information required to build the configuration template for the `ExpArrB` class. There is a single line corresponding to the recognized mean inter-arrival configuration parameters.

```
1.1 "Mean interarrival time" -marr 100
```

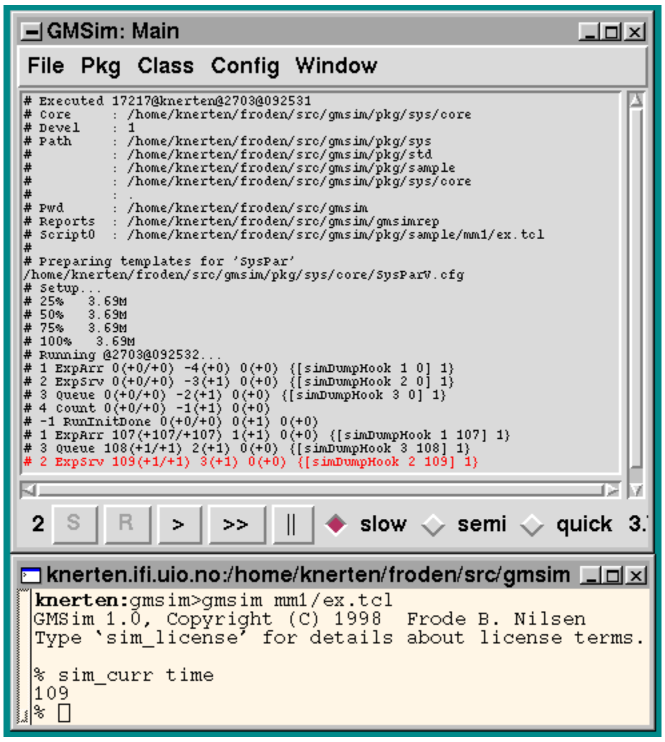
E.4 Script file `ex.tcl`

The script file sets up and starts a simulation using the `mm1` package. One object for each of the classes are instantiated and then linked. A statistics counting the arrival is also created. The statistics is assigned to the arrival object. An associated statistics hook is also defined. It will be called whenever the statistics is affected. A dump window is prepared for each of the objects along with a report window for the arrival statistics.

The appearing windows will be as shown in figures E.2 and E.3. when the script is run. The dump windows for each of the three objects are shown in the latter figure. The bottomost window in figure E.2 is the command interpreter from where GMSim was launched. The main window shows the log as the script file is evaluated.

The reader is strongly recommended to check the documentation for each sim the `sim_xxx` commands in this example.

```
# Supress all command in log
sim_log mask ""

# script hook used by arrival statistics
proc arrHook {who num} {
    if {$num > 0} {
        # write time and current count to report
        sim_rep write arep "[sim_curr time]: [sim_stats read $who]"
    }
    # invoke hook at every arrival
    return [list [expr $num + 1] ""]
}

# This example depends on mm1 package
sim_pkg require mm1
```

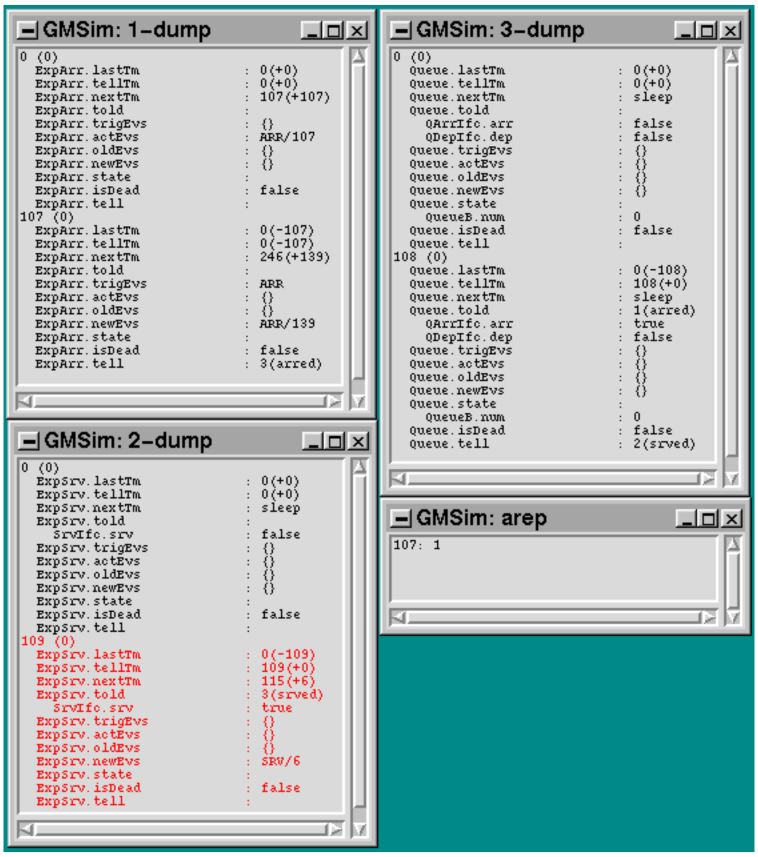


```
# Executed 17217@knerten@2703@092531
# Core     : /home/knerten/froden/src/gmsim/pkg/sys/core
# Devel    : 1
# Path     : /home/knerten/froden/src/gmsim/pkg/sys
#          : /home/knerten/froden/src/gmsim/pkg/std
#          : /home/knerten/froden/src/gmsim/pkg/sample
#          : /home/knerten/froden/src/gmsim/pkg/sys/core
#          : .
# Pwd      : /home/knerten/froden/src/gmsim
# Reports  : /home/knerten/froden/src/gmsim/gmsimrep
# Script0  : /home/knerten/froden/src/gmsim/pkg/sample/mm1/ex.tcl
#
# Preparing templates for 'SysPar'
/home/knerten/froden/src/gmsim/pkg/sys/core/SysParV.cfg
# Setup...
# 25%   3.69M
# 50%   3.69M
# 75%   3.69M
# 100%   3.69M
# Running @2703@092532...
# 1 ExpArr 0(+0/+0) -4(+0) 0(+0) {[simDumpHook 1 0] 1}
# 2 ExpSrv 0(+0/+0) -3(+1) 0(+0) {[simDumpHook 2 0] 1}
# 3 Queue 0(+0/+0) -2(+1) 0(+0) {[simDumpHook 3 0] 1}
# 4 Count 0(+0/+0) -1(+1) 0(+0)
# -1 RunInitDone 0(+0/+0) 0(+1) 0(+0)
# 1 ExpArr 107(+107/+107) 1(+1) 0(+0) {[simDumpHook 1 107] 1}
# 3 Queue 108(+1/+1) 2(+1) 0(+0) {[simDumpHook 3 108] 1}
# 2 ExpSrv 109(+1/+1) 3(+1) 0(+0) {[simDumpHook 2 109] 1}
```

```
knerten:gmsim>gmsim mm1/ex.tcl
GMSim 1.0, Copyright (C) 1998  Frode B. Nilsen
Type `sim_license' for details about license terms.

% sim_curr time
109
% 
```

Figure E.2: *The main and command windows for the* `ex.tcl` *example.*

```
# gstats is a standard package for statistics
sim_pkg require gstats

# class parameters
sim_par set ExpArr "-#queue" 1
sim_par set ExpSrv "-#queue" 1
sim_par set Queue "-#srv" 1

# allocate space for 3 objects and 1 statistics
sim_alloc 4

# create arrival, server and queue objects
set arr [sim_new ExpArr]
set srv [sim_new ExpSrv]
set queue [sim_new Queue]
# create count statistics
set arrcnt [sim_new Count]

# link objects
sim_link set $arr queue $queue
sim_link set $queue srv $srv
sim_link set $srv queue $queue

# prepare for run, slow speed
sim_speed slow
sim_run setup

# assign statistics and set hook
sim_stats assign $arr -ArrCnt $arrcnt
sim_hook add $arrcnt arrHook

# open statistics report and dumps for objects
sim_rep open arep -mode w
sim_rep won arep -width 30 -height 20
```



```
0  (0)
  ExpArr.lastTm                      :  0(+0)
  ExpArr.tellTm                      :  0(+0)
  ExpArr.nextTm                      :  107(+107)
  ExpArr.told                        :
  ExpArr.trigEvs                     :  {}
  ExpArr.actEvs                      :  ARR/107
  ExpArr.oldEvs                      :  {}
  ExpArr.newEvs                      :  {}
  ExpArr.state                       :
  ExpArr.isDead                      :  false
  ExpArr.tell                        :
107  (0)
  ExpArr.lastTm                      :  0(-107)
  ExpArr.tellTm                      :  0(-107)
  ExpArr.nextTm                      :  246(+139)
  ExpArr.told                        :
  ExpArr.trigEvs                     :  ARR
  ExpArr.actEvs                      :  {}
  ExpArr.oldEvs                      :  {}
  ExpArr.newEvs                      :  ARR/139
  ExpArr.state                       :
  ExpArr.isDead                      :  false
  ExpArr.tell                        :  3(arred)
```

```
0  (0)
  ExpSrv.lastTm                      :  0(+0)
  ExpSrv.tellTm                      :  0(+0)
  ExpSrv.nextTm                      :  sleep
  ExpSrv.told                        :
    SrvIfc.srv                       :  false
  ExpSrv.trigEvs                     :  {}
  ExpSrv.actEvs                      :  {}
  ExpSrv.oldEvs                      :  {}
  ExpSrv.newEvs                      :  {}
  ExpSrv.state                       :
  ExpSrv.isDead                      :  false
  ExpSrv.tell                        :
109  (0)
  ExpSrv.lastTm                      :  0(-109)
  ExpSrv.tellTm                      :  109(+0)
  ExpSrv.nextTm                      :  115(+6)
  ExpSrv.told                        :  3(srved)
    SrvIfc.srv                       :  true
  ExpSrv.trigEvs                     :  {}
  ExpSrv.actEvs                      :  {}
  ExpSrv.oldEvs                      :  {}
  ExpSrv.newEvs                      :  SRV/6
  ExpSrv.state                       :
  ExpSrv.isDead                      :  false
  ExpSrv.tell                        :
```

```
0  (0)
  Queue.lastTm                       :  0(+0)
  Queue.tellTm                       :  0(+0)
  Queue.nextTm                       :  sleep
  Queue.told                         :
    QArrIfc.arr                      :  false
    QDepIfc.dep                      :  false
  Queue.trigEvs                      :  {}
  Queue.actEvs                       :  {}
  Queue.oldEvs                       :  {}
  Queue.newEvs                       :  {}
  Queue.state                        :
    QueueB.num                       :  0
  Queue.isDead                       :  false
  Queue.tell                         :
108  (0)
  Queue.lastTm                       :  0(-108)
  Queue.tellTm                       :  108(+0)
  Queue.nextTm                       :  sleep
  Queue.told                         :  1(arred)
    QArrIfc.arr                      :  true
    QDepIfc.dep                      :  false
  Queue.trigEvs                      :  {}
  Queue.actEvs                       :  {}
  Queue.oldEvs                       :  {}
  Queue.newEvs                       :  {}
  Queue.state                        :
    QueueB.num                       :  0
  Queue.isDead                       :  false
  Queue.tell                         :  2(srved)
```

```
107:  1
```

Figure E.3: *The report windows for the* `ex.tcl` *example.*

```
sim_dump won "$arr $queue $srv" -width 40 -height 40

# start run
sim_run start »

# peform 3 state transitions
for {set i 0} {$i < 3} {incr i} {
    sim_run go >
}
```

Appendix F

Core commands

Below is the full documentation of the user significant script commands made available by the `core` package. See figure 4.3 for an overview of mode specific commands. The remaining commands are general and can be invoked in any mode.

F.1 `sim_alloc`

```
sim_alloc <num>
  Assumes mode == -2
  Allocates space for the specified number of items. An item is a dead object,
  alive object or statistics. Items must subsequently be instantiated by
  the ’sim_new’ command.
```

F.2 `sim_curr`

```
sim_curr <what>
  Assumes mode == -2
  Returns information about current status according to specified
  tag. The following tags are recognized at all times, i.e.\ both in
  the ordinary tcl-command loop and any in hook function
    obj    Id of object currently in control
    class  Class tag of object in control
    time   Current simulation time
    trcnt  Current state transition. Note that negative transition
           designations are used in the initialization sequence.
  The following tags only makes sense when called from the  tcl-command loop.
  The info is prepared as the most recently visited object relinquish
  control to the  script domain
    deadcnt Number of dead objects
    dump    Any dump information generated
    msg     Returned messages. This include info about called hook functions
    log     The log entry, i.e.\ response line, written at return to the
            tcl-domain
```

F.3 `sim_dump`

```
sim_dump on <obj-list>
sim_dump off <obj-list>
  Turns dump on/off for specified alive object(s). When dump is turned on a
  corresponding report is opened.

sim_dump won <obj-list> [-width <width> -height <height>]
sim_dump woff <obj-list>
  Opens/close a standard report-window if dump is currently on for specified
  alive object(s).

sim_dump who
  Returns list of active dump-reports

sim_dump reset <obj-list>
  Resets and opened dump-report

sim_dump print <obj-list>
  Prints dump information for specified alive objects. This information
  is always avaialble regardless of wheter dump-reporting
  is turned on or off.

sim_dump all [<file>]
  Prints dump information for all alive objects to specified file. If file
  is not specified, stdout is used and less is spawned.
```

F.4 sim_free

```
sim_free
  Assumes mode == {-1, 0, 1}.
  Deletes instantiated items and frees space allocated by 'sim_alloc'.
  For a new setup phase, 'sim_alloc' must be called again.
```

F.5 sim_h

```
sim_h [<user command>]
  Display available documentation/help for specified command or
  variable. The documentation text is retrieved direcly from the
  source file. (Try sim_h sim_h) If noe help subject is specified, all
  user commands and variables are listed.
```

F.6 sim_hook

```
sim_hook add sys <procname>
  Adds <procname> as a system hook. A system hook is called each time
  control returns to the tcl-domain during a simulation run.

sim_hook add <objid> <procname>
  Adds <procname> as a hook for specified alive object. An object hook
  is called at each vist to an alive object. If an object hook returns
  a non-empty string, the simulation is breaked immediately after the
  visit is completed. The returned value is indicated in the status
  line together with the hook name. An object hook procedure must be
  defined according to 'proc XXX {who time} {...}'. The first argument
  is the nummeric tag of the current object. The second argument is
  the current time


sim_hook add <statsid> <procname>
  Adds <procname> as a hook for specified statistics. A statistics
  hook is called whenever a new sample is formed, and the sample
  number matches a next-designation specified by the user. A
  statistics hook must return a two-element list where the second
  element is the next sample number at which the hook should be
  executed. This is the next-designation. A statistics hook is always
  called at initialization and the returned value gives the first
  next-designation. If the first element of the returned list is a
  non-empty string, the simulation is breaked immediately after the
  current object is completed. The returned value is indicated in the
  status line together with the hook name. A statistics hook procedure
  must be defined according to 'proc XXX {who num} {...}'. The first
  argument is the nummeric tag of the actual statistics. The second
  argument is the current sample number.

sim_hook del sys <procname>
sim_hook del <objid> <procname>
sim_hook del <statsid> <procname>
  Delets <procname> as hook

sim_hook get sys
sim_hook get <objid>
sim_hook get <statsid>
  Returns current hooks
```

F.7 sim_info

```
sim_info sys [-<flag>]...
  Returns system info according to following flags. All flags are returned
  if no flags are specified.
    -nlpkg      List of recognized but unloaded packages
    -lpkg       List of loaded packages
    -aclass     List of recognized alive classes
    -dclass     List of recognized dead classes
    -pclass     List of recognized parameter classes
    -sclass     List of recognized statistics classes
    -naobj      # of objects allocated for registery (see 'sim_alloc')
    -nobj       # of objects currently registered (see 'sim_new')
    -tres       Time resolution
    -mem        Memory allocated by process from OS.
    -hooks      System hooks (see 'sim_hook')
    -mode       Mode designation
    -devel      Indicates if developemt or production version is loaded
    -win        Indicats whether the graphical user-interface is avialble

sim_info <pkg> [-<flag>]...
  Returns info about specified packet according to following flags. All flags
  are returned if no flags are specified
    -depend    Packet dependencies
    -dclass    List of defined dead classes
    -aclass    List of defined alive classes
    -pclass    List of defined parameter classes
    -pclass    List of defined statistics classes
    -cdefs     List of defined symbols used during compilation
    -inst      Installation path

sim_info <class> [-<flag>]...
  Returns info about specified class (real or virtual) according to
  following flags. All flags are returned if no flags are specified.
    -type      Type of class (dead, alive, par, stats)
    -subclass  Flat list of constituent sub-classes.
    -hsubclass Hierarchical sub-class relationship
    -pkg       Package who defines class
    -num       Number of instances
    -who       List of instance tags
    -cpar      List of recognized class parameters
    -opar      List of recognized object parameters
  For alive classes these flags are specifically recognized
    -interf     List of conforming interfaces
    -evs        List of recognized events
    -links      List of recognized link-types. Each type comprises a sub-list
                formatted as {<name> <interface> [OPT]}. The second element
                is the interface requirement for the link. The final element
                is OPT if a link of this type is optional. Oterwise at least
                one link must exist for an object instance.
    -meas       List of defined measure.
  For dead classes these flags are specifically recognized
    -links      See above
  Note that 'sim_par get <class>' gives information about significant class
  parameter settings and default object instance parameter values.
  See also 'sim_xpar' and 'sim_xvpar'

sim_info <obj> [-<flag>]...
  Returns info about specified object instance according to following flags.
  All flags are returned if no flags are specified.
    -type     Type of object (dead, alive, par, stats)
    -class    Instantiated class
  For alive objects these flags are specifically recignized
```

```
    -stats    Assignments of statistics to measures
    -sched    Next time when object is due. If this is emty, the object
              is currently not scheduled.
    -links    Actual links
    -hooks    List of hooks
  For dead objects these flags are specifically recognized
    -links    See above
  For statistics these flags are specifically recognized
    -hooks    List if asociated hooks
    -nextHook Next sample number at which to run hooks (next-designation)
    -numHook  Number of times hook is called
  In addition specific flags are recognized depending on the features
  of the specified object. Finally, note that 'sim_par get <obj>'
  gives information about significant parameters settings. See also
  'sim_xpar' and 'sim_xvpar'. Further, 'sim_link get <obj>' gives info
  about specific links settings and 'sim_stats read <obj>' gives info
  about collected statistics.
```

F.8 sim_link

```
sim_link set <obj> <tag> <idx-list> <to-list>
  Assumes mode == -1
  Sets links of type <tag> emerging from object <obj>. The actual links
  set are given by <idx-list> with corresponding objects destinations in
  <to-list>. The destination objects must have an interface conforming to
  the <tag> link type.

sim_link set <obj> <tag> <to>
  A shorthand for  'sim_link set <obj> <tag> 0 to'

sim_link get <obj> <tag> [<idx-list>]
  Returns object destinations for links of type <tag> emerging from
  object <obj>. If <idx-list> is not given, all links of type <tag> are
  are returned. Otherwise only the specified elements are returned.

sim_link get <obj>
  Returns object destinations for all links of each type emergning from
  object <obj>.
```

F.9 sim_log

```
sim_log mask
  Returns current log-mask

sim_log mask <mask list>
sim_log mask all
  sets log-mask as specified or all recognized levels

sim_log level
  Returns recognized level designations for commands

sim_log level <cmd> [<lvl>]
  Set level designation for given command. If the level parameter is
  left out, the level designation for command is removed.
```

F.10 sim_new

```
sim_new <class> [-<parname> <parval>]...
  Assumes mode == -1
  Instantiates an item, i.e. a dead object, an alive object or
  a statistics, of the specified class. The next available identifier is
  returned. Items are given nummerical tags with numbering starting at
  1. Significant parameters can optionally be supplied.
```

F.11 sim_par

```
sim_par set <class-list> -<parname> <parval> [-<parname> <parval>]...
  Sets template default parameter value(s) for specified class. Defaults are
  settable according to current mode. Multiple classes can be set
  simultaneously provided parameters have identical names.

sim_par set <obj-list> -<parname> <parval> [-<parname> <parval>]...
  Sets significant object parameters for specified object(s). Parameters
  are significant according to current mode. Multiple
  objects can be set simultaneously provided they have identical
  parameter names.

sim_par get <class>  [-<parname>]...
  Get template default parameter values for specified class.

sim_par get <class> | <obj>  [-<parname>]...
  Get significant parameters for specified class or object. Parameters
  are significant according to current mode.
```

F.12 sim_pkg

```
sim_pkg names [<pattern>]
  Returns names of all recognized packages

sim_pkg loaded [<pattern>]
  Returns list of packages currently in use

sim_pkg unloaded [<pattern>]
  Returns list of packages not in use

sim_pkg require <pkgname>
  Assumes mode == -2.
  Loads specified package. The <pkgname> should be unqualified (i.e.
  without leading Sim_) and is expected to be installed in an equally
  named directory which must be a subdirectory of a directory which is
  searcable by way of the ’sim_path’ variable. Package dependencies
  are resolved automatically at load time.
```

F.13 sim_rep

```
sim_rep path
  Returns path user for reports. This is based on the value of the
  ’sim_repdir’ variable.

sim_rep who <pattern>
  Returns names of all opened reports with names matching
  pattern (glob-style). To check if a specific report exists, do
  if {[sim_rep who <repname>] == <repname>} {...}

sim_rep mrk <pattern>
  Returns names of all known report marks with names matching
  pattern (glob-style)

sim_rep mrkinit <mrk>
  Initializes a report mark. Any existing associations with reports
  is cleared.

sim_rep mrkbeg <mrk> <rep> [-offs <inc>]
  Initiates a new associates of mark <mrk> with report <rep>. Any existing
  associtions for the mark is cleared. The new association starts at the
  current line of <rep>, possibly adjusted by and offset of <inc> lines.

sim_rep mrkend <mrk> [-offs <inc>]
  Ends an association started by ’sim_rep mrkbeg’. The association ends at
  the current line (of the associated report), pussibly adjusted by an
  offset of <inc> lines.

sim_rep open <rep-list> [-mode w | a]
```

```
  Opens a report with specified mode. If mode is ’w’ (default) the report
  file is initially empty. Mode ’a’ means append so that any existing
  entries in the report file will be retained.

sim_rep close <rep-list>
  Close a report. Note that the report file is NOT removed, hence a
  report can subsequenlty be opened in append mode. Any existing
  report-windows are remowed when a report closed.

sim_rep reset <rep-list>
  Clears an opened report.

sim_rep write <rep-list> <str>
  Write string to specified report. If a associated report window exists,
  the newly written string will appear correspondingly.

sim_rep won <rep-list> [-width <width> -height <height>]
  Opens report-windows for specified reports. The
  geometry of the window (in text units) can optionally be specified.

sim_rep woff <rep-list>
  Close report windows
```

F.14 sim_run

```
sim_run setup
  Assumes mode == {-1, 0}
  Prepare for run by initializing default class parameters and object
  parameters.

sim_run start > | »
  Assumes mode == 1.
  Start a simulation run with specified speed.

sim_run go > | »
  Assumes mode == 2
  Continues a breaked simulation run.

sim_run advance > | »
  Combines start and go according to current mode.

sim_run break
  Assume mode == 2c
  Breaks an ongoing continuous simulation run.

sim_run done
  Assumes mode == 2
  Stops a simulation run by scheduling SysDone immediately. The mode
  designation is retained.

sim_run stop
  Assumes mode == 2.
  Stops a simulation by immediately returning to mode == 1

sim_run reset
  Assumes mode == 1.
  Resets a prepared simulation run. To start a new run, ’sim_run setup’
  must be performed
```

F.15 sim_sched

```
sim_sched oprint
  Prints contents of global scheduler queue to stdout in ordered format.

sim_sched iprint
  Prints contents of the global scheduler queue to stdout in a format
  according to the internal datastructure used (currently pairing-heap)

sim_sched <astats | rstats> [-<flag>]...
```

```
  Returns statistics, absolute or relative, on the global schduler
  queue. Computation of absolute statistics is based on the time
  difference between an entry and the _frontmost_ entry. Relative statistics
  is based on the time differences between neighbouring entries in the queue.
  Note that the statistics is computed when this command is invoked.
  The information returned is according to the following flags.
  All flags are returned if no flags are specified.
    -#pend      Number of pending objects (due at current time)
    -#queue     Number of future scheduled objects
    -#scan      The number of entries scanned to find the
                _first_ entry which is due next.
    -max        Maximum value for computed statistics
    -mean       Mean for computed statustics
    -distr [<binlist>] Distribution for computed statistics. An optional
                argument is parsed as the bins for which the frequency
                distribution is computed. If this argument is missing ten
                equally spaced bins covering the complete range is used.

sim_sched pstats [-<flag>]...
  Returns profiling statistics about the scheduler queue, if
  available. It will be available if the core is compiled with the
  QPROF symbol defined. Profiling statistics are different from
  absolute and relative statsitics since it is updated at every
  operation on the queue. The ordinary statistics are computed only
  when the command is invoked. The information returned is according
  to the following flags. All flags are returned if no flags are
  specified.
    -del        Average number entries removed for each new scheduled time.
                This reflects the concurreny of the system.
    -queue      Average number in queue after each delete operation
    -scan       Average number of entries scanned per retrieved entry.
```

F.16 sim_source

```
sim_source <file>
  Sources and evaluates the specified script file. The filename is either a
  complete path-name or a name relative to one of the directories in the
  'sim_path' variable. The file is sourced at the global level, and
  'sim_srcdir' is set to the directory component of the file location.
```

F.17 sim_speed

```
sim_speed [slow | semi | quick]
 Reports or sets current simulation speed
```

F.18 sim_stats

```
sim_stats assign <objid> -<mesnam> <statsid-list> ...
  Assigns statistics to specified measure(s) for the given alive
  object. Note that more than one statistics can be assigned to the
  same measure. The statsid-list must hold numerical tags as returned
  by the 'sim_new' command. If the particular designation 0 is used
  for <statsid-list>, any previous assignments are removed.

sim_stats assigned <objid>
  Returns which statistics are assigned to the measures for the
  specified alive object. An assignment of 0 means that no statistics
  are assigned.

sim_stats read <statsid>
  Returns the reading of the specified statistics. The format and
  semantic of the returned information depends on the features of the
  actual statistics class.

sim_stats reset <statsid> [<value>]
```

```
  Reset the specified statistics. The semantics of a reset depends on
  the features of the actual class. If a value is supplied the statistics is
  reset to that value.
```

F.19 sim_xpar

```
sim_xpar <class> | <obj>
  Displays a modal X-dialogbox facilitating setting of parameters.
  All non-significant parameters are inactive and their default
  values displayed.
```

F.20 sim_xvpar

```
sim_xvpar <class> | <obj>
  Displays a X-widow showing the current parameters. The window
  is inactive, hence multiple such windows may exists simultaneously.
```

Appendix G

Core variables

Below is the documentation of user significant global variables that are introduced by the `core` package.

G.1 `sim_batch`

```
sim_batch
  Is a variable telling whether gmsim is started in batch mode. In
  batch mode standard error and standard ouput are redirected to files
  'stderr' and 'stdout' in the current report dierctory. The user is
  responsible for setting this variable prior to startup. After
  startup the variable should not be modified.
```

G.2 `sim_path`

```
sim_path
  Is a variable holding a list of directories. For each directory listed
  the immediate sub-directories are searched for installed packages.
  The user can make new packages known to the system by setting the path
  variable appropriately prior to startup. The path variable is also used
  when searching for script files in response to the 'sim_source' command.
```

G.3 `sim_repdir`

```
sim_repdir
  Is a variable specifying the directory used for reporting. This variable
  should be set prior to startup and should not be modified subsequently.
```

G.4 `sim_rephist`

```
sim_rephist
  Is a variable specifying how many lines to save for report windows.
```

G.5 `sim_script`

```
sim_script
  This variable is expected to be set prior to startup and should contain a
  list of names scriptfiles to use a application scripts.
```

G.6 sim_srcdir

```
sim_srcdir
  This variable is set by the ’sim_source’ command to the directory location
  of the file which is currently sourced. In this way sourced scripts can
  determine their own location by considering this variable.
```

G.7 sim_wdisp

```
sim_wdisp
  This array variable should be set prior to startup in order to indicate
  which windows of the graphical user interface to display initially. Currently
  two windows with keys ’curr’ and ’queue’ are supported. The former is a
  window showing the status of the current object. The latter window shows
  statistics concering the scheduler queue.
```

Bibliography

[1] BANKS, J., AND CARSON, J. *Discrete-Event System Simulation*. Prentice-Hall, 1984.

[2] BRATLEY, P., FOX, B., AND SCHRAGE, L. *A Guide to Simulation*. Springer-Verlag, 1987.

[3] BROWN, R. Calendar queues: A fast O(1) priority queue implementation for the simulation event set problem. *Communications of the ACM 31*, 10 (1988), 1220–1227.

[4] CINLAR, E. *Introduction to Stochastic Processes*. Prentice-Hall, Inc., 1975.

[5] COX, D., AND MILLER, H. *The Theory of Stochastic Processes*. John Wiley and Sons, 1965.

[6] FISHMAN, G. *Concepts and Methods in Discrete Digital Simulation*. John Wiley and Sons, 1973.

[7] FOX, B., AND GLYNN, P. Estimating time averages via randomly-spaced observations. *SIAM Journal on Applied Mathematics 47*, 1 (1987), 186–200.

[8] FREDMAN, M., SEDGEWICK, R., SLEATOR, D., AND TARJAN, R. The pairing heap: A new form of self-adjusting heap. *Algorithmica 1* (1986), 111–129.

[9] GLYNN, P. On the role of generalized semi-Markov processes in simulation output analysis. In *Proc. of the 1983 Winter Simulation Conference* (1983), pp. 38–42.

[10] GLYNN, P. A GSMP formalism for discrete event systems. *Proc. of the IEEE 77*, 1 (1989), 14–23.

[11] GLYNN, P. Special issue: Generalized semi-Markov processes. *Discrete Event Dynamic Systems: Theory and Applications 6*, 1 (1996).

[12] GLYNN, P., AND IGLEHART, D. Simulation methods for queues: An overview. *Queuing Systems: Theory and Applications 3* (1988), 221–256.

[13] HAAS, P., AND SHEDLER, G. Regenerative generalized semi-Markov processes. *Communications in Statistics - Stochastic Models 3*, 3 (1987), 409–438.

[14] HO, Y. Performance evaluation and pertubation analysis of discrete event dynamical systems. *IEEE Transaction on Automatic Control 32*, 7 (1987), 563–572.

[15] HO, Y. Scanning the issue: Dynamics of discrete event systems. *Proc. of the IEEE 77*, 1 (1989), 3–6.

[16] JONASSEN, A., AND DAHL, O. Analysis of an algoritm for priority queue administration. *BIT 15* (1975), 409–422.

[17] JONES, D. An empirical comparison of priority-queue and event-set implementations. *Communications of the ACM 29*, 4 (1986), 300–311.

[18] K. CHUNG, J. S., AND REGO, V. A performance comparison of event calendar algorithms: an empirical approach. *Software Practice and Experience 23*, 10 (1993), 1107–1138.

[19] KINGSTON, J. Analysis of tree algorithms for the simulation event list. *Acta Informatica 22* (1985), 15–33.

[20] KINGSTON, J. Analysis of henriksen's algorithm for the simulation event set. *SIAM Journal on Computing 15*, 3 (1986), 887–902.

[21] KLEINROCK, L. *Queuing Systems: Vol. 1 Theory*. Wiley, 1975.

[22] LAW, A., AND KELTON, W. *Simulation Modeling and Analysis*, second ed. McGraw-Hill, 1991.

[23] MCCORMAC, W., AND SARGENT, R. Analysis of future event set algorithms for discrete event simulation. *Communications of the ACM 24*, 12 (1981).

[24] MORSE, P., AND KIMBAL, G. *Methods of Operations Research*. MIT press and Wiley, 1951.

[25] NIKOLOPOULOS, S., AND MACLEOD, R. An experimental analysis of event set algorithms for discrete event simulation. *Microprocessing and Microprogramming 36* (1992), 71–81.

[26] NILSEN, F. Efficient flit-level simulation. In *Proc. of the 1997 Summer Computer Simulation Conference* (Arlington, VA, July 1997), pp. 79–84.

[27] OUSTERHOUT, J. *Tcl and the Tk Toolkit*. Addison-Wesley Publishing Company, 1994.

[28] RIPLEY, B. *Stochastic Simulation*. John Wiley and Sons, 1987.

[29] SARGENT, R. Verifying and validating simulation models. In *Proc. of the 1996 Winter Simulation Conference* (1996), pp. 55–64.

[30] SCHASSBERGER, R. Insensitivity of steady-state distributions of generalized semi-Markov processes: Part I. *The Annals of Probabilbity 5*, 1 (1977), 87–99.

[31] SCHASSBERGER, R. Insensitivity of steady-state distributions of generalized semi-Markov processes: Part II. *The Annals of Probabilbity 6*, 1 (1978), 85–93.

[32] SCHASSBERGER, R. Insensitivity of steady-state distributions of generalized semi-Markov processes with speeds. *Advances in Applied Probabilbity 10* (1978), 836–851.

[33] SEDGEWICK, R. *Algorithms*. Addison-Wesley Pub. company, Inc., 1983.

[34] SEINDAL, R. *The GNU m4 macro processor*. The GNU Project, the Free Software Foundation (FSF). <http://www.fsf.org>.

[35] SHEDLER, G. *Regeneration and Networks of Queues*. Springer-Verlag, 1987.

[36] SHEDLER, G. *Regenerative Stochastic Simulation*. Academic Press, Inc., 1993.

[37] SLEATOR, D., AND TARJAN, R. Self-adjusting binary search trees. *Journal of the ACM 32*, 3 (1985), 652–686.

[38] STROUSTRUP, B. *The C++ Programming Language*, second ed. Addison-Wesley, 1991.

[39] VAUCHER, J., AND DUVAL, P. A comparison of simulation event list algorithms. *Communications of the ACM 18*, 4 (1975), 223–230.