

Hierarchical Modeling and Distributed Simulation with HIT

Abstract

The Hierarchical Tool HIT is a software tool for hierarchical modeling and performance evaluation of discrete event systems. Besides analytical and numerical solution techniques HIT provides the evaluation of models by sequential simulation. Here we present concepts for optimistic distributed simulation of HIT-models by partitioning them with respect to subhierarchies. The main goals of the concept being presented are the preservation of model structure even in lower levels of the realization (e.g. use of the process view of simulation throughout all levels of abstraction) and distribution transparency on the modeling level (homogeneous model specification for all solution techniques).

1 Introduction

Most techniques for Parallel Discrete Event Simulation (PDES) are based on the *Logical Process (LP) paradigm* [1, 2, 3, 4]. Modelers prefer more application oriented modeling paradigms. The PDES of such application oriented models can be realized by a transformation into models following the LP-paradigm. For instance, the PDES of Petri net and queueing network models is described in [5, 6] and [7, 8, 9]. Petri nets and queueing networks offer (from a modelers point of view) only quite restricted means of abstraction and little support for the structuring of large models. To cope with

models of complex systems modelers demand for more sophisticated facilities for model structuring, like hierarchical model description and modularization. This was one motivation for the development of the Hierarchical Tool HIT [10, 11].

HIT is a software tool for model based performance evaluation of discrete event systems. Its main application areas are computer and communication systems, although HIT is not restricted to them. Models are specified using the high-level language HI-SLANG [12] manually or with support through the graphical user-interface HITGRAPHIC [10], [Reference to be added after reviewing]. HIT offers a broad spectrum of performance measures including *throughput*, *turnaroundtime*, *population*, *occupation* which can be combined with different estimators, e.g. *mean*, *variance*, *bounds*, *frequency intervals*. The computation of measures is requested in an evaluation description. Automatic experiment series with pairs of model and evaluation descriptions are supported.

Here, we present concepts for the PDES of HIT-models. We apply the LP-paradigm, but refine the internal structure of the individual logical processes to preserve and exploit the structure of the HIT-model. Experiments with a prototypical implementation of the distributed HIT simulator performed on a workstation cluster are discussed.

2 HIT-Models

HIT-models are described using the HIT-paradigm. It comprises a separation of *load* and *machine*, modularization, and further structuring by functional refinement and structural inclusion, which both lead to hierarchies.

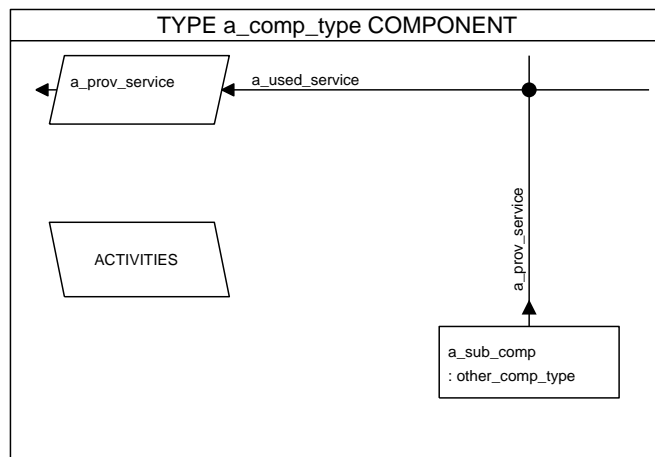


Figure 1: Component with Service and Subcomponent

2.1 Components, Services and Processes

The main building blocks of HIT-models are components. A component (fig. 1) is a complex consisting of a load and a machine part.

The Load is separated into (parameterized) services. A service is the description of a function to be executed. The pattern of actions to realize the function is described by control flow statements including concurrency or by stochastic automata. An action can be the call of a *'used service'*, which is declared in the service, but realized as a service in another component. The set of used services builds up the interface to the machine. Processes, which are service instances in execution, can be created locally in an activities section, or, if the service is provided by the component for external usage, by a call in a higher layer.

The Machine is modularized into (sub)components, which may be structurally included or referenced for (shared) usage. Subcomponents may provide services.

Load and machine are bound by referring each *used service* of the services to a *provided service* of the subcomponents for execution. A model is the component at the top of the hierarchy. It cannot provide services for external usage.

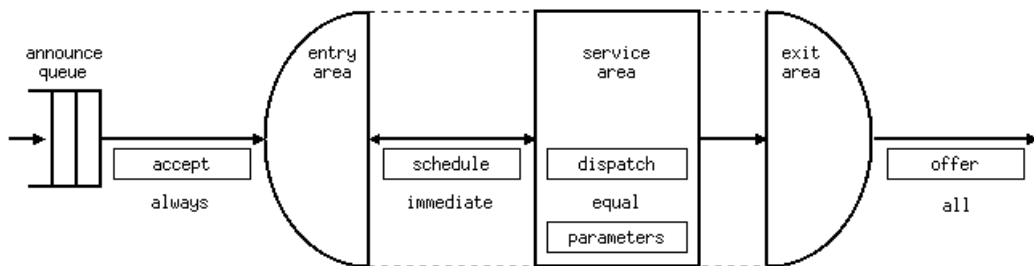


Figure 2: Announce Queue, Component Areas and Component Control Procedures

2.2 Component Control

Components autonomously decide about acceptance and progress of their processes. Whenever a process in a component calls a service of a subcomponent, it places an announcement into the announce queue of that subcomponent. Within components processes are kept in three different areas (fig. 2). They are controlled by the component's control procedures, which perform transitions of processes to, between, and from areas and the announce queue. Components decide when to *accept* an announced service call (*announce queue* \rightarrow *entry area*), when to grant or deny progress to a process (*schedule*: *entry area* \leftrightarrow *service area*), how fast a process is served (*dispatch*) and when a process may leave the component (*offer*: *exit area* \rightarrow *./.*). Only processes currently kept in the service area may progress. A process, for which service has been completed is transferred from the service area to the exit area automatically.

For the control procedure types there exists a set of predefined procedures applicable to all components. For procedure type *accept*, the procedure *always*, that allows all announced processes to enter at once, is the default. The access can be restricted by procedure *limited*, which may cause queueing of announcements. For procedure type *schedule*, procedure *immediate* is the default. It transfers all processes from the entry area into the service area immediately. Other predefined *schedule* procedures select only some processes for service randomly (procedure *random*) or based on the

| type | provided services |
|------------|--|
| server | request(amount) |
| prioserver | request(amount, prio) |
| synchsend | send(mess), receive() |
| nowaitsend | send(mess), receive() |
| semaphor | p(), v() |
| tokenpool | allocate(num), release(num), destroy(num), produce(num) |
| counter | change(amount[],prio) |

Table 1: Standard Component Types

arrival sequence (*fcfs*, *lcfs*, *lcfspr*). The procedure *lcfspr* may also cause preempts from the service area to the entry area. The default *dispatch* procedure *equal* assigns a fixed amount of service capacity to each process in the service area. In case of the *dispatch* procedure *shared* a fixed service capacity is distributed uniformly among the processes. For both kinds of *dispatch* procedures there exist variants with state dependencies. For the control procedure type *offer* the procedure *all*, which offers all processes without restriction, is the default.

Control procedures are activated by events, for instance announcements of processes requesting service or completion of service requests. The execution of control procedures triggers other control procedures of the same component or other components.

2.3 Standard Component Types

Table 1 lists standard component types of HIT. A *server* typically models time consuming tasks. A *prioserver* allows the assignment of different priorities to processes being served.

Synchsend and *nowaitsend* model communication between processes. *Synchsend* is used for rendezvous couplings: if *send* is called before *receive*, the sending process has to wait for the receiving process, and vice versa. *Nowaitsend* uses a message buffer of fixed capacity. Here a sender only blocks if the capacity is exceeded. A receiver has to wait if the buffer is empty.

The well known *semaphor* with its operations p (try/enter) and v (leave) is often used to protect critical sections. A *tokenpool* models general space consumption of processes. If the number of tokens available is not sufficient to grant service to a process trying to allocate or destroy tokens, the process is blocked until the specified number of tokens becomes available by other processes releasing or producing tokens. A *counter* can roughly be described as a multi-dimensional tokenpool.

2.4 Sequential Simulation of HIT-models

Besides analytical and numerical solution techniques based on separable queueing networks or Markovian analysis, HIT provides the evaluation of models by sequential simulation. The sequential simulator of the HIT-system is based on *SIMULA / Class Simulation* [13] and uses a *process view of simulation* [14]. Processes are managed by a central sequencing set (event list). HI-SLANG specifications of HIT-models are transformed into SIMULA programs using Class Simulation including coroutines.

3 Distributed Simulation of HIT Models

Discrete event simulations can be partitioned with respect to (1) space, (2) time, or (3) space and time for parallelization [15]. One of our goals is to exploit the hierarchical structure of HIT-models, so we chose to partition models with respect to space. This is accomplished by distributing the event list among different CPUs and letting each of them manage some subset of the processes. Different PDES protocols such as the conservative CMB-method [2], Jeffersons optimistic Virtual Time paradigm [3], or variants of these protocols [16, 4] can be applied for synchronization between the distributed event lists.

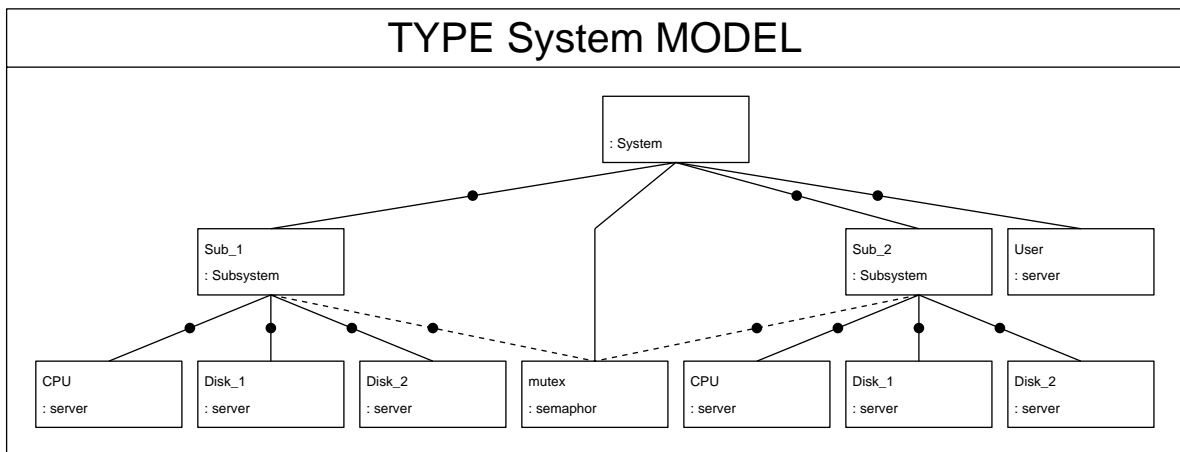


Figure 3: Survey of hierarchical HIT-Model with shared Subcomponent

3.1 The Approach

The basic idea for distributed simulation of HIT-models is simple: components form the granularity of the partitioning. The components of a model are partitioned into subsets, which are assigned to different CPUs. Within a subset simulation is performed sequentially. Events within one subset can be handled locally, independent and parallel to other subsets. Component interaction has been formulated as a message based protocol. Synchronization between different CPUs is only necessary when events cross boundaries of subsets.

In principle the component hierarchy could be partitioned in an arbitrary way. But we believe that *subhierarchies* (subtrees of a component hierarchy like that depicted in fig. 3) form 'canonical' partitions in many cases. Of course, if the model contains shared components that are part of more than one subhierarchy, a decision has to be made, into which subset the shared component shall be put.

Situations in which this approach is believed to potentially yield speedup are those where many concurrent processes exist in a strongly structured system, occasionally competing for resources and exhibiting some *pattern of locality* during their 'visits' to different subsystems. Such situations are

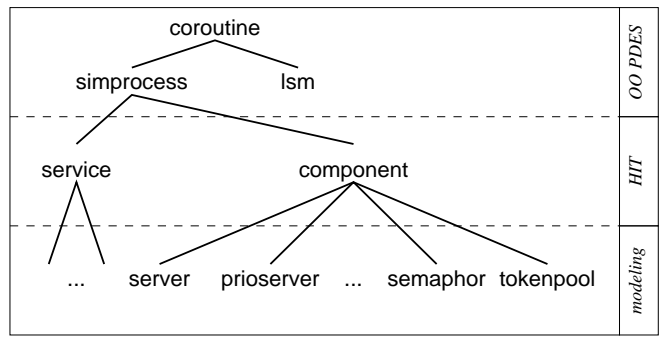


Figure 4: Levels in the Class Hierarchy

recognized in many complex real world systems like e.g. *client server systems*, *mobile communication systems*, and *distributed systems* in general.

A prototype distributed simulator for HIT-models employing Jeffersons Time Warp protocol [3] with Lazy Cancellation [17] has been designed and implemented following this concept [Reference to be added after reviewing]. We chose Time Warp with Lazy Cancellation because it can be modified to master *simultaneous events with causal dependencies*, which are inevitable in simulations of HIT-models or other process oriented models. A subset of HI-SLANG can already be transformed automatically to C++ programs by the prototype.

3.2 The Design

The generated C++ programs reflect the structure of the original HIT-models and are built on the class hierarchy depicted in fig. 4. This class hierarchy identifies three different levels of abstraction: *OOPDES*-, *HIT*- and *modeling*-level.

3.2.1 OOPDES-Level

For the implementation of optimistic methods such as Time Warp it is necessary to checkpoint and rollback processes occasionally. Checkpointing a process means to save the state of the process'

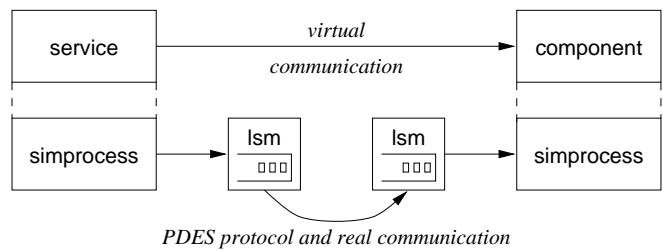


Figure 5: Virtual and Real Communication

variables and thread of execution. The checkpoint operation has to make copies of (1) the object representing the process, (2) the process' stack, and (3) the CPU registers including the program counter. Class attribute variables are included in the object, whereas local variables are contained in the stack. The state of the thread of execution is captured by the stack and program counter. Reestablishing a previously checkpointed state is called a 'rollback'. Class `coroutine` implements SIMULA-like coroutines with checkpoint and rollback operations via light weight processes. It is described in more detail in [Reference to be added after reviewing].

Class `simprocess` is derived from class `coroutine` and implements time-stamped message passing between processes. ('time' is 'virtual simulation time'.) All messages are passed via instances of class `lsm` (local simulation machine). There is exactly one `lsm` object per CPU. Class `lsm` implements event list handling and the PDES protocol for synchronization (fig. 5). The actual communication between `lsm`s is realized via TCP/IP sockets, but is implemented in a modular fashion by a separate class. This allows an easy change of the used communication mechanisms. Another important task of class `lsm` is the management of multiple versions (checkpointed states) of processes and memory management (fossil collection).

The classes `simprocess` and `lsm` together hide all details of the employed PDES protocol, especially the circumstance that processes may be checkpointed and rolled back. Class `coroutine`, class `simprocess` and class `lsm` constitute the *OOPDES-level* design of the distributed simulator. They

provide basic mechanisms for *object oriented parallel discrete event simulation*, independent from the specific simulation, i.e. HIT.

3.2.2 HIT–Level

From class `simprocess` further abstract classes are derived: class `component` and class `service` correspond to the basic building blocks of HIT–models. They are super classes for all component types of HIT–models and form the *HIT–level* design of the distributed simulator. Basic mechanisms specific to HIT, for instance service calls and component control, are realized by them. These mechanisms form the ‘protocol’ of HIT.

3.2.3 Modeling–Level

Finally class `component` and class `service` are refined by classes for the standard component types and user–defined types. This constitutes the *modeling–level* design of the distributed simulator, the functionality of HITs standard types and user–defined component types. The underlying protocol of HIT is not visible at this level.

3.3 Benefits of the Concept

The realization of components and services (standard or user–defined) completely abstracts from the underlying PDES protocol and especially from the circumstance that they may be checkpointed and rolled back by the optimistic protocol. This is a notable benefit of the concept presented. OOPDES–level design (mechanisms of parallel discrete event simulation), HIT–level design (mechanisms of HIT such as component control) and modeling–level design (functionality of components) are separated. This clearly reduces complexity and enables reusability.

Another benefit is the reduction of complexity in the transformation task. The transformation from one high-level language (HI-SLANG) to another (C++ plus the described classes) is by far less complex than the transformation to (e.g.) finite state machines as proposed in [18]. Much of the complexity of the traditional transformation task can be moved to the modular design of appropriate classes. Throughout the concept the process- and object-oriented view of simulation is used. It is a clear advantage if no 'paradigm switch' is necessary.

This also enables the possibility to preserve the structure of the model at the level of realization. The preserved structure can be exploited in various ways, e.g. for incremental state saving, as implemented in the prototype of the distributed HIT-simulator. Here the granularity for incremental state saving is not single state variables (too fine), subhierarchies or components (too coarse), but objects of class `simprocess`: only those activated or modified between consecutive checkpoints are actually saved. We think that this granularity is a reasonable compromise between space- and time-efficiency.

4 Experiments

Various experiments have been carried out to study the influence of (1) *simulation parameters of the PDES protocol* and (2) *model structure* on the performance of the prototypical distributed HIT-simulator. Here we present one experiment that examines the *scaling* of parallel HIT-simulations and the influence of *load asymmetry*.

4.1 Simulation Parameters

Important parameters of the PDES protocol are GVT computation- and checkpoint-frequencies. The Global Virtual Time (GVT) [3] is a measure of the global progress of a parallel simulation. Its most

common use is termination detection and to decide when old checkpointed states can be deleted (fossil collection).

A GVT computation is initiated by an lsm (the *master*) by broadcasting a GVT start message to all other lsms (the *slaves*). The master collects acknowledgements from all slaves and then broadcasts a GVT stop message. Subsequently, the master collects GVT reply messages (containing the local GVT estimates) from all slaves. Finally, the master broadcasts a GVT update message (containing the minimum of all GVT estimates, which serves as lower bound to the actual GVT) to all slaves. (For details and alternative GVT algorithms see [3, 19].)

There is no use in starting GVT computations too often or having more than one of them running simultaneously, thus, we chose the following mechanism of *cycling GVT mastership*: Initially the first lsm is the master, it is the only lsm allowed to initiate a GVT computation. When it has completed a GVT computation, it passes this right to the next lsm in a cyclic manner. The master initiates a GVT computation, if no GVT computation is in progress and at least one of the following conditions holds: (1) it has no more events to simulate (pseudo termination); (2) it runs short in memory and cannot fossil-collect sufficient memory because the GVT estimate is too low; (3) it experienced 50 local increments of model-time.

Pseudo termination may indicate global termination, so an update of GVT is needed to check if the predefined global simulation stop time is already reached. In the second case an improved GVT estimate is needed to fossil-collect more memory. The last case ensures that GVT computations are initiated regularly and the right of mastership keeps cycling. The first two conditions may as well indicate that an lsm has run too far ahead of the other lsms. If this is true, this lsm can only wait for new events, a rollback, or an improved GVT estimate.

In the experiment shown here, fossil collections were executed as soon as improved GVT values

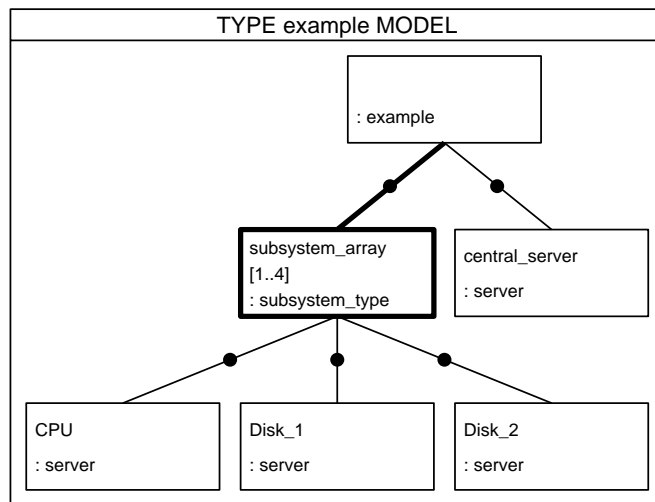


Figure 6: Example Model

became available, because long fossil collections, executed seldomly, caused the lsms to progress less synchronously.

Local checkpoints were performed independently every 5 local increments of model-time. Each lsm was assigned 10 MBytes of main memory. (Recall that there is exactly one lsm per CPU.) These 10 MBytes were no strict limit and lsms were allowed to temporarily use more memory, but lsms violating this limit were forced to perform wait-cycles. The observation, that led to this strategy, is that slow lsms keep the GVT low and faster lsms cannot delete old checkpointed states, thus, are consuming more and more memory. A reasonable way to cope with such situations (beside dynamic load balancing) is to slow-down lsms running too far ahead.

4.2 Model Structure

The model used as benchmark for these experiments consists of a central server, four identical subsystems and 20 processes cycling within and between the subsystems. Each subsystem contains three servers modeling the CPU and two disks of a computer system (fig. 6).

| CPUs | seq. | parallel | | |
|--------------------|------|----------|-------|-------|
| | 1 | 2 | 3 | 4 |
| events [1000] | 1180 | 1370 | 1690 | 1510 |
| overhead [%] | | 16.5 | 43.4 | 28.0 |
| checkpoints | | 30900 | 38300 | 33900 |
| rollbacks | | 880 | 960 | 1520 |
| fossil collections | | 3010 | 4270 | 3540 |
| exec. time [s] | 559 | 482 | 445 | 285 |
| speedup | | 1.16 | 1.26 | 1.96 |

Table 2: Results

One cycle within a subsystem consists of a visit to the CPU and a visit to one of the two disks (with asymmetric probability of 0.25 versus 0.75). A process performs (an average of) 100 cycles (geometrically distributed) within one subsystem before it leaves the subsystem, visits the central server and is routed (with equal probability) to the next subsystem. A simulation is stopped when the specified model-time is reached.

4.3 Results

The model was simulated sequentially (via 1 lsm on 1 processor) and partitioned into two, three and four subsets for parallel simulation on a cluster of up to four workstations¹ connected via a 10 MBit/s ethernet and TCP/IP protocol.

In the two CPUs experiment each CPU was assigned two subsystems, in the four CPUs experiment each CPU was assigned one subsystem. The central server was assigned to one of these CPUs. In the three CPUs experiment one CPU was assigned one subsystem, one CPU was assigned one subsystem and the central server, and one CPU was assigned two subsystems. Each simulation run was repeated five times and averages of the individual results were calculated.

¹Sun SPARCstation 5, 70 MHz FMI, MB86904 CPU, 32 MBytes main memory, SunOS 4.1.3

Table 2 shows some results. The sequential simulation executed 1.18 millions events, the three CPUs parallel simulation took 1.69 millions events. This is an overhead of 43.4% caused by events that were rolled back and reexecuted. These results demonstrate the drastic effect of an asymmetric load.

The number of checkpoints is roughly proportional to the number of events executed in the parallel simulations. The number of rollbacks differs not very much between the two and three CPUs simulation, because the heavy loaded lsm (which simulates two subsystems) slows down the other CPUs in the three CPUs experiment. This slow-down results in an increased number of 4270 fossil collections in our example. The speedup is approximately 1.2 for the two and three CPUs simulations, and nearly two with four CPUs.

Table 3 shows typical observed distribution of CPU times between different tasks. It seems that context switching between light weight processes is very expensive in comparison to event management and event execution. Thus, we expect a considerable improvement if our coroutine implementation could be based on faster context switching, which has to be provided by hardware or operating system software. Event execution, which can be considered the 'productive' part of a simulation, is one third in the sequential case and one quarter in the parallel simulation.

More experiments have been carried out with the prototypical distributed HIT-simulator to examine the influence of GVT computation- and checkpoint-frequencies, partitioning, degree of concurrency and locality, memory limitation, simplification of HIT-mechanisms, hardware and operating system. They are discussed in [Reference to be added after reviewing].

| | seq. | par. |
|----------------------------------|-------------|-------------|
| context switching | 50% | 35% |
| process (event execution) | 33% | 25% |
| lsm (event management) | 17% | 20% |
| checkpoint+rollback+fossil coll. | | 10% |
| communication | | 10% |

Table 3: Typical CPU Time Distribution

5 Summary and Conclusions

A concept for optimistic object-oriented parallel discrete event simulation of HIT-models has been presented. Models are partitioned with respect to subhierarchies, components form the basic granularity.

The realization of modeling- and HIT-level functionality completely abstracts from the underlying PDES protocol. Throughout the concept the process- and object-oriented view of simulation is used. Transformation from HI-SLANG to C++ is by far less complex than transformation to more low-level representations. Much of the complexity of the traditional transformation task can be moved to the modular design of appropriate classes.

Preservation of model structure is possible and can be exploited in various ways. For incremental state saving single processes have been chosen as granularity, not single variables.

A prototype has been implemented to demonstrate the feasibility of the concept. The primary focus has been on concepts and not on optimized performance. Thus, first experimental results are not that epoch making, but reasonable speedups can already be achieved in some cases. We think that the concept presented here may lead to more modular and even efficient parallel simulations.

Acknowledgments

<To be added after reviewing>

References

- [1] K. Chandy and J. Misra, "Asynchronous Distributed Simulation Via a Sequence of Parallel Computations," *CACM*, vol. 24, no. 4, pp. 198–206, 1981.
- [2] J. Misra, "Distributed Discrete-Event Simulation," *Computing Surveys*, vol. 18, no. 1, pp. 39–65, 1986.
- [3] D. Jefferson, "Virtual Time," *ACM Transactions on Programming Languages and Systems*, vol. 7, pp. 404–425, July 1985.
- [4] A. Ferscha, "Parallel and Distributed Simulation of Discrete Event Systems," in [20], ch. 35, pp. 1003–1041.
- [5] A. Ferscha and G. Chiola, "Accelerating the Evaluation of Parallel Program Performance Models Using Distributed Simulation," in *Proc. 7th Int. Conf. on Computer Performance Evaluation*, pp. 231–252, Springer-Verlag, May 1994.
- [6] G. Balbo and G. Chiola, "Stochastic Petri Nets Simulation," in *Proceedings of the 1989 Winter Simulation Conference*, pp. 266–276, 1989.
- [7] D. Nicol, "High Performance Parallelized Discrete Event Simulation of Stochastic Queueing Networks," in *Proceedings of the Winter Simulation Conference*, (San Diego, California), pp. 306–314, 1988.
- [8] D. Nicol, "Conservative Parallel Simulation of Priority Class Queueing Networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, pp. 294–303, May 1992.
- [9] D. Wagner and E. Lazowska, "Parallel Simulation of Queueing Networks: Limitations and Potentials," *ACM SIGMETRICS, Performance Evaluation Review*, vol. 17, no. 1, pp. 146–155, 1989.

- [10] H. Beilner, J. Mäter, and C. Wysocki, "The Hierarchical Evaluation Tool HIT," in Bause and Beilner [21], pp. 6–9, also http://ls4-www.informatik.uni-dortmund.de/QM/hit_kb.ps.
- [11] H. Beilner, J. Mäter, and N. Weißenberg, "Towards a Performance Evaluation Environment: News on HIT," in *Proc. Modelling Techniques and Tools for Computer Performance Evaluation*, (Palma de Mallorca, Spain), 1988.
- [12] M. Büttner, *HI-SLANG Reference Manual for the Hierarchical Evaluation Tool HIT*. Universität Dortmund, Informatik IV, D-44221 Dortmund, 1994.
- [13] G. M. Birtwistle, O. J. Dahl, B. Myrhaug, and K. Nygaard, *SIMULA begin*. Sweden: Lund, 1973.
- [14] W. R. Franta, *The Process View of Simulation*. North-Holland, 1977.
- [15] R. Bagrodia, K. Chandy, and W. Liao, "A Unifying Framework for Distributed Simulation," *ACM Transactions on Modeling and Computer Simulation*, vol. 1, no. 4, pp. 348–385, 1991.
- [16] D. Nicol and R. Fujimoto, "Parallel Simulation Today," *Annals of Operations Research*, no. 53, pp. 249–285, 1994.
- [17] Gafni and Anat, *Space Management and Cancellation Mechanisms for Time Warp*. PhD thesis, Dept. of Computer Science, University of Southern California, 1985.
- [18] R. Pooley, "Object Oriented Discrete Event Simulation Using C++," in [22], (Dept. of Computer Science, University of Edinburgh), September 1995.
- [19] S. Bellenot, "Global Virtual Time Algorithms," in [23] *Proceedings of the SCS Multiconference on Distributed Simulation*, pp. 122–127, 1990.
- [20] A. Y. H. Zomaya, ed., *Parallel & Distributed Computing Handbook*. Series on Computing Engineering, McGraw-Hill, 1996.

- [21] F. Bause and H. Beilner, eds., *Performance Tools - Model Interchange Formats (Performance Tools '95 / MMB '95)*, (D-44221 Dortmund, Germany), Universität Dortmund, Informatik IV, Forschungsbericht-Nr. 581/1995, July 1995.
- [22] M. Paterok, ed., *Performance Tools '95 and MMB '95 — Tutorials*, (Heidelberg, Germany), IBM Scientific Center, IBM Scientific Center, September 1995.
- [23] D. Nicol, ed., *Distributed Simulation, Proceedings of the SCS Multiconference on Distributed Simulation, 17-19 January, 1990, San Diego, California.*, vol. 22, no. 1 of *Simulation Series*. SCS, 1990.