A Discrete-event Simulation Tool for the Analysis of Simultaneous Events

Patrick Peschlow
University of Bonn
Department of Computer Science IV
Roemerstr. 164
53117 Bonn, Germany
peschlow@cs.uni-bonn.de

Peter Martini
University of Bonn
Department of Computer Science IV
Roemerstr. 164
53117 Bonn, Germany
martini@cs.uni-bonn.de

ABSTRACT

Discrete-event simulation is a very popular technique for the performance evaluation of systems, and in widespread use in network simulation tools. It is well known, however, that discrete-event simulation suffers from the problem of simultaneous events: Different execution orders of events with identical timestamps may lead to different simulation results. Current simulation tools apply tie-breaking mechanisms which order simultaneous events for execution. While this is an accepted solution, a legitimate question is: Why should only a single simulation result be selected, and other possible results be ignored?

In this paper, we argue that confidence in simulation results may be increased by analyzing the impact of simultaneous events. We present a branching mechanism which examines different execution orders of simultaneous events, and may be used in conjunction with, or as an alternative to tie-breaking rules. We have developed a new simulation tool, MOOSE, which provides branching mechanisms for both sequential and distributed discrete-event simulation. While MOOSE has originally been developed for network simulation, it is fully usable as a general simulation tool.

1. INTRODUCTION

Simulation is in widespread use for the performance evaluation of computer networks and communication protocols. The most popular technique used in network simulation is *discrete-event simulation* (DES). The underlying concept of discrete-event models is simple: The state of the simulated system is stored as a set of variables, and actions are modeled as events scheduled for execution at discrete points in simulation time. In network simulation, typical events are the beginning and the end of packet transmissions, or the expiration of timers. An important advantage of DES is that standard techniques exist for parallel and distributed simulation on multiprocessor systems or interconnected hosts. Compared to *sequential* DES on a single processor, *parallel and distributed simulation* (PDES) is able to speed up simulation runs and provides more memory. This facilitates the simulation of very large networks and the scalability analysis of communication protocols.

Despite its popularity, discrete-event simulation suffers from the well-known problem of *simultaneous events*: The execution order of two or more events with identical timestamps is unspecified. Different execution orders, however, may lead to different simulation results. Moreover, some execution orders may cause behavior not intended by the system modeler. In previous research, several techniques for handling simultaneous events have been proposed and implemented in simulation tools. Most approaches are based on *tie-breaking rules*, which use priorities to enforce deterministic event execution orders. Tie-breaking rules have two main benefits: First of all, properly assigned event priorities guarantee the reproducibility of simulation runs. Secondly, user-defined priorities may be used to prevent unintended behavior, i.e. execution orders not in the sense of the model. While, especially in distributed simulation, a consistent assignment of priorities is a complex task, adequate solutions for tie-breaking rules have been found. We give a detailed overview of tie-breaking rules in section 2.

Although tie-breaking rules are generally accepted as a solution to the problem of simultaneous events, they have a considerable drawback: Often, a single correct execution order of simultaneous events and thus a single correct simulation result do not exist. We illustrate this with two examples: Assume a completely filled packet buffer, with an enqueue event and a dequeue event scheduled to happen simultaneously. Now, in case the dequeue event is executed first, buffer capacity is freed and the arriving packet may be enqueued. Otherwise, the packet has to be dropped. Both execution orders are correct in the sense of the simulation model, yet they may well lead to different results. This means, however, that selecting a single simulation result by ordering the events according to a tie-breaking rule implicitly labels the other result as irrelevant. As a second example, consider a communication protocol using retransmission timers, e.g. TCP [29]. Whenever a timeout event and the corresponding acknowledgement reception happen simultaneously, the execution order decides whether the packet will be retransmitted or not. This is a typical example of simultaneous events arising due to limited timestamp precision. Note that user-defined priorities are not a solution in these cases, since they would bias results by favoring certain event types, e.g. timeouts.

An alternative to tie-breaking rules is the use of a *branching mechanism*. Branching means examining the different execution orders of simultaneous events, and analyzing their impact on simulation results by splitting the simulation into different branches. The computation of a whole set of simulation results was first proposed in [32], and a first step towards a branching mechanism for sequential DES was taken in [1] by introducing a formal framework for the analysis of simultaneous events. Analyzing simultaneous events may increase confidence in simulation results as well as the simulated system itself, especially in cases where the effects of simultaneous events cannot be estimated in advance.

In this paper, we present a discrete-event simulation tool, MOOSE, which provides branching mechanisms for the analysis of simultaneous events in both sequential, and parallel and distributed DES. Since branching may noticeably increase simulation run-times, we have particularly focused on its efficiency, and also support using combinations of branching and tie-breaking rules. The structure of MOOSE is highly modular, which makes the branching mechanism easily configurable and extensible. While MOOSE was primarily designed for network simulation, it is nevertheless fully usable as a general-purpose simulation tool.

The paper is structured as follows: In section 2, we give an overview of previous research on simultaneous events, and discuss the handling of simultaneous events in network simulation tools. In section 3, we introduce the general branching concept and its implementation for sequential DES. Section 4 describes our design and implementation of branching for parallel and distributed simulation. In section 5, we give an overview of the components of MOOSE and the configuration of the branching mechanism. Section 6 summarizes the paper and outlines directions of possible future work.

2. SIMULTANEOUS EVENTS

In this section, we give an overview of previous research on simultaneous events. In the process, we also discuss the handling of simultaneous events in network simulation tools.

2.1 Tie-breaking rules

In sequential DES, all pending events are stored in a single, global *future event list* (FEL). Events are always executed in increasing timestamp order, and may modify the system state, schedule new or delete existing events. When the FEL contains two or more events with identical timestamps, they are referred to as *simultaneous events*. The most common way of handling simultaneous events is to simply execute them in the order they are returned by the data structure used for the FEL. Since there is only a single thread of execution in sequential DES, this is sufficient to guarantee deterministic event scheduling, and thus reproducibility. Many network simulation tools follow this approach, e.g. JiST/SWANS [14], or J-Sim [12].

Although the simple approach ensures reproducibility, it has an important disadvantage: Depending on the FEL implementation, FIFO behavior is not guaranteed, i.e. simultaneous events are not necessarily executed in the same order they are inserted into the FEL. For example, this may happen with the commonly used binary heap data structure. If FIFO behavior is not guaranteed, however, this may lead to effects not intended by the system modeler. A classic example of unexpected behavior caused by the FEL scheduling policy is described in [27]. Thus, in order to guarantee FIFO behavior, some simulation tools rely on tie-breaking rules with unique IDs as event priorities. All IDs are generated by a global counter and automatically assigned to events when they are created. By sorting events with equal timestamps according to their IDs, the FEL ensures FIFO behavior independent of its implementation. Network simulators which use this technique are e.g. GTNetS [10], and ns-2 [19]. However, a more flexible way of avoiding unintended execution orders are user-defined priorities, which may be assigned at the beginning of the simulation, or stored in a library. Network simulation tools with support for user-defined priorities are NCTUns [18], and the sequential version of GloMoSim [9]. Both tools provide standard priorities which may be used by the application programmer.

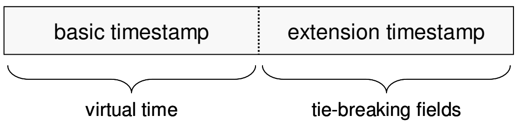


Figure 1: Extended event timestamp.

We summarize that tie-breaking rules are usually kept simple in sequential DES, since reproducibility is inherently guaranteed. The majority of network simulation tools does not provide the user with any means of specifying priorities.

In parallel and distributed simulation, the system state is partitioned into logical processes, which are distributed onto the participating processors or hosts. Each logical process (LP) has its own FEL and is responsible for handling all events related to its part of the state. Other events are forwarded to their destination LPs. Since this creates the possibility of events arriving at LPs out of order, a synchronization mechanism is required to ensure the correctness of the simulation. Commonly, *conservative* or *optimistic* synchronization is used. For a detailed overview of PDES techniques and synchronization mechanisms see [8].

In contrast to sequential DES, reproducibility is not inherently guaranteed in PDES. In two simulation runs with identical setup, simultaneous events may arrive at their destination LP in different orders, e.g. due to network delays or background load. As a consequence, an appropriate tie-breaking mechanism is required in order to ensure reproducibility. Still, many PDES network simulation tools like OMNET++ [20] and PDNS [22] abstain from the use of such tie-breaking rules.

The first challenge with regard to the realization of tie-breaking rules in PDES is that synchronization mechanisms have to respect event priorities, although they only know about event timestamps. For example, with conservative synchronization, an event may only be executed when it is guaranteed that no simultaneous event with a higher priority will be received. With optimistic synchronization, an already executed event has to be rolled back when a simultaneous event with a higher priority arrives. An elegant solution has been found, however, by extending event timestamps with additional bits which encode priorities. This way, synchronization mechanisms do not have to be modified in order to respect event priorities. Figure 1 shows the structure of an extended event timestamp. The *basic timestamp* contains the floating point value representing simulation time, while the *extension timestamp* contains tie-breaking information. See [30] for a recent example.

Although assigning consistent priorities to events, and using the timestamp-encoding scheme, ensures reproducibility for some simulation scenarios, this is not sufficient in the general case. Further mechanisms are required when LPs are able to schedule *zero-delay events* at each other. Zero-delay events have identical scheduling and execution time, and are commonly used to implement query events, or to model advances in simulation time that are too small for the time resolution. For further examples, see [7]. A simulation model contains a *zero-delay cycle*, if two or more LPs are able to send zero-delay events in a cyclic scheme. Zero-delay cycles are a fundamental challenge in PDES: If the simulation model contains a zero-delay cycle, conservative synchronization mechanisms are in danger of deadlock. With optimistic synchronization, zero-delay cycles may catch the simulation in an endless rollback loop [16, 26]. This means that tie-breaking rules do not only have to guarantee reproducibility, but also ensure progress of the simulation.

With regard to zero-delay cycles, research has concentrated on two approaches. The first approach makes restricting assumptions about the occurrence of zero-delay cycles in order to realize tie-breaking rules which are solely based on user-defined priorities. As shown in [13], with some synchronization mechanisms it is necessary that the simulation model does not contain any zero-delay cycle at all. With other synchronization mechanisms, however, it is only required that zero-delay events are not actually sent in a cycle during the simulation. It is also much easier to fulfill the common requirement that the result of a distributed simulation is equal to the result of the corresponding sequential simulation, when the simulation model is free of zero-delay cycles [4]. In case assumptions about zero-delay cycles are not possible, it is necessary to follow a different approach: Supporting zero-delay cycles by including information about the causality of events into priorities. This may be realized with sophisticated tie-breaking rules based on automatically assigned priorities. As an example, the tie-breaking rule in [16] extends event timestamps by a tuple consisting of an age field, as well as an identifier and a message counter of the LP which created the event. In [26], a detailed discussion of automatic tie-breaking rules is given. Furthermore, it is shown that they may be extended by user-defined priorities, as long as the user obeys certain rules. Altogether, this approach constitutes the most complete tie-breaking solution to the reproducibility problem.

In PDES network simulation tools, only simple variants of these approaches have been implemented. The parallel version of GloMoSim, which is based on the ParSec simulation kernel [21], provides two priority values for tie-breaking, which is not sufficient for reproducibility. In SWAN [28], which is based on DaSSF [6], tie-breaking with automatically assigned priorities (process ID, age) may be used optionally. This, however, does neither include user-defined priorities nor guarantee reproducibility in all cases.

In summary, research has produced adequate tie-breaking rules which guarantee reproducibility in PDES. Nevertheless, parallel and distributed network simulation tools only use simple variants of them, if at all.

2.2 Analyzing simultaneous events

The impact of simultaneous events on simulation results was first discussed in [32] in the context of aviation simulation. Later, examples from network simulation were presented in [31], where otherwise identical simulations setups led to noticeably different results when different tie-breaking rules were used. It was concluded that, for a proper evaluation of the simulation model, all execution orders of simultaneous events would have to be taken into account [26].

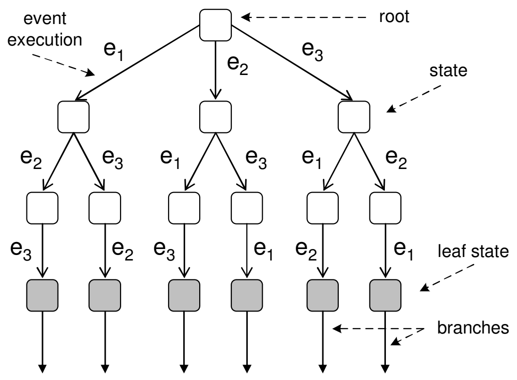


Figure 2: Execution tree for 3 simultaneous events.

First steps towards the analysis of simultaneous events were taken in [1] by establishing a new term of event interaction and a formal framework for the analysis of simultaneous events. The framework contains a mechanism for specifying user knowledge about event interactions, which can be used to identify simultaneous events worth of examination during the simulation. Furthermore, it was proposed to analyze simultaneous events with a branching mechanism if different execution orders may lead to different states. Finally, branching algorithms for sequential DES were outlined.

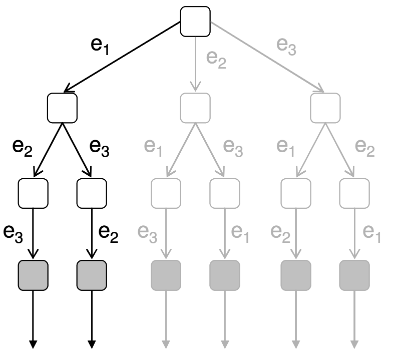
Based on this framework, we have developed a branching mechanism for both sequential DES and PDES. In the following sections, we describe the design and implementation of our branching mechanism, as well as its usage in the simulation tool MOOSE.

3. BRANCHING IN SEQUENTIAL DES

In this section, we first describe the general design of our branching mechanism. Then, we present our implementation for sequential DES.

The possible execution orders of simultaneous events can be visualized as an *execution tree* (cf. figure 2). Nodes represent system states, and transitions conform to event executions. Every path from the root to a leaf state corresponds to a different execution order. The task of a branching mechanism may then be summarized as follows: When simultaneous events are detected, their execution orders shall be analyzed by calculating the leaf states of the execution tree. Then, the simulation shall be continued for all different leaf states. We refer to the resulting simulation runs as *branches*. Every branch is identified by a unique *branch ID*.

In sequential DES, the detection of simultaneous events is simple: It only has to be checked whether the FEL is headed by two or more events with identical timestamps. Accessing the smallest and the second smallest element in the FEL, and comparing their timestamps, is straightforward with the commonly used binary heap implementation. However, if a data structure which bundles events with identical timestamps, e.g. the *enhanced heap* [2], is used for the FEL implementation instead, the overhead of detecting simultaneous event is negligible.



	e_1	e_2	e_3
e_1	N	N	N
e_2	N	N	**P**
e_3	N	**P**	N

APK matrix

Figure 3: Using a-priori-knowledge.

When simultaneous events have been detected, the next step is to calculate the leaf states of the execution tree. In principle, this may be accomplished with standard search algorithms. However, since exploring the execution tree includes saving and restoring possibly large system states, traversing the whole execution tree may get time-consuming when the number of simultaneous events is large. Here, an important observation is that several nodes of the execution tree may be equal. In particular, leaf states may be equal. Thus, it may not be necessary to compute the whole execution tree in order to determine the different leaf states. Depending on the simulation scenario, it is even possible that *all* execution orders lead to one and the same leaf state. Therefore, we use two techniques to keep execution trees small by eliminating unnecessary (or duplicate) computations: *Node comparisons* and *a-priori-knowledge*.

Node comparisons are possible after every transition in the execution tree. Whenever a newly created node is equal to an existing node, the branches are immediately merged together. For the comparisons, each node is represented by a tuple $(\mathsf{E}_{\mathsf{exec}}, \mathsf{E}_{\mathsf{pend}}, \mathsf{E}_{\mathsf{new}}, \mathsf{S})$, where $\mathsf{E}_{\mathsf{exec}}$ is the set of already executed simultaneous events, $\mathsf{E}_{\mathsf{pend}}$ is the set of simultaneous events yet to execute, and $\mathsf{E}_{\mathsf{new}}$ is the set of all events created on the corresponding branch. Together, the three event sets represent the relevant part of the scheduler state, while S represents the current system state, e.g. the set of system state variables. Since the system state may contain a large number of variables, the scheduler states are always compared first. Only when they are equal, the system state variables of the nodes are compared. We implemented node comparisons by storing nodes in buckets according to $\mathsf{E}_{\mathsf{exec}}$. This way, we are able to identify potentially equal nodes immediately. In order to maximize the efficiency of state comparisons, we provide the programmer with means of ordering the system state variables for comparison, so that differing states are identified very fast.

While node comparisons eliminate all duplicate states, they also have two disadvantages: With large system states they may get costly, and they can only be performed *after* the execution of events. Therefore, we use an additional mechanism, a-priori-knowledge (APK), which is much cheaper in terms of computing resources and can already be employed before events are executed. The concept of APK was introduced as part of the framework in [1]. The basic idea is to store programmers' expert knowledge about possible event interactions permanently in a repository, and provide means to access and utilize this knowledge at simulation run time. In our implementation, three types of event interactions may be specified: Two events e_1 and e_2 are *non-interacting* (N), if both execution orders $(\mathsf{e}_1; \mathsf{e}_2)$ and $(\mathsf{e}_2; \mathsf{e}_1)$ lead to *identical* states whenever e_1 and e_2 occur simultaneously. Similarly, with *surely-interacting* (S) events, it is guaranteed that both execution orders always lead to *different* states. Finally, with *possibly-interacting* (P) events, both ways are possible, depending on the circumstances. Event interactions can be formalized as a function $\mathsf{APK} : \mathsf{E} \times \mathsf{E} \rightarrow \{\mathsf{N}, \mathsf{P}, \mathsf{S}\}$, where E is the set of all types of events which may occur in the simulation. During the computation of the execution tree, the APK function is evaluated. Whenever a subset of simultaneous events is found to be non-interacting, only one execution order is calculated. Thus, depending on the number of non-interacting events, many event executions are avoided. Furthermore, since node comparisons are never required for non-interacting and surely-interacting events, they are only performed after the execution of possibly-interacting events.

With regard to the implementation, APK may be utilized in different ways. Our implementation keeps an interaction table of events, which is updated when events are executed. Each simultaneous event keeps counters representing its interaction with the other events. As soon as the counters suggest that an event is non-interacting with the other remaining events, it is executed immediately. Furthermore, we keep APK tables small by setting up only the required APK for each simulation run. This is done automatically during the initialization of the simulation modules at the beginning of the simulation. Altogether, APK usage may often result in a very fast calculation of execution trees, which makes it a powerful tool for analyzing simultaneous events efficiently. Figure 3 shows an example, with the APK function visualized as a symmetric matrix.

In addition to preventing unnecessary transitions in the execution tree, APK provides a way of combining branching with tie-breaking rules. We extend the APK function by user-defined priorities: $\mathsf{APK}^2 : \mathsf{E} \times \mathsf{E} \rightarrow \{\mathsf{N}, \mathsf{P}, \mathsf{S}, \mathsf{N}_<, \mathsf{N}_>\}$. If e_1 has a higher priority than e_2, then $\mathsf{APK}^2(\mathsf{e}_1, \mathsf{e}_2) = \mathsf{N}_<$. The definition of $\mathsf{N}_>$ is analogous. Note that, in contrast to APK, the function APK^2 is not symmetric. At run time, the branching mechanism always executes higher-prioritized events before events with lower priorities.

During the calculation of the execution tree, zero-delay events may be created. While they are not as problematic as in PDES (cf. section 4), they nevertheless have to be handled in a consistent way. In our implementation, zero-delay events are scheduled immediately after their creation, and thus included into the branching process. As a consequence, however, all APK values have to account for possible zero-delay events, too.

When the set of different leaf states of the execution tree has been computed, the simulation has to be continued for all resulting branches. Therefore, all leaf states are saved to hard disk, together with a snapshot of the global system state, and one of them is restored immediately for simulation. Whenever simultaneous events are encountered again, another execution tree is analyzed which may lead to the creation of further branches. When the end of a branch is reached, i.e. the termination condition of the simulation is fulfilled, one of the saved branches is restored and the simulation is resumed. Finally, when all branches have been completed, the simulation is over.

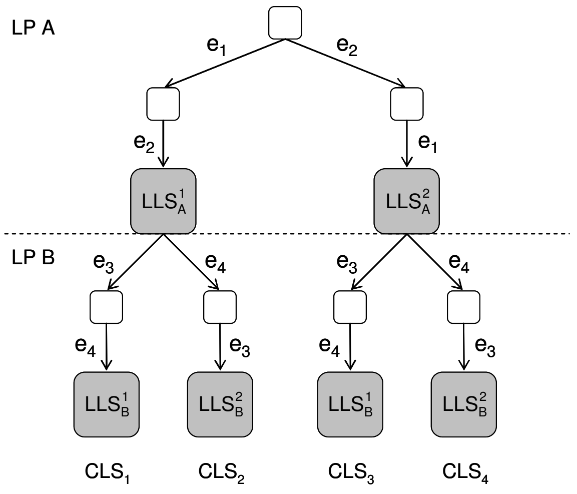


Figure 4: Combined execution tree for two LPs.

The described branching mechanism guarantees the calculation of all possible simulations results. It is efficient in the sense that, with the help of APK and node comparisons, unnecessary overhead is reduced and duplicate branches are avoided. Generally, we found the computation of execution trees to have almost negligible run-time, in the order of seconds. Furthermore, when compared to independent runs with different tie-breaking rules, all computations up to a branching point are shared among the branches. Nevertheless, the run-time of a simulation may become large when many branches are created. If, at a branching point, n different branches are created, the remaining simulation takes about n times as long as with tie-breaking rules. Therefore, it may pay off to compute multiple branches in parallel by distributing them onto other hosts or processors (if they are available). During the design and implementation of the branching mechanism for PDES, however, we discovered a way which may strongly increase performance even when additional hosts are not available. We will return to this at the end of section 4.4.

4. BRANCHING IN PDES

The design of a branching mechanism for parallel and distributed simulation presents several challenges when compared to sequential DES. First of all, when simultaneous events with timestamp t are detected at an LP, its FEL does not necessarily contain all events for t. Furthermore, simultaneous events may occur at different LPs at the same time, yet with different timestamps. While, in sequential DES, the analysis of different sets of simultaneous events with different timestamps is always performed in increasing timestamp order, this is not guaranteed in distributed simulation and leads to the danger of globally inconsistent simulation states [26]. Finally, calculating the execution tree is a complex task, since simultaneous events with identical timestamps may be spread among different LPs.

It is important that possible zero-delay events sent between LPs are taken into account during the calculation of the execution tree. Otherwise, there is the danger of "overlooking" possible execution orders [26]. Therefore, the analysis of simultaneous events at a timestamp t has to be performed in coordinated fashion, and either a global control or a communication protocol between the LPs is required. Each LP l is only able to calculate its *local execution tree* and its *local leaf states* (LLS). A local leaf state represents the state of l after executing the events in a specific order, and is defined as a 2-tuple $\mathsf{LLS}_\mathsf{l}^\mathsf{i} = (\mathsf{S}^\mathsf{i}, \mathsf{E}^\mathsf{i})$, $\mathsf{i} = 1 \ldots \mathsf{n}$, with n being the number of different leaf states. S^i is the resulting system state of l, and E^i is the set of future events created by l on the corresponding branch. In the process of branching, the *combined leaf states* (CLS) of the *combined execution tree* have to be computed. Figure 4 shows an example of a combined execution tree for a simple scenario with two LPs. Every CLS contains the LLS of all LPs $\mathsf{l}_1, \ldots, \mathsf{l}_\mathsf{N}$ on the corresponding branch: $\mathsf{CLS}_\mathsf{k} = \{\mathsf{LLS}_{\mathsf{l}_1}^{\mathsf{i}_1}, \ldots, \mathsf{LLS}_{\mathsf{l}_\mathsf{N}}^{\mathsf{i}_\mathsf{N}}\}$, where $\mathsf{i}_1, \ldots, \mathsf{i}_\mathsf{N}$ are the identifiers of the different LLS, and k is the unique ID of the resulting branch. Note that one and the same LLS is usually contained in more than one CLS.

In the following, we divide the discussion of the PDES branching mechanism into three sections: The detection of simultaneous events, the distributed branching procedure, and the computation of the resulting branches. Each section gives an overview of the used concepts, based on the detailed descriptions in [24, 25], and at the same time presents and discusses interesting implementation details.

4.1 Detection of simultaneous events

We use a central control, the *Branching Manager* (BM), for the coordination of the LPs. When an LP detects simultaneous events, say with timestamp t, in its local FEL, it reports them to the BM and requests a global synchronization at t. This is done in order to reach a consistent simulation state, and may be realized with a simple communication protocol as is commonly used for barrier synchronizations. In case the BM receives another report about simultaneous events with a timestamp $\mathsf{t}' < \mathsf{t}$, it cancels the former barrier and issues a new one at t'. Note that, with optimistic synchronization, this may require artificial rollbacks. When all LPs have stopped the simulation at the smallest reported timestamp, a *branching point* is reached [24].

In order to avoid unnecessary global synchronizations, an LP may check the interaction of simultaneous events already *before* reporting them to the BM, if APK (cf. section 3) is available. If all events are found to be non-interacting, the LP refrains from reporting them. The number of global synchronization points may be reduced further if the user is only interested in certain types of simultaneous events. For example, when evaluating a new TCP variant, the user may be interested in simultaneous occurrences of events like "received acknowledgement" and "timeout", since their execution order may well affect the behavior and throughput of TCP flows. In contrast, the user may consider other simultaneous events as irrelevant for the concrete simulation study and decide not to examine them. We therefore offer the possibility of declaring *candidate events* during simulation initialization. In the course of the simulation, LPs only report simultaneous candidate events to the BM. All other simultaneous events are handled normally, e.g. with tie-breaking rules. This provides a way of using the branching mechanism in conjunction with tie-breaking rules [25].

We examined two different ways of implementing candidate events. The first implementation simply distinguishes candidate events from normal events. While a boolean flag would be sufficient, it is impractical to make the scheduler and the synchronization mechanism aware of this distinc-

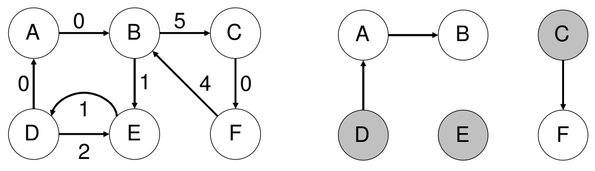


Figure 5: Lookahead graph and influence graph.

tion. A solution is to integrate candidate event information into extended event timestamps (cf. section 2.1), so that timestamps are identical only for candidate events. However, depending on the concrete tie-breaking rule, the extension timestamp may contain arbitrary combinations of bits. Therefore, whenever a candidate event is created, we set its extension timestamp to "$1\dots1$", i.e. the highest possible value. This way, all candidate events with identical basic timestamps have completely identical timestamps, and are thus detected as simultaneous events. This implementation ensures that branching is only carried out on simultaneous events that are of interest to the user. Note that all non-candidate events with the same basic timestamp have already been executed when simultaneous candidate events are detected. Nevertheless, in our opinion, this is the most convenient way. If, for example, the extension timestamps of candidate events were set to "$0\dots0$" instead, further candidate events generated by other simultaneous events might be missed. If this danger exists, however, we suggest to simply declare the other events as candidate events, too.

An alternative implementation of candidate events is possible by using APK tables: All pairs consisting of two candidate events are set possibly-interacting, while all other event pairs are declared non-interacting. This is an elegant, transparent solution and allows for an exact specification of simultaneous events of interest.

4.2 Distributed branching procedure

When a branching point at timestamp t has been reached, the combined execution tree has to be calculated. Since LPs with only a single event at t may generate or receive zero-delay events, they have to be included into the branching process. Therefore, as a first step, the BM discovers all LPs with events at t by sending query messages. Let the set of LPs with at least one event at t be M. Then, the BM has to coordinate the branching process among the LPs in M so that their interdependencies are respected and all combined leaf states are calculated.

The complexity of the branching procedure varies depending on possible zero-delay events, or cycles (cf. section 2.1). Therefore, the BM utilizes available lookahead information between the LPs in order to determine its further actions. Let i and j be two LPs, and t the local simulation clock of LP i. Then, i can guarantee that any event it schedules at j has a timestamp of at least $\mathsf{t}+\mathsf{x}$. We define $\mathsf{lookahead}_{\mathsf{i},\mathsf{j}}$ as the largest value $\mathsf{x} \geq 0$ which holds that guarantee. If i is not able to send events to j at all, then $\mathsf{lookahead}_{\mathsf{i},\mathsf{j}} = \infty$.

We define a directed weighted graph $L = (\mathsf{V}, \mathsf{E})$ with $\mathsf{V} = \{\mathsf{v} : \mathsf{v} \text{ is LP}\}$, $\mathsf{E} = \{(\mathsf{u}, \mathsf{v}) : \mathsf{lookahead}_{\mathsf{u},\mathsf{v}} < \infty\}$, and a weight function $w : \mathsf{E} \to \mathbb{R}$ with $w(\mathsf{u}, \mathsf{v}) = \mathsf{lookahead}_{\mathsf{u},\mathsf{v}}$. L is called the *lookahead graph*.

From L, we derive the *influence graph*, a directed unweighted graph $I = (\mathsf{V}', \mathsf{E}')$ with $\mathsf{V}' = \mathsf{V}$, and $\mathsf{E}' = \{(\mathsf{u}, \mathsf{v}) \in \mathsf{E} : w(\mathsf{u}, \mathsf{v}) = 0\}$. I reflects possible zero-delay events between the LPs. An example of a lookahead graph as well as the derived influence graph is shown in figure 5.

While lookahead information can often be extracted from simulation models, it is well known that especially in dynamic simulation scenarios the specification of *precise* lookahead values is sometimes hard. However, this is unproblematic here: In order to construct the influence graph, it is sufficient to know whether lookahead values are greater than or equal to zero. For example, in network simulation, the concrete values of propagation delays are irrelevant, as long as they are greater than zero. Therefore, it should be possible to construct influence graphs for many simulation models found in practice.

At the beginning of the simulation, if lookahead information is available, the BM computes I and performs a cycle check with well-known graph algorithms. The result defines the branching routine used for the coordination of LPs during the simulation: *Acyclic branching* or *cyclic branching*.

If I is available and acyclic, the BM triggers branching at individual LPs iteratively, depending on possible zero-delay events. All LPs stay ready to calculate their LLS on receipt of a *branch message* from the BM. During the calculation of its local execution tree, an LP l schedules all internal zero-delay events immediately after their creation, just as in sequential DES. When l is finished calculating its LLS, it stores them (i.e. the corresponding system state variables as well as all created future events) to hard disk. Finally, l sends identifiers of the saved states to the BM, together with all created zero-delay events for other LPs. This way, the BM can deliver all created zero-delay events to their destination LPs in the further branching process by including them into the branch messages. Since it may be necessary to branch an LP more than once (with different sets of zero-delay events as input), every LP saves its original state before branching, and restores that state when receiving the next branch message.

When calculating its local execution tree, an LP l uses node comparisons and APK (cf. section 3) in order to eliminate duplicate branches. This guarantees that each LP only returns different LLS to the BM. Since the local execution tree of l only contains events handled by l, it is possible to reduce the two concepts to LP scope: For node comparisons, the tuple introduced in section 3 is replaced by $(\mathsf{E}_{\mathsf{exec}}, \mathsf{E}_{\mathsf{pend}}, \mathsf{E}_{\mathsf{new}}, \mathsf{S}_{\mathsf{l}})$, where the new component S_{l} denotes the state variables of l. Thus, much smaller sets of state variables have to be compared than with sequential DES. Furthermore, with regard to APK, it is clear that two events handled by different LPs are always non-interacting. We define the *local APK function* of l as $\mathsf{APK}_{\mathsf{l}} : \mathsf{E}_{\mathsf{l}} \times \mathsf{E}_{\mathsf{l}} \to \{\mathsf{N}, \mathsf{P}, \mathsf{S}, \mathsf{N}_{<}, \mathsf{N}_{>}\}$, where E_{l} is the set of all events that may be scheduled at l. The possible values of the local APK function are the same as in sequential DES. Local APK usage reduces the size of APK tables considerably.

The iterative branching procedure is based on sets of *safe* LPs. Branching at an LP l is called safe when it is guaranteed that no other LP will create any zero-delay events destined to l. In every iteration, the BM computes the *safe set* $\mathsf{Z} = \{\mathsf{v} \in \mathsf{V} : \mathsf{indeg}(\mathsf{v}, I) = 0\}$. Z contains the LPs in I without any incoming edges (in figure 5, these are the gray-colored LPs). With Z, the *branch set* $\mathsf{B} = \mathsf{M} \cap \mathsf{Z}$ may be computed. After sending branch messages to the LPs in B, and receiving their computed LLS, the BM updates the combined execution tree and adds new LPs to M depending on the destinations of created zero-delay events. Finally, the BM calculates the next safe set and proceeds with the next iteration. Since I is acyclic, eventually $\mathsf{M} = \emptyset$, and all CLS have been calculated. Algorithm 1 summarizes acyclic branching in pseudocode.

Algorithm 1 : Acyclic branching

```
input: influence graph $I = (\mathsf{V}', \mathsf{E}')$
  $\mathsf{M} \Leftarrow$ all LPs $\in \mathsf{V}'$ with events at $\mathsf{t}$
  $\mathsf{Z} \Leftarrow \{\mathsf{v} \in \mathsf{V}' : \mathsf{indeg}(\mathsf{v}, I) = 0\}$ // initialize safe set
  while $\mathsf{M} \neq \emptyset$ do
    $\mathsf{B} \Leftarrow \mathsf{M} \cap \mathsf{Z}$ // compute branch set
    $\mathsf{M} \Leftarrow \mathsf{M} \setminus \mathsf{Z}$ // update $\mathsf{M}$
    $I \Leftarrow I \setminus \mathsf{Z}$ // update $I$
    if $\mathsf{B} \neq \emptyset$ then
      perform local branching for all LPs in $\mathsf{B}$
      for all received LLS $\mathsf{s}$ do
        update combined execution tree with $\mathsf{s}$
      end for
      for all newly created zero-delay events $\mathsf{e}$ do
        $\mathsf{M} \Leftarrow \mathsf{M} \cup$ destination LP of $\mathsf{e}$
      end for
    end if
    $\mathsf{Z} \Leftarrow \{\mathsf{v} \in \mathsf{V}' : \mathsf{indeg}(\mathsf{v}, I) = 0\}$ // update safe set
  end while
```

If I contains a cycle, or is not available at all, there is no clear ordering of the LPs in M, and cyclic branching has to be used. We provide two alternative strategies, which the user may choose from at the beginning of the simulation: *Event-based branching* and *batch-based branching*.

The event-based branching algorithm guarantees the calculation of the complete combined execution tree. First of all, the BM requests information about the events at t from all LPs in M, in order to construct a global view. In the following, it emulates the branching procedure used for sequential simulation (cf. section 3) by always triggering the execution of single events, thus largely ignoring the LP partitioning. Each LP l waits for *event execution requests* from the BM. On reception of an event execution request for event e, l executes e and saves the resulting state. Then, l performs node comparisons with already existing states and sends a reply containing the result as well as any created zero-delay events to the BM. The BM updates the combined execution tree and delivers all zero-delay events to their destination LPs. Note that, in contrast to acyclic branching where APK is used by the LPs, here the BM uses APK to avoid unnecessary event executions.

Obviously, event-based branching has a much higher synchronization overhead when compared to acyclic branching. However, if the user declares candidate events, and APK is already used during the detection of simultaneous events, the set of simultaneous events will typically be small in many simulation scenarios. Thus, in most cases arising in practice, event-based branching should easily be feasible. Algorithm 2 summarizes event-based branching.

The alternative algorithm, batch-based branching, reduces the complexity of the branching process noticeably but only guarantees the calculation of a (potentially large) subset of branches. The BM sends branch messages to all LPs in M immediately and updates the combined execution tree with all received LLS. In the next iteration, M is defined as the set of all destination LPs of zero-delay events. Then, new

Algorithm 2 : Event-based branching

```
  $\mathsf{M} \Leftarrow$ all LPs with events at $\mathsf{t}$
  for all LPs $\mathsf{l}$ in $\mathsf{M}$ do
    request the set of events with timestamp $\mathsf{t}$ from $\mathsf{l}$
  end for
  $\mathsf{E} \Leftarrow$ all received events // complete event set
  Emulate sequential branching of $\mathsf{E}$.
```

branch messages including the zero-delay events are sent to the LPs in M, together with identifiers of their corresponding LLS of the last iteration. The LPs then restore these LLS one after another, and perform local branching for the received zero-delay events. This procedure is repeated until no more zero-delay events are created. Thus, batch-based branching is similar to acyclic branching, with the difference that an LP may enter M more than once. See algorithm 3.

Algorithm 3 : Batch-based branching

```
  $\mathsf{M} \Leftarrow$ all LPs with events at $\mathsf{t}$
  while $\mathsf{M} \neq \emptyset$ do
    perform local branching for all LPs in $\mathsf{M}$
    $\mathsf{M} \Leftarrow \emptyset$ // clear $\mathsf{M}$
    for all received LLS $\mathsf{s}$ do
      update combined execution tree with $\mathsf{s}$
    end for
    for all newly created zero-delay events $\mathsf{e}$ do
      $\mathsf{M} \Leftarrow \mathsf{M} \cup$ destination LP of $\mathsf{e}$ // update $\mathsf{M}$
    end for
  end while
```

In this section, we have discussed different algorithms for the coordination of branching. Acyclic branching takes advantage of available information in the form of an influence graph, and may be used when zero-delay cycles are not possible. If zero-delay cycles may occur, however, cyclic branching has to be performed. Depending on the simulation scenario, either event-based branching or batch-based branching may be preferable. An important property of all algorithms presented here is that they guarantee both progress and reproducibility of the simulation.

4.3 Computation of branches

The branching mechanism for sequential DES (cf. section 3) stores branches on hard disk by taking snapshots of global system states. Then, they are computed successively, one after another. In PDES, however, the partitioning into LPs makes it possible to reduce simulation run-times by using *cloning* techniques. The basic idea of cloning is to explore alternative scenarios, but share common computations among them. This is achieved with the help of *virtual LPs*, which represent LPs in specific scenarios, and *physical LPs*, which perform the actual computations, e.g. execute events. By mapping multiple virtual LPs to the same physical LP, computations can be shared among different scenarios. A physical LP which performs computations for more than one virtual LP is called a *shared LP*. Cloning is usually triggered at *decision points*, when one or more LPs are required to show different behavior. For example, an LP may represent a packet marker, and the possible alternative actions are to mark a packet in order to drop it, or let the packet proceed. At a decision point, new scenarios are created globally, and new physical LPs are created for all LPs that are immediately affected by the alternative actions. For all other LPs, virtual LPs are created and mapped to existing physical LPs. In the further course of the simulation, all shared clones monitor the messages they receive. Whenever an LP receives different messages for different scenarios, it clones itself, which leads to the creation of a new physical LP and an update of the LP mapping table. For an overview of cloning techniques see e.g. [3, 11].

Branching points logically correspond to decision points, and branches correspond to scenarios. Therefore, we have implemented cloning as an alternative to our original method (in the following referred to as *classical branching*). We have realized a flexible integration of cloning based on an interface: At a branching point, when the combined execution tree has been calculated, the interface receives all combined leaf states. Then, with classical branching, the save-and-restore approach is taken. With cloning, however, the combined leaf states are examined and new physical LPs are created for all involved LPs. All other LPs are set as shared clones accordingly. Finally, all future events created in the process of branching are extracted from the combined leaf states and scheduled at their receiving LPs. Then, the simulation is resumed, and all branches are simulated at once, sharing event computations as much as possible [25].

4.4 Performance

In this section, we examine the performance of our PDES branching mechanism. We have conducted several simulation series with different synchronization mechanisms, and compare the run-times of branching to corresponding simulation runs with tie-breaking rules. The branching mechanism was configured to use cloning techniques. For all simulations, we used 4 hosts with clock speeds of 1.8GHz and Ubuntu linux 2.6.15-26-38. We ran multiple replications of all simulations, and calculated 95% confidence intervals.

For our simulations, we used the synthetic workload model described in [23], since it provides easy configurable scenarios with well-defined behavior. The model consists of nodes, which send events to each other. Each node is mapped to its own LP. When a node executes an event with timestamp t, it schedules a new event at another, randomly chosen node. The timestamp of the new event is randomly chosen from the interval $[\mathsf{t}+1, \mathsf{t}+10]$. We configured the model to consist of 300 nodes, with 3 initial events at every node. Additionally, we explicitly scheduled 3 simultaneous events at one specific node at the beginning of the simulation, and another 3 simultaneous events shortly afterwards. Thus, a total of $3! \cdot 3! = 36$ branches had to be computed. With regard to APK, we set all events to be possibly-interacting, so that execution trees had to be explored completely. All simulations were run for 1,000 units of virtual time.

Let $t_{\mathsf{branching}}$ be the run-time of a simulation with branching, and t_{tiebreak} the run-time with a tie-breaking rule instead. Then we define the *branching overhead* as

$$BO = \frac{t_{\mathsf{branching}}}{t_{\mathsf{tiebreak}}}$$

Obviously, the smaller the branching overhead, the better the performance of the branching mechanism. Note that, in the simulation scenarios at hand, with classical branching we would always have $BO \approx 36$ (the number of branches), independent of the synchronization mechanism.

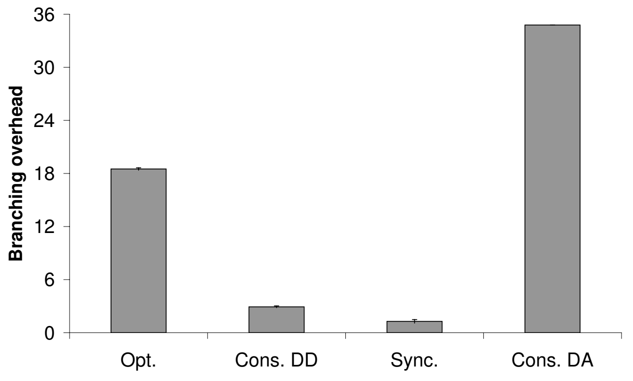


Figure 6: Run-time of the branching mechanism compared to tie-breaking.

Figure 6 shows the branching overhead when cloning is used. For all synchronization mechanisms we find $BO < 36$. First of all, this means that the simulations with cloning run faster than with classical branching. There are, however, large differences depending on the synchronization mechanism. With optimistic synchronization (“Opt.”), $BO \approx 18.5$, which means that the computation of all 36 branches takes about 18 times as long as the simulation with a tie-breaking rule. If, however, a conservative deadlock detection scheme (“Cons. DD”) is used, our results show $BO \approx 3$. This means that the simulations with branching only took three times as long as the corresponding simulations with tie-breaking rules. Even more encouraging are the results with a completely synchronous, globally clocked synchronization mechanism (“Sync.”). With $BO \approx 1.3$, the cost of branching is almost negligible. We therefore conclude that the use of cloning is especially profitable when frequent global synchronization between the LPs is required. This is confirmed by our simulations with a conservative deadlock avoidance scheme (“Cons. DA”). Here, $BO \approx 35$, which means that the use of cloning brings almost no benefit. We attribute this to the fact that the increasing number of LPs also leads to a much larger amount of null messages.

In summary, cloning always achieved speedup in comparison to classical branching. Furthermore, with some synchronization mechanisms, the branching overhead was extremely small. Thus, it is possible to analyze the effects of simultaneous events at a relatively low cost. If we take into account that the examined simulation scenarios are far from optimal for cloning (we made sure that all shared LPs were eventually cloned), this an encouraging result.

When comparing the PDES branching mechanism to the implementation for sequential DES, we observe that detecting and analyzing simultaneous events is much more complicated in PDES. On the other hand, this is more than compensated by the performance gains resulting from the use of cloning, which is possible due to the partitioning into LPs. We therefore think that an LP partitioning may also benefit the analysis of simultaneous events in *sequential* simulation. It has already been shown that the use of PDES concepts for sequential simulation speeds up simulation runs in some simulation scenarios [5, 15]. Therefore, we expect noticeable performance improvements when incorporating cloning into the sequential branching mechanism.

5. MOOSE

As the preceding sections have shown, analyzing simultaneous events is much more complex than the use of tie-breaking rules. While the calculation of execution trees may be implemented largely independent of the simulator core, it requires some effort to "hook" the branching mechanism into an existing simulation tool. For example, a sequential simulation tool has to provide the following functionality which is essential for branching:

- Simultaneous events have to be detected explicitly by the event scheduler.
- It must be possible to save and restore global states. This includes the system state variables of the simulation modules as well as the state of the simulator.
- All simulation modules have to support state comparisons, so that duplicate states may be removed from execution trees.

Additional functionality is required in case of a parallel and distributed simulation tool:

- The simulator has to support a consistent way of stopping the simulation at a branching point. Although this is easily realized with modified barrier synchronizations (cf. section 4.1), it may require some extensions to an existing simulator core.
- Depending on the implementation of candidate events (cf. section 4.1), they either have to be supported by extended event timestamps, or APK tables have to be made accessible by the scheduler.
- Events which are created during the branching process have to be "intercepted" instead of being scheduled normally by the LPs. These events have to be stored together with the LP system states in order to represent combined leaf states correctly.
- For an efficient computation of branches, it is necessary to use cloning techniques (cf. section 4.3). The implementation of a cloning mechanism, however, has several implications, as described in [11].

For these reasons, we have decided to develop a new simulation tool, MOOSE [17], which implements the branching mechanisms for both sequential DES and PDES. Furthermore, MOOSE also supports the handling of simultaneous events with tie-breaking rules, as well as several other features commonly found in simulation tools. It is, however, still under active development and by no means completed. In the following, we give an overview of MOOSE, with a focus on the components related to the branching mechanism.

MOOSE is written in Java and can be divided into four logical components (cf. figure 7). The *simulator core* contains all objects responsible for running the simulation, e.g. schedulers, LPs, and communication instances. Apart from sequential DES, MOOSE supports multithreading for parallel simulation and different synchronization mechanisms for distributed simulation.

The *branching* component contains the algorithms for the analysis of simultaneous events described in this paper. In case of PDES, there are specific interdependencies with the simulator core: For a consistent detection of simultaneous events, every synchronization mechanism provides a method for globally synchronizing all LPs. Furthermore, some parts of the two techniques for computing branches, i.e. cloning and classical branching (cf. section 4.3), have been implemented separately for each synchronization mechanism.

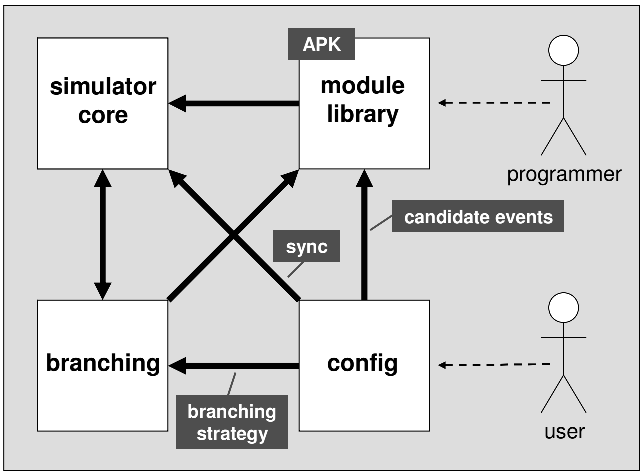


Figure 7: Central components of MOOSE.

The *module library* contains the simulation modules, representing e.g. network technologies and communication protocols. Although MOOSE has been developed for network simulation in the first place, there are general base classes for modules, which allows for the specification of any discrete-event simulation model. The base classes also provide standard interfaces for the execution of events, event scheduling, and trace output, so that module implementations are completely independent of the internals of the simulator core. In order to be able to compare system states during branching, each simulation module has to implement a standard function for an equality check. Furthermore, if specified by the programmer, a module may provide an APK library to increase the efficiency of branching (cf. section 3). If a module does not provide APK, all its events are assumed to be possibly-interacting.

The *configuration* for specific simulation runs may be done either with command line parameters or a configuration file. Currently, MOOSE configuration files are written in *Groovy*, an object-oriented scripting language for the Java platform. However, support for other scripting languages, e.g. *Jython*, is easily possible. Apart from standard settings like the scheduler type, or whether sequential or parallel and distributed DES is used, the strategy for handling simultaneous events may be specified: Tie-breaking rules, classical branching, or branching with cloning techniques. Furthermore, the user may mark event types, or pairs of events, as candidate events for branching.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have discussed the issue of simultaneous events in discrete-event simulation, which is the most popular technique used for network simulation. We have argued that the user should have a means of examining different execution orders of simultaneous events, and presented a new simulation tool, MOOSE, which provides this functionality.

Simultaneous events are analyzed with a branching mechanism, which we have designed and implemented for both sequential, and parallel and distributed simulation. With APK tables and candidate events, the branching mechanism may be used together with tie-breaking rules.

Our experiments with cloning techniques have shown encouraging results with regard to the overhead of branching. It appears that cloning can make branching a practical method. Therefore, we plan to use the LP paradigm for sequential branching, too. We expect this to lead to a much more efficient analysis of simultaneous events in sequential simulation.

In future work, we are also going to examine the performance of branching in a wider range of simulation scenarios, and perform case studies in network simulation. In the process, we will further extend the module base as well as the APK library of the simulator.

7. REFERENCES

[1] C. Barz, R. Göpffarth, P. Martini, and A. Wenzel. A new framework for the analysis of simultaneous events. In *Proceedings of the 2003 Summer Computer Simulation Conference (SCSC '03)*, 2003.

[2] L. Bononi, M. Bracuto, G. D'Angelo, and L. Donatiello. Analysis of high performance communication and computation solutions for parallel and distributed simulation. In *Proceedings of the 2005 International Conference on High Performance Computing and Communications (HPCC '05)*, 2005.

[3] D. Chen, S. J. Turner, W. Cai, B. P. Gan, and M. Y. H. Low. Algorithms for HLA-based distributed simulation cloning. *ACM Transactions on Modeling and Computer Simulation*, 15(4):316–345, 2005.

[4] B. A. Cota and R. G. Sargent. Simultaneous events and distributed simulation. In *Proceedings of the 1990 Winter Simulation Conference (WSC '90)*, 1990.

[5] R. Curry, C. Kiddle, R. Simmonds, and B. Unger. Sequential performance of asynchronous conservative PDES algorithms. In *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation (PADS '05)*, 2005.

[6] DaSSF. version 3.2.5. `http://www.crhc.uiuc.edu/~jasonliu/projects/ssf/`.

[7] R. M. Fujimoto. Zero lookahead and repeatability in the high level architecture. In *Proceedings of the 1997 Spring Simulation Interoperability Workshop*, 1997.

[8] R. M. Fujimoto. *Parallel and Distributed Simulation Systems.* Wiley Interscience, 2000.

[9] GloMoSim. version 2.03. `http://pcl.cs.ucla.edu/projects/glomosim/`.

[10] GTNetS. `http://www.ece.gatech.edu/research/labs/MANIACS/GTNetS/`.

[11] M. Hybinette and R. M. Fujimoto. Cloning parallel simulations. *ACM Transactions on Modeling and Computer Simulation*, 11(4):378–407, 2001.

[12] J-Sim. version 1.3. `http://www.j-sim.org/`.

[13] V. Jha and R. Bagrodia. Simultaneous events and lookahead in simulation protocols. *ACM Transactions on Modeling and Computer Simulation*, 10(3):241–267, 2000.

[14] JiST/SWANS. version 1.0.6. `http://jist.ece.cornell.edu/`.

[15] C. Kiddle, R. Simmonds, and B. Unger. Channel based sequential simulation. In *Proceedings of the 37th Conference on Winter Simulation (WSC '05)*, 2005.

[16] H. Mehl. A deterministic tie-breaking scheme for sequential and distributed simulation. In *Proceedings of the 6th Workshop on Parallel and Distributed Simulation (PADS '92)*, 1992.

[17] MOOSE - Module-based Object-Oriented Simulation Environment. `http://web.cs.uni-bonn.de/IV/MOOSE/`.

[18] NCTUns. version 3.0. `http://nsl.csie.nctu.edu.tw/nctuns.html`.

[19] Ns-2. `http://www.isi.edu/nsnam/ns/`.

[20] OMNET++. version 3.4b2. `http://www.omnetpp.org/`.

[21] ParSec. version 1.1. `http://pcl.cs.ucla.edu/projects/parsec/`.

[22] PDNS. version 2.27.1a. `http://www.cc.gatech.edu/computing/compass/pdns/`.

[23] P. Peschlow, T. Honecker, and P. Martini. A flexible dynamic partitioning algorithm for optimistic distributed simulation. In *Proceedings of the 21st Workshop on Principles of Advanced and Distributed Simulation (PADS '07)*, 2007.

[24] P. Peschlow and P. Martini. Analyzing simultaneous events in distributed simulation. In *Proceedings of the 19th European Modeling and Simulation Symposium (EMSS '07)*, October 2007.

[25] P. Peschlow and P. Martini. Efficient analysis of simultaneous events in distributed simulation. In *Proceedings of the 11th IEEE International Symposium on Distributed Simulation and Real Time Applications (DS-RT '07)*, October 2007.

[26] R. Rönngren and M. Liljenstam. On event ordering in parallel discrete event simulation. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation (PADS '99)*, 1999.

[27] T. J. Schriber and D. T. Brunner. Inside discrete-event simulation software: How it works and why it matters. In *Proceedings of the 2006 Winter Simulation Conference (WSC '06)*, 2006.

[28] SWAN. version 1.0.1a. `http://www.eg.bucknell.edu/swan/`.

[29] TCP. Transmission Control Protocol (RFC 793), 1981.

[30] X. Wang, S. J. Turner, and S. J. E. Taylor. COTS simulation package (CSP) interoperability - a solution to synchronous entity passing. In *Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation (PADS '06)*, 2006.

[31] A. Wenzel. Experiences with simultaneous events using discrete-event simulation. In *Proceedings of the IASTED International Conference on Modeling and Simulation*, 1999.

[32] F. Wieland. The threshold of event simultaneity. In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation (PADS '97)*, 1997.