Proceedings of the 1998 Winter Simulation Conference
D.J. Medeiros, E.F. Watson, J.S. Carson and M.S. Manivannan, eds.

GMSim: A TOOL FOR COMPOSITIONAL GSMP MODELING

Frode B. Nilsen

Telenor RD
P.O. Box 83
N-2007 Kjeller, NORWAY

ABSTRACT

The development of a discrete-event simulation tool, called GMSim, based on the generalized semi-Markov process (GSMP) formalism is described. The GSMP representation comprises both analysis and simulation in a unified framework. This paper focuses on the simulation aspect and how to deal with a combinatorially exploding state space. A compositional GSMP modeling methodology is proposed, which in turn is combined with an object-oriented programming approach.

A key feature of the resulting tool is the close resemblance with the underlying mathematical structure. This facilitates coherent modeling and also an efficient implementation. The tool is completely generic and extendible by Tcl script programming. Application specific components are developed by C++ programming in combination with M4 macro processing.

1 INTRODUCTION

The generalized semi-Markov process (GSMP) formalism gained interest about ten years ago as a convenient way to describe the dynamics found in stochastic discrete-event systems (Glynn 1989, Shedler 1993, Haas and Shedler 1987, Glynn 1996). Unlike most approaches, the GSMP formulation facilitates modeling and reasoning within a *common* framework. The description is at the same time both a precise mathematical setting for analysis and a discrete-event simulation algorithm.

The GSMP framework does *not* aim at closed-form solutions. Quantitative results must be obtained by simulation but a flavor of qualitative theory *can* be established (Glynn 1989). The latter is the reason for applying the framework in the first place. The key point is that theoretically sound and computationally efficient methods for estimation and experimental design are readily available from the GSMP view (Glynn 1983, Glynn and Iglehart 1988).

For simple systems GSMP modeling is straight-forward and the implementation of a corresponding simulation model follows almost immediately. However, as the number of events and components in the state description grows, combinatorial explosion quickly arises. Intractability follows from the fact that the state/event combinations to consider become too numerous to handle.

The contribution of this paper is first the development of a compositional GSMP modeling technique. As always, decomposition is the solution to make complex models tractable. The second contribution is how this can be combined with an object-oriented programming approach for implementation of simulation models. The overall idea is illustrated in figure 1 and the resulting simulation tool is

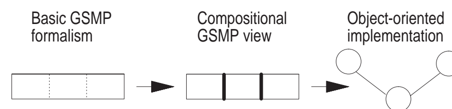

Figure 1: The Idea Behind the GMSim Development

called GMSim (Nilsen 1998). In addition to the theoretical foundation, the paper discusses the implementation of GMSim and how the underlying mathematical structure is exposed to the programmer.

To the knowledge of the author compositional GSMP modeling has not previously been addressed. Neither are we aware of any available simulation tool that directly reflects the GSMP view in an object-oriented environment. We conjecture that the preciseness of the GSMP foundation leads to a well-structured and hence efficient implementation of GMSim itself. The tool facilitates compositional develoment of simulation models but preserves a coherent view.

Otherwise, 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 (Strostrup 1991) augmented by a set of M4 macros (Seindal). The tool is completely generic and application specific components are incorporated by run-time linking. Flexibility is obtained by using the Tcl/Tk script environment (Ousterhout 1994) for simulation control. The tool can be extended in arbitrary ways by script programming. Scheduling in GMSim is based on two-level strategy where a pairing-heap (Fredman et al. 1986) is used for the global queue.

GMSim is based on standard SW components and is released with the source code under the GNU General Public Licence terms. Hence, it is an open-ended tool suitable for research on simulation methodology. It has successfully be used (Nilsen 1997) for studying the performance of wormhole-switched (Ni and McKinley 1993) communication systems.

Note that only a subset of the features in GMSim are discussed in this paper. The full documentation of the tool is available in (Nilsen 1998).

1.1 Organization

The rest of this paper is organized as follows. A summary of the basic GSMP formalism is given in section 2. Section 3 describes the proposed technique to accomplish compositional GSMP modeling. Together these two sections provide the theoretical foundation for the GMSim development.

Section 4 gives an overview of the GMSim tool and its features. This is followed by an illustrative $M/M/1$ queuing example in section 5. An outline of the the C++ programming interface is provided in section 6. Section 7 describes the scheduling algorithm used by GMSim before the paper is concluded in section 8.

2 THE BASIC GSMP FORMALISM

A generalized semi-Markov process (GSMP) 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 (Glynn 1989, Glynn 1983, Glynn and Iglehart 1988) 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 are allowed. The former is based on (Haas and Shedler 1988) and the latter on (Shedler 1993).

Formal definition of a GSMP is in terms of an embedded Markov chain $\mathbf{X}_k$ that describes a continuous-time process $\mathbf{S}(t) \in \mathcal{S}$ at successive epochs of state transition. A one-dimensional illustration is provided in figure 2. Note that $\mathcal{S}$ signifies an application specific

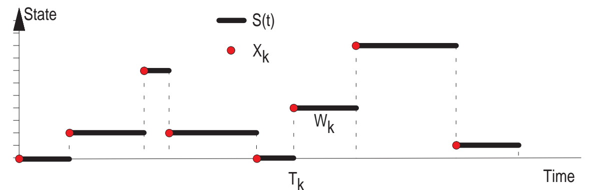

Figure 2: One-dimensional Illustration of a Generalized Semi-Markov Process.

state space which is assumed to be finite or countable. A GSMP process is multi-dimensional in the general case, hence the vector notation.

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 (Cox and Miller 1965) often used to obtain Markov behavior in stochastic models. As always, Markov behavior simplifies analysis. A Markov-renewal condition (Cinlar 1975) is in turn imposed on the compound chain $(\mathbf{X}_k, W_k)$. In addition time-homogeneity (Cinlar 1975, Glynn 1989) is assumed. The details are beyond the scope of this paper 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 (Haas and Shedler 1988) by the joint probability distribution function

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

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 $c_i' \in [0, a_i]$.

The vector $\mathbf{c}^*$ in equation (1) refers to the 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 (1) suggesting that independence is at play. This contributes to the analyticity of a GSMP.

3 COMPOSITIONAL GSMP MODELING

The basic GSMP formulation is tractable for simple applications. As the number of components in the state description and the number of events grow, combinatorial explosion quickly arises. This means that the number of state/event combinations to consider become too numerous to handle. As always, the solution is to decompose the problem.

Our development of a compositional GSMP view 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, hence $\mathcal{S} = \mathcal{S}_a \times \mathcal{S}_b \times \mathcal{S}_c$ and 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. The set of triggering events is decomposed in the same way $\mathcal{D}^* = \mathcal{D}_a^* \cup \mathcal{D}_b^* \cup \mathcal{D}_c^*$ where $\mathcal{D}_j^* \subseteq \mathcal{E}_j$ 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 certain independence restrictions are imposed. Our interest is to obtain 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 3.

The underlying idea is to support an object-oriented

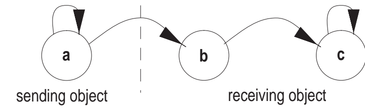

Figure 3: The Idea of a Compositional GSMP View.

programming approach. Component a and c take the role of objects. The self-point arrows suggests that each object have self-driving capabilities. Further, component a acts as 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 introduced in section 4. 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 requirements. The probability density functions $p(\cdot; \mathbf{s}, \mathcal{D}^*, \mathbf{c}^*)$ are separable in the sense that $p(\mathbf{s}'; \mathbf{s}, \mathcal{D}^*, \mathbf{c}^*) = p(\mathbf{s}_a'; \mathbf{s}_a, \mathcal{D}_a^*, \mathbf{c}_a^*) \cdot p(\mathbf{s}_b'; \mathbf{s}_a, \mathcal{D}_a^*, \mathbf{c}_a^*) \cdot p(\mathbf{s}_c'; \mathbf{s}_{b,c}, \mathcal{D}_{b,c}^*, \mathbf{c}_c^*)$. Further, New events are generated component-wise according to $\mathcal{N} = \mathcal{N}_a(\mathbf{s}_a'; \mathbf{s}_a, \mathcal{D}_a^*, \mathbf{c}_a^*) \cup \mathcal{N}_b(\mathbf{s}_a'; \mathbf{s}_a, \mathcal{D}_a^*, \mathbf{c}_a^*) \cup \mathcal{N}_c(\mathbf{s}_{b,c}'; \mathbf{s}_{b,c}, \mathcal{D}_{b,c}^*, \mathbf{c}_c^*)$. Likewise, the decision about which old events to retain are made component-wise according to $\mathcal{O} = \mathcal{O}_a(\mathbf{s}_a'; \mathbf{s}_a, \mathcal{D}_a^*, \mathbf{c}_a^*) \cup \mathcal{O}_b(\mathbf{s}_a'; \mathbf{s}_a, \mathcal{D}_a^*, \mathbf{c}_a^*) \cup \mathcal{O}_c(\mathbf{s}_{b,c}'; \mathbf{s}_{b,c}, \mathcal{D}_{b,c}^*, \mathbf{c}_c^*)$. Finally, new events are scheduled according to component-wise probability distribution functions $F_a(\cdot; \mathbf{s}_a', e_i, \mathbf{s}_a, \mathcal{D}_a^*)$, $F_b(\cdot; \mathbf{s}_b', e_i, \mathbf{s}_a, \mathcal{D}_a^*)$ and $F_c(\cdot; \mathbf{s}_{b,c}', e_i, \mathbf{s}_{b,c}, \mathcal{D}_{b,c}^*)$.

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

4 GMSim OVERVIEW

The GMSim tool is structured as shown in figure 4 and consists of a number of packages to be loaded into a

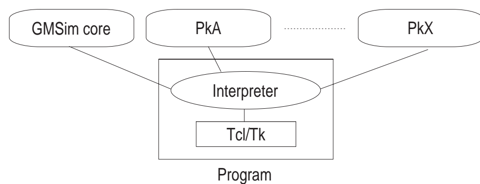

Figure 4: The Package Structure of GMSim

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.

The specific 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`. The full set of commands is documented in (Nilsen 1998).

The new script commands are used to build and manage simulation models. A model comprises items instantiated from various prototype classes. Since the class concept in GMSim builds directly on C++ classes, the full power of inheritance and polymorphism characteristic for object-oriented programming is available.

An item that participate in the simulation is called an alive object. A simulation model can also comprise other kinds of objects like dead objects, configuration objects and statistics (Nilsen 1998). Due to space limitations these features of GMSim are not discussed in this paper.

An alive object corresponds to a component in the compositional GSMP formulation discussed in section 3. The object must be instantiated from a class which has a piece of behavioral code written according to the compositional GSMP view. Connections between objects are formed by links and interfaces. This is illustrated in figure 6 for a particular example to be discussed later. Interactions takes place in terms of the connecting links. Objects possess one or more interfaces (shaded) and the links are typed according to the interfaces they conform to. Hence, only compatible objects can be linked.

The GMSim core provides *no* classes. This is left to user-defined packages. Each package is expected to implement additional classes according to the compositional GSMP view. The recognized classes depend on which packages are loaded. Hence, the modeling capabilities of GMSim can be extended in arbitrary ways without recompiling the core.

The operation of GMSim is illustrated in figure 5. It alternates between the binary domain and the script

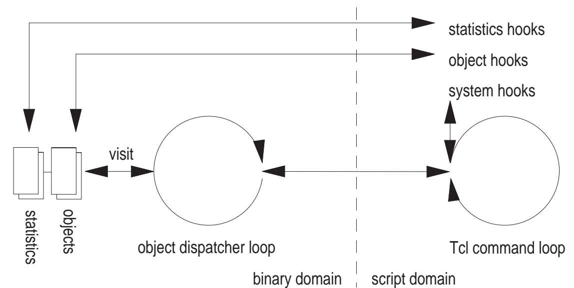

Figure 5: The Operation of GMSim

domain. The command loop is responsible for parsing script commands. 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. For each visited object a piece of behavioral code is executed before control is relinquished. Scheduling is discussed in more detail in section 7.

The package system represents a way to extend the modeling capabilities of GMSim. 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. This is illustrated in figure 5 and the procedures are called script hooks. Further discussion of the script hook facilities are beyond the scope of this paper.

GMSim provides a graphical user interface if the hosting program supports Tk. Figure 7 shows the appearing screen for a particular simulation example. The main window shows a log of responses as script commands are evaluated. 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. 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.

The verboseness features together with the facilities for simulation control are indispensable debugging aids. This is particularly important for complex models with many objects and involved interactions. Note however that the debugging features can be turned off to gain execution speed for batch runs.

Note finally that GMSim includes a system where configuration parameters for objects can be set and read from the script domain.

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

5 EXAMPLE: $M/M/1$ QUEUE

To get an idea of GMSim in action we consider a $M/M/1$ queuing (Kleinrock 1975) example. The queue model comprises three objects as depicted in figure 6. The

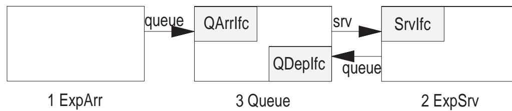

Figure 6: The $M/M/1$ Queuing Model.

involved classes, links and interfaces are also shown. The leftmost object generates arrivals which are immediately fed to the intermediate queuing object. The rightmost object corresponds to the service facility. The source-code for the implementation of this example is available from the GMSim source distribution. Otherwise, the reader is referred to section 6 which describes the programming interface of GMSim.

The script file used to build the model and launch a particular simulation is listed below.

```
# 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
# 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 $arrcnt arrHook
# open statistics report and dumps for objects
sim_rep open arep -mode w
sim_rep won arep -width 30 -height 20
sim_dump won "$arr $queue $srv" -width 40 -height 40
# start run
sim_run start »
# perform 3 state transitions
for {set i 0} {$i < 3} {incr i} {
    sim_run go >
}
```

We leave this without further comments but note that it involves two additional packages called `mm1` and `gstats`. Dump reports for each of the three objects are also prepared.

If this script file is sourced by GMSim the appearing windows will be as shown in figures 7 and 8.

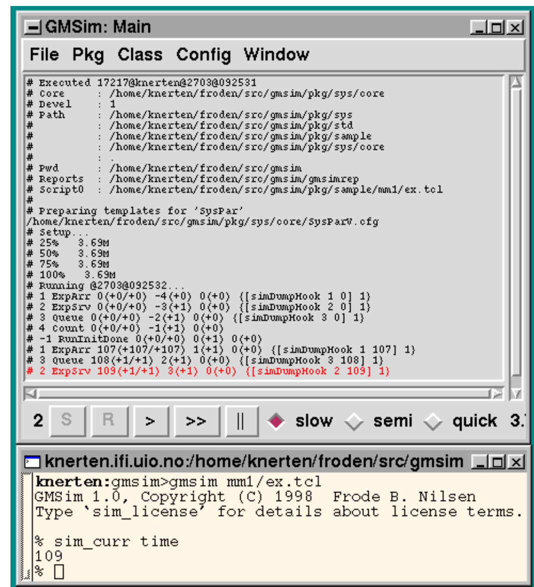

Figure 7: The Main and Command Windows for the $M/M/1$ Queuing Example

Except from menus and buttons, the main window displays the simulation log. The last (dimmed) line shows the response after the most recent simulation step. The three windows in figure 8 show the requested dump reports. The lower window in figure 7 is the command window. As commands are entered here they are passed to the interpreter. This makes GMSim ideal for interactive use. However, the tool can also be used in non-interactive (batch) mode. Then all windows are suppressed.

6 PROGRAMMING

The following subsections give a flavor of how object-oriented programming according to the GSMP view takes place. Macro expansion and the concept of C++ hook functions are central to the discussion. Note that this is completely different from script hooks as discussed in section 4. See (Nilsen 1998) for a complete specification of the programming interface.

Figure 8: The Report Windows for the $M/M/1$ Queuing Example

6.1 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:
  ...
};
```

where the names prefixed by `sim_` are M4 macros. As the macros are expanded code is automatically generated that takes care of all interactions with the GMSim core. Various kinds of blocks are recognized corresponding to the different kinds of prototype classes. In section 6.2 we discuss *the* most important case, alive classes, in more detail. Note that as the block-opening macro `sim_xxx` is expanded, the actual class inherits a common hierarchy of base classes pertinent to GMSim. This is invisible to the programmer.

The `sim_use` macro macro is always used to specify inheritance. 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 member function declarations.

For each of the block constructs there is a set of 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. All instances are called in response to a hook invocation.

6.2 Alive Classes

An alive class is declared by the construct

```
sim_vaclass(name) sim_use(...) {
  sim_events(...);
  sim_links(...);
sim_data:
  ...
sim_hooks:
  <std. hooks>
  void nextState (void);
  Sim_UsrTime nextOccur (int ev);
  void verbose(ostream &os);
sim_body:
  ...
};
```

Emerging links are specified by the `sim_links` macro. The type designation of a link must correspond to an interface declared elsewhere. In addition to specifying ordinary inheritance, the `sim_use` macro is used to declare that the class conforms to a particular interface. Only links with a conforming type can connect to an instantiated object.

In accordance with the GSMP view a number of events can be defined for the class by the `sim_events` macro. The following event-set identifiers are also introduced in the scope of the class: `trigEvs`, `actEvs`, `oldEvs`, and `newEvs`. These sets are used to express the behavior at a state transition according to the GSMP formulation in section 2.

A state description comprises ordinary C++ variables in the `sim_data` section. The `nextState` hook is responsible for maintaining the state. The hook is invoked at every state transition and is *the* most important hook as it implements the behavioral model for a class. This includes handling of the event sets `oldEvs` and `newEvs`. By default, the assignment `oldEvs = actEvs` is 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 `verbose` hook can be used to extend the verboseness when dumps are prepared. It is a good habit

to always supply such a hook since it is often of great help in tracking programming errors.

Otherwise, there is a number of standard hooks for each class (not shown) that can be used to set and check configuration parameters, and initialize and clean the object to a known state.

7 SCHEDULING

Every alive object is responsible for proper scheduling of its own events. 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 employs a two-level scheduling strategy. This is efficient since the number of entries N in the global queue is reduced.

There are several ways to implement the global scheduler queue (McCormac and Sargent 1981, Jones 1986, Chung et al. 1993). The methods can be classified depending on whether they use time-mapping (Kingston 1986, Brown 1988) or maintain a balanced tree structure (Kingston 1985, Sleator and Tarjan 1985). The former class employs the principle of hashing. It has the potential of performing a queuing operation [2] in $O(1)$ time, thus being independent of queue size N. Unfortunately, this works well only if N and the scheduling distribution does not vary too much during the course of a simulation.

The tree based methods are more robust to dynamic variations. The provision is that the tree is balanced so as to keep a queuing operations bounded by $O(\log N)$. Since N can often be quite large in complex simulations, logarithmic behavior is essential. One way to achieve 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.

In GMSim a tree based pairing heap (Fredman et al. 1986) 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 variations in N and also in the scheduling distribution. In the amortized sense a queuing 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 probably 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.

[2] This includes both an insertion and a removal.

8 CONCLUDING REMARKS

The contribution of this paper has been the development of a compositional GSMP view and how this can be combined with an object-oriented programming approach. The proposed methodology deals with combinatorial exploding state space which is otherwise characteristical for complex systems.

We have restricted attention to the simulation aspect of the GSMP framework but it is important to keep in mind that the reason for applying the mathematical description in the first place is the flavor of qualitative theory that can be established. E.g. asynchronous sampling (Bratley et al. 1987, Fox and Glynn 1987), which is generally considered to be efficient, follows easily from the GSMP view (Glynn 1988).

Since the systematic and well-structured GSMP formulation is directly reflected by GMSim, we argue that the tool is both consistent and efficient. Combined with the debugging features, the run-time linking property and the scripting facilities, we think that GMSim represents a versatile and convenient simulation tool. The fact that it is distributed with the source code also makes it suitable for research on simulation methodology.

Even if we strongly believe that GMSim is efficient with respect to execution time, this remains to be properly documented. One direction for future research is to study the performance of GMSim compared to other tools.

ACKNOWLEDGMENTS

This work is mainly supported by grant no. 100722/410 from the Norwegian Research Council. Additional funds have been provided by Telenor RD.

REFERENCES

Bratley, P., Fox, B.L., and Schrage, L.E. 1987. *A Guide to Simulation*. Springer-Verlag.

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

Chung, K., Sang, J., and Rego, V. 1993. A performance comparison of event calendar algorithms: an empirical approach. *Software Practice and Experience*, 23(10):1107–1138.

Cinlar, E. 1975. *Introduction to Stochastic Processes*. Prentice-Hall, Inc.

McCormac, W.M. and Sargent, R.G. 1981. Analysis of future event set algorithms for discrete event simulation. *Communications of the ACM*, 24(12).

Cox, D.R. and Miller, H.D. 1965. *The Theory of Stochastic Processes*. John Wiley and Sons.

Fox, B.L. and Glynn, P.W. 1987. Estimating time averages via randomly-spaced observations. *SIAM Journal on Applied Mathematics*, 47(1):186–200.

Fredman, M.L., Sedgewick, R., Sleator, D.D., and Tarjan, R.E. 1986. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1:111–129.

Glynn, P.W. 1983. On the role of generalized semi-Markov processes in simulation output analysis. In *Proc. of the 1983 Winter Simulation Conference*, pages 38–42.

Glynn, P.W. and Iglehart, D.L. 1988. Simulation methods for queues: An overview. *Queuing Systems: Theory and Applications*, 3:221–256.

Glynn, P.W. 1989. A GSMP formalism for discrete event systems. *Proc. of the IEEE*, 77(1):14–23.

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

Haas, P.J. and Shedler, G.S. 1987. Regenerative generalized semi-Markov processes. *Communications in Statistics - Stochastic Models*, 3(3):409–438.

Jones, D.W. 1986. An empirical comparison of priority-queue and event-set implementations. *Communications of the ACM*, 29(4):300–311.

Kingston, J.H. 1985. Analysis of tree algorithms for the simulation event list. *Acta Informatica*, 22:15–33.

Kingston, J.H. 1986. Analysis of henriksen's algorithm for the simulation event set. *SIAM Journal on Computing*, 15(3):887–902.

Kleinrock, L. 1975. *Queuing Systems: Vol. 1 Theory*. Wiley.

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

Ni, L.M. and McKinley, P.K. 1993. A survey of wormhole routing techniques in direct networks. *IEEE Computer*, 26(2):62–76.

Nilsen, F.B. 1997. Efficient flit-level simulation. In *Proc. of the 1997 Summer Computer Simulation Conference*, pages 79–84, Arlington, VA, , July 1997.

Nilsen, F.B. 1998. GMSim: A generalized semi-Markov simulation environment. Research Report 258, Dept. of Informatics, Univ. of Oslo, Norway, April 1998. <http://www.ifi.uio.no/˜froden/gmsim>.

Ousterhout, J.K. 1994. *Tcl and the Tk Toolkit*. Addison-Wesley Publishing Company.

Shedler, G.S. 1993. *Regenerative Stochastic Simulation*. Academic Press, Inc.

Sleator, D.D. and Tarjan, R.E. 1985. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686.

Stroustrup, B. 1991. *The C++ programming language*. Addison-Wesley, second edition.

AUTHOR BIOGRAPHIES

FRODE B. NILSEN is currently employed as a Research Scientist at Telenor RD (the dominant PNO in Norway) working on strategic development of the access network. Previously he was employed as a Research Assistant at the Dept. of Informatics, Univ. of Oslo. He received a M.S. and Ph.D. from the same place in 1993 and 1998, respectively. His research interests are network architectures, high-speed communication, data communications and methods for performance evaluation.