

Efficient Analysis of Simultaneous Events in Distributed Simulation

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

Abstract

Simultaneous events are a fundamental challenge in distributed discrete-event simulation: Different execution orders may lead to different simulation results. Commonly, simultaneous events are handled by tie-breaking rules in order to guarantee reproducibility. A second approach, however, is examining different execution orders with a branching mechanism. By analyzing the effects of simultaneous events, confidence in simulation results may be increased. Naturally, branching may get expensive with large numbers of simultaneous events. Thus, efficiency is crucial for branching in order to be a practical method.

In this paper, we present an efficient branching mechanism for simultaneous events in distributed simulation, which may be used in conjunction with, or as an alternative to tie-breaking rules. We apply techniques which optimize the detection of simultaneous events and reduce the number of branches as much as possible. Furthermore, we use cloning to share computations among different branches.

1 Introduction

Distributed discrete-event simulation is a very popular technique for the performance evaluation of systems. Compared to sequential simulation, distributed simulation is able to meet high memory demands of complex scenarios, and speeds up simulation runs. One of the fundamental challenges in distributed simulation is the problem of simultaneous events: Different execution orders may lead to different states and thus to different simulation results.

Commonly, simultaneous events are handled by tie-breaking rules, which use priority schemes in order to enforce deterministic event orderings. While an adequate implementation of tie-breaking rules is a nontrivial task in distributed simulation, satisfactory solutions have been found. We give a detailed overview of previous research on tie-

breaking rules in section 2. For a general introduction to distributed simulation techniques see e.g. [8].

An important aspect has mostly been neglected by the research community: What impact do simultaneous events have on simulation results? With tie-breaking rules, one of many possible orderings is chosen every time simultaneous events occur. However, in many cases there exist several execution orders that lead to correct simulation results. As an example, consider a completely filled packet buffer, with an enqueue event and a dequeue event scheduled to happen simultaneously. Then, both execution orders are correct in the sense of the simulation model, yet they may well lead to different results. Consequently, selecting a single simulation result according to a tie-breaking rule implicitly labels all other results as irrelevant (or wrong).

Should we care? In our opinion, the answer depends on the simulation scenario and the purpose of the simulation. If it is known that, in the simulation scenario at hand, simultaneous events do not (or only marginally) influence the simulation results, then tie-breaking rules are perfectly adequate, efficient solutions. However, examples where simultaneous events may influence simulation results noticeably have been shown, e.g. in aviation simulation [25] and network simulation [24]. Furthermore, with relatively abstract simulation models like e.g. communication protocols in early stages of design, different execution orders of simultaneous events may have a significant effect on simulation results. Therefore, the user should have a means of analyzing the impact of simultaneous events.

We have developed a *branching mechanism* for the analysis of simultaneous events in distributed simulation. The branching mechanism detects simultaneous events, analyzes their interactions, and evaluates their impact by splitting the simulation run into different branches, thus computing a set of possible simulation results. Naturally, efficiency is crucial for the applicability of branching, which may become costly when the number of different branches is large. Therefore, in this paper, we present several techniques which increase efficiency at different stages of the

branching process. We optimize the detection of simultaneous events, and examine event interactions in order to cut down the number of branches. Furthermore, we incorporate cloning techniques into the branching mechanism in order to share computations among different branches.

The paper is structured as follows: In section 2 we discuss previous research on simultaneous events, and give an introduction to cloning mechanisms. In section 3 we present our branching mechanism and the techniques we apply for maximizing its efficiency. Section 4 summarizes the paper and outlines possible directions for future work.

2 Current state of research

In this section, we give an overview of the current state of research in the area of simultaneous events. Furthermore, we summarize previous work on cloning techniques.

2.1 Simultaneous events

In discrete-event simulation, two or more events with identical timestamps are referred to as *simultaneous events*. Commonly, simultaneous events are handled by *tie-breaking rules*, which use event priorities in order to enforce well-defined orderings. In sequential simulation, the definition of priority schemes is straightforward.

In distributed simulation, the system state is partitioned into logical processes (LPs), which are distributed onto the participating hosts. Each logical process has its own event list and handles all events related to its part of the state. Other events are forwarded to their destination LPs. Since events may arrive at LPs out of order, conservative or optimistic synchronization mechanisms are used to ensure the correctness of the simulation.

The realization of tie-breaking rules is a nontrivial task in distributed simulation. With conservative synchronization, any guarantee (e.g. a null message) has to correctly reflect event priorities. With optimistic synchronization, an already executed event has to be rolled back when a simultaneous event with a higher priority is received [21]. In order to avoid making synchronization mechanisms aware of priorities, event timestamps are usually extended by additional bits which encode priorities [23]. Handling simultaneous events gets complex when the simulation model contains zero-delay cycles: Conservative synchronization mechanisms are in danger of deadlock, and optimistic synchronization may be caught in an endless rollback loop [17, 22]. Thus, apart from reproducibility, tie-breaking rules also have to ensure progress of the simulation.

In order to realize tie-breaking rules, research has concentrated on two approaches. The first approach supports zero-delay cycles by defining sophisticated, automatic tie-breaking mechanisms, e.g. [17, 22]. As an example, the tie-

breaking rule in [17] 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 principle, these approaches solve the problem of zero-delay cycles. However, they are also subject to criticism, since the overhead may noticeably slow down the simulation, and the automatic assignment of priorities raises the danger of inconsistencies when combined with user-defined priorities [22]. Therefore, the second approach focuses on user-defined priorities and low overhead. As shown in [14], user-defined locally consistent tie-breaking rules may be realized if the communication between LPs fulfills certain lookahead requirements. [6] discusses zero-delay events with a view to the common requirement that the result of a distributed simulation is equal to the result of the corresponding sequential simulation. It is shown that this can be guaranteed easily if the simulation model is free of zero-delay cycles.

Simultaneous events have also been discussed in the context of simulation standards. Modifications to the original time management services of the High Level Architecture [12] guarantee reproducibility in the presence of simultaneous events and zero-delay events [7]. The proposed scheme allows each federate to control the ordering of simultaneous events locally. Furthermore, a mechanism for ordering simultaneous events in distributed simulations based on DEVS models is proposed in [16].

A totally different question, however, was raised in [25]: Are tie-breaking rules really satisfactory as a general solution? As the examples in [24, 25] show, different tie-breaking rules may yield noticeably different simulation results. This implies that ordering simultaneous events according to tie-breaking rules may bias simulation results (cf. also [15]), and motivates a different approach: Increasing confidence in simulation results by examining more than one execution order of simultaneous events. First steps into this direction were taken in [2] by establishing a formal framework for the analysis of simultaneous events in sequential simulation. Furthermore, branching was proposed in order to calculate a set of simulation results. Our branching mechanism [20] is based on this framework.

2.2 Cloning

Cloning techniques for exploring alternative scenarios in distributed simulation were introduced in the context of interactive simulation [10]. At *decision points*, e.g. specified by the user, alternative actions may lead to different *scenarios*. While the traditional method of exploring alternative scenarios consists of running independent replications, the basic idea of cloning is to share common computations among different scenarios. In order to achieve that, the simulation is divided into a virtual and a physical layer. *Virtual LPs* belong to specific scenarios and are identified by their

LP ID, and a scenario ID. In contrast to that, *physical LPs* perform the actual computations, e.g. event executions. By mapping multiple virtual LPs to the same physical LP, computations are shared. A physical LP which has two or more associated virtual LPs is called a *shared clone*.

At a decision point, new scenarios are created globally, and new physical LPs are instantiated for LPs which are directly affected by the alternative actions. For all other LPs, virtual LPs with the new scenario IDs are created and mapped to existing physical LPs. The creation of new scenarios is usually done conservatively so that it is never rolled back, even with optimistic synchronization. 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. Since clones are only created on-demand, when the effects of alternative actions spread among LPs, this mechanism is called *incremental cloning*. It has been shown that cloning can reduce simulation run-times dramatically when compared to independent replications. Cloning is most advantageous when the differences between alternative scenarios spread slowly, or not at all. A comprehensive summary is given in [11].

Recently, two techniques which further increase the performance of incremental cloning have been proposed: *Just-in-time cloning* [9] delays cloning until it is absolutely necessary, and thus avoids rollbacks of on-demand cloning when optimistic synchronization is used. *Merging* [1] identifies cloned LPs in alternative scenarios which have converged to the same state again, and merges them together in order to avoid redundant computations.

In contrast to incremental cloning, *entire cloning* is a technique which clones all LPs immediately at a decision point. Thus, all scenarios run independently, and there are no shared computations. However, entire cloning takes advantage of the fact that processors are sometimes idle in a distributed simulation, and reduces these idle times by computing multiple scenarios in parallel. Furthermore, message comparisons are not necessary, which reduces overhead. An important aspect is that entire cloning may even be applied if there are no decision points at all: In [3], entire cloning is used to run independent simulations in parallel, in order to get statistically meaningful results faster.

Cloning mechanisms have also been developed for HLA-based distributed simulations. The term *active cloning* has been introduced to denote cloning at a decision point, while on-demand cloning based on message comparisons is referred to as *passive cloning*. Active cloning is realized with the help of global synchronization points for all federates. The performance results once again show the advantages of an incremental cloning approach. A detailed summary can be found in [5].

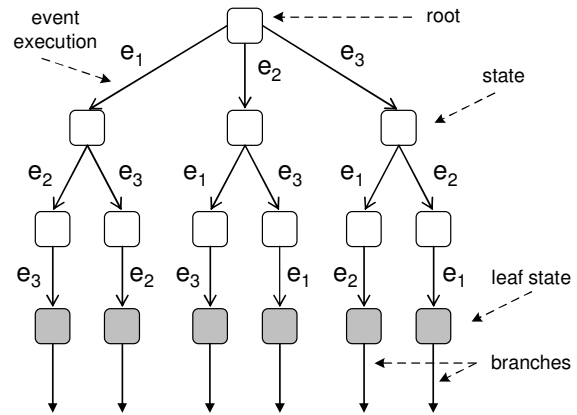


Figure 1. Execution tree for 3 simultaneous events e_1 , e_2 , and e_3 .

3 Analysis of simultaneous events

In this section, we first describe the branching mechanism as a basis for the following discussion. Then, we focus on the efficiency of branching.

A given set of simultaneous events can be visualized as an *execution tree* (cf. figure 1). Nodes represent system states, and transitions conform to event executions. Every path from the root to a leaf state corresponds to an execution order. Thus, when simultaneous events are detected, the task of a branching mechanism is to calculate the leaf states of the execution tree, and continue the simulation for all different leaf states. We refer to the resulting simulation runs as *branches*. In distributed simulation, the analysis of simultaneous events presents a variety of challenges:

Consistency Simultaneous events may occur at different LPs at the same time, and have to be analyzed in coordinated fashion to avoid inconsistent global simulation states. Thus, a global control or a communication protocol between the LPs is required for branching.

Completeness Depending on the simulation scenario, arbitrary combinations of LPs with simultaneous events are possible. Due to possible zero-delay events or zero-delay cycles, the calculation of the complete execution tree is a complex task.

Complexity For a given set E of simultaneous events there exist $|E|!$ possible event orderings. Therefore, the worst-case run-time of branching is exponential in the number of simultaneous events, and unlikely to be improved. As a consequence, with very large numbers of simultaneous events, a complete analysis will not be feasible with today's hardware. However, this also means that for practical use the efficiency of branching should be increased as much as possible.

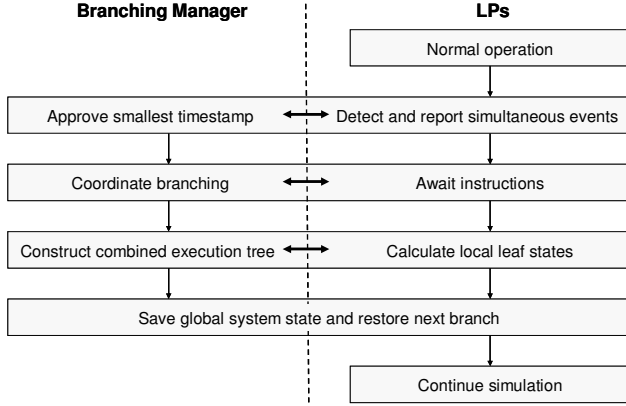


Figure 2. General structure of branching.

In [20], we have introduced a branching mechanism for simultaneous events and discussed the two aspects *consistency* and *completeness* in detail. In this paper, we focus on the most critical aspect, the *complexity* of branching.

Our branching mechanism uses an on-demand global control, the *Branching Manager* (BM), which coordinates the LPs during the branching process. Whenever an LP detects simultaneous events, it reports them to the BM and requests a global synchronization of all LPs. This is done in order to reach a consistent, error-free simulation state, the *branching point*. Whenever multiple LPs report simultaneous events, the synchronization takes place at the smallest reported timestamp t_{\min} . The global synchronization procedure may be realized in different ways, cf. [20].

At a branching point, the BM issues branching requests to the LPs with events at t_{\min} in an iterative manner, depending on possible zero-delay events. Every LP l then explores its *local execution tree* and calculates its *local leaf states* (LLS). An LLS represents the state of l after a specific event execution order and is defined as a 2-tuple $LLS_l^i = (S^i, E^i)$, $i = 1 \dots n$, 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.

With the received LLS, the BM then computes the *combined leaf states* (CLS) of the *combined execution tree*. A CLS contains information about the LLS of all LPs l_1, \dots, l_N on the corresponding branch: $CLS_k = \{LLS_{l_1}^{i_1}, \dots, LLS_{l_N}^{i_N}\}$, where i_1, \dots, i_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 may be contained in more than one CLS. For further details about the branching procedure see [20].

Finally, when all CLS have been calculated, they are saved to hard disk, together with a snapshot of the state of all uninvolved LPs. One of the states is restored immediately, the LPs are initialized accordingly, and the simulation proceeds with the computation of the current branch. This is repeated until all branches have been computed.

The general structure of the branching mechanism is summarized in figure 2. In the following, we present techniques for increasing the efficiency of branching. We discuss three stages of the branching mechanism separately: The detection of simultaneous events, the calculation of the execution tree, and the computation of branches.

3.1 Detection of simultaneous events

Synchronizing all LPs every time simultaneous events are detected may clearly result in a lot of unnecessary overhead: If two events are known to lead to the same simulation state regardless of their execution order, analyzing them is useless. Furthermore, the user may be interested in certain types of simultaneous events only, and consider others irrelevant. For example, when evaluating a communication protocol like TCP, the simultaneous events “received acknowledgement” and “timeout” may be of interest, since their execution orders can easily lead to different system states. We therefore restrict branching to specific event types by declaring *candidate events* during simulation initialization. In the course of the simulation, the LPs only report simultaneous candidate events to the BM. All other simultaneous events are handled by tie-breaking rules.

In order to distinguish candidate events from normal events, the structure of event timestamps may be utilized. For tie-breaking, event timestamps are usually divided into a *basic timestamp* field containing the virtual time value, and an *extension timestamp* used to generate unique timestamps. We simply set the extension timestamp of a candidate event to “1...1” when it is created, so that two candidate events will have identical timestamps whenever they have identical basic timestamps.

The notion of candidate events ensures that simultaneous events are only reported when they are of interest to the user. Apart from the extended timestamp scheme, there is yet another way of declaring candidate events. We will describe it as part of the next section.

3.2 Calculation of the execution tree

At a branching point, the BM coordinates the branching process at the involved LPs. Each LP has to calculate its LLS, return information about them to the BM¹, and await further instructions. Obviously, calculating all LLS can get very time-consuming when there are many simultaneous events. An important observation, however, is that several nodes of an execution tree may be equal. Thus, an LP l does not necessarily have to traverse its whole local execution tree in order to calculate its LLS.

¹Depending on the branching strategy, this includes information about created zero-delay events. We ignore zero-delay events here for the sake of clarity and refer to [20] for details.

We have developed two techniques to keep execution trees small by eliminating unnecessary duplicate computations: *Node comparisons* and *a-priori-knowledge*. Node comparisons are performed after every transition in the execution tree. When an event has been executed, the newly created node is compared to other nodes. If an equal node is found, the branches are merged together immediately. For the comparisons, each node is represented by a tuple $(E_{\text{exec}}, E_{\text{pend}}, E_{\text{new}}, S_l)$, with E_{exec} being the set of already executed simultaneous events, E_{pend} the set of simultaneous events yet to execute, E_{new} the set of events created during branching, and S_l the system state variables of l . The three event sets represent the current state of the scheduler and are always compared first. Only when they are equal, the (potentially large) system states are compared.

Node comparisons eliminate all duplicate states and allow for calculation of a set of unique LLS. However, they have two disadvantages: Comparisons may be costly if system states are large, and can only be performed *after* the execution of events. We therefore use an additional mechanism, *a-priori-knowledge* (APK), which is much cheaper in terms of computing resources, and can already be employed *before* event executions. The basic idea of APK is to store programmers' expert knowledge about possible event interactions permanently in a repository, and utilize it at simulation run time. Three types of event interactions may be specified: Two events e_1 and e_2 are *non-interacting*, if both execution orders $(e_1; e_2)$ and $(e_2; e_1)$ lead to identical states whenever these events occur simultaneously. Similarly, with *surely-interacting* events, it is guaranteed that both execution orders lead to different states whenever they occur simultaneously. Finally, with *possibly-interacting* events, both ways are possible, depending on the circumstances. The concept of APK was introduced as part of the sequential simulation framework in [2]. Since, in distributed simulation, events handled by different LPs are always non-interacting, we reduce APK to LP-scope. Let E_l be the set of all events which may be scheduled at LP l . Then we define the *local APK function* of l as $\text{APK}_l : E_l \times E_l \rightarrow \{N, P, S\}$. Table 1 explains the possible values of the APK function.

| $\text{APK}_l(e_1, e_2)$ | Interaction of events e_1 and e_2 |
|--------------------------|---|
| N | e_1 and e_2 are <i>non-interacting</i> |
| P | e_1 and e_2 are <i>possibly-interacting</i> |
| S | e_1 and e_2 are <i>surely-interacting</i> |

Table 1. Possible values of the APK function.

During the calculation of its local execution tree, each LP evaluates its local APK function. Whenever a subset of simultaneous events is found to be non-interacting, only one execution order is calculated. Moreover, surely-interacting events never require node comparisons. In fact, node comparisons only have to be performed after the execution of

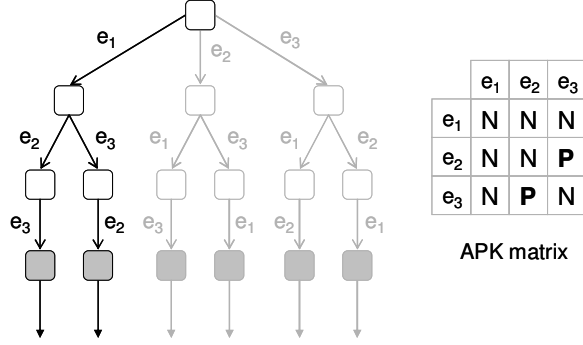


Figure 3. Using a-priori-knowledge.

possibly-interacting events. Thus, depending on the available amount of APK, many event executions and node comparisons are avoided. Figure 3 shows an example, with the APK function visualized as a symmetric matrix.

APK does not only prevent unnecessary transitions in the execution tree, but may also be used in conjunction with candidate events. When an LP detects candidate events, it can check their interaction before reporting them to the BM. If they are found to be non-interacting, the LP refrains from reporting them. Moreover, APK tables provide an elegant way to *declare* candidate events: By setting the APK function of pairs of candidate events to possibly-interacting, and all other event pairs to non-interacting, an exact specification of simultaneous events of interest is possible.

We conclude that APK usage ensures that any unnecessary overhead like, e.g., reporting non-interacting simultaneous events, comparing equal nodes, or creating duplicate branches, does not occur. This makes APK essential for an efficient analysis of simultaneous events.

3.3 Computation of branches

The techniques presented so far eliminate all branches that are duplicates or irrelevant to the user. Still, the actual computation of the remaining branches may become a major source of inefficiency. The branching mechanism, as shown in figure 2, stores global system states to hard disk and computes branches successively. In related application areas, however, cloning techniques have been shown to reduce simulation run-times when compared to independent successive replications. Therefore, we have integrated cloning into the branching mechanism in order to increase its efficiency. In the following, we will refer to our original branching mechanism as *classical branching*.

Cloning is triggered at decision points when one or more LPs are required to show different behavior, i.e. use differing event handling routines. All other LPs become shared LPs which perform common computations for different scenarios. Section 2.2 gives an overview of cloning techniques.

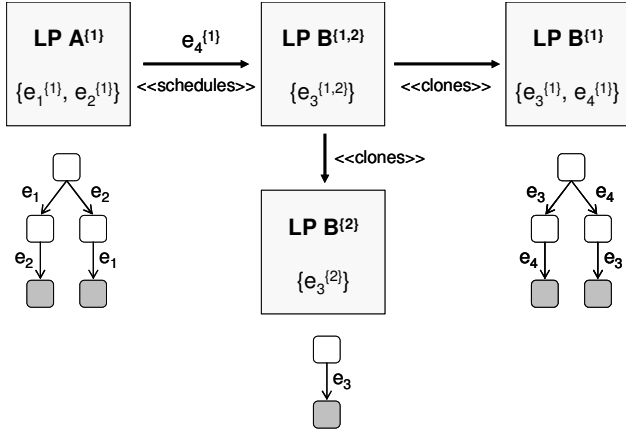


Figure 4. Passive cloning during branching.

A comparison of our branching mechanism with existing cloning mechanisms reveals both similarities and differences. On the one hand, branching points logically correspond to decision points, and both are treated conservatively so that the creation of new branches is never rolled back. However, there are also noticeable differences: With simultaneous events, it is not known beforehand whether new branches really have to be created, since the differences between alternative execution orders only show during the calculation of the combined execution tree. Furthermore, there are no different event handling routines.

We have chosen a solution based on an interface, which allows a seamless integration of cloning as an alternative to classical branching. At a branching point, when the combined execution tree has been calculated, the interface receives the CLS as well as the corresponding branch IDs. With classical branching, the save-and-restore approach is taken. With cloning, however, the active cloning algorithm (cf. section 2.2) is performed, based on the information contained in the CLS. This involves creating new physical LPs for all involved LPs, and setting all other LPs as shared clones accordingly. Note that, since an LLS may be contained in multiple CLS, the newly created physical LPs may still be shared clones at the same time. When active cloning is completed, all events created in the process of branching have to be extracted from the CLS and delivered to their destination LPs. Finally, the simulation continues, and all branches are simulated in parallel. In the further course of the simulation, passive cloning is performed based on message comparisons (cf. section 2.2).

In addition to sharing event computations among different scenarios, cloning has another potential advantage when compared to classical branching, since simultaneous events may occur in different scenarios at the same time. Now, if simultaneous events occur at shared LPs, the whole process of detecting, reporting, and analyzing them is per-

formed only once. During the branching process, however, additional care is required: When analyzing simultaneous events, the BM has to take into account to which scenarios they belong. First of all, this has to be reflected correctly when assigning branch IDs to newly created branches. Furthermore, passive cloning may be required in the process of branching. Figure 4 shows an example: The physical LPs $A^{\{1\}}$ and $B^{\{1,2\}}$ (a shared clone for branches 1 and 2) are about to calculate their LLS. However, during branching, $A^{\{1\}}$ generates a zero-delay event, e_4 , for virtual LP $B^{\{1\}}$. In order to preserve consistency, $B^{\{1,2\}}$ has to perform passive cloning upon reception of e_4 . Only then, the newly created physical LPs $B^{\{1\}}$ and $B^{\{2\}}$ may calculate their LLS. In this example, $B^{\{2\}}$ remains with a single event, e_3 , which it may simply execute.

3.4 Performance results

In this section, we present performance results which we obtained by replacing classical branching with the cloning mechanism described in section 3.3. We examined the speedup of cloning compared to classical branching in different simulation series. Apart from the different strategies to compute branches (i.e. cloning or classical branching), the respective simulation runs had identical setups. We ran multiple replications of all simulations, and all figures include confidence intervals with a confidence level of 95%.

Some aspects of cloning have already been evaluated extensively in related application areas (see e.g. [5, 11]). Generally, cloning has been shown to be most advantageous when the differences between alternative scenarios (i.e. the effects of simultaneous events) do not spread among many LPs. We have confirmed this result for our implementation. In the following, we will focus on another aspect of cloning which has not been evaluated so far. While the techniques presented in sections 3.1 and 3.2 are independent of the synchronization mechanism, this is not the case for the cloning technique. Therefore, we ran all simulations with different conservative and optimistic synchronization mechanisms, in order to examine their impact on the run-time of cloning in comparison to classical branching.

In order to examine the performance of cloning in a set of scenarios with well-defined behavior, we used the synthetic workload model described in [19]. The model consists of nodes sending events to each other. When executing an event with timestamp t , a node schedules a newly created event at another, randomly chosen node. The timestamp of the new event is randomly chosen from the interval $[t + 1, t + 10]$. Each node is mapped to its own LP.

For the evaluation of cloning, we extended the model so that we were able to specify simultaneous events at specific timestamps in advance. With regard to APK (cf. section 3.2), all simultaneous events were possibly-interacting, so

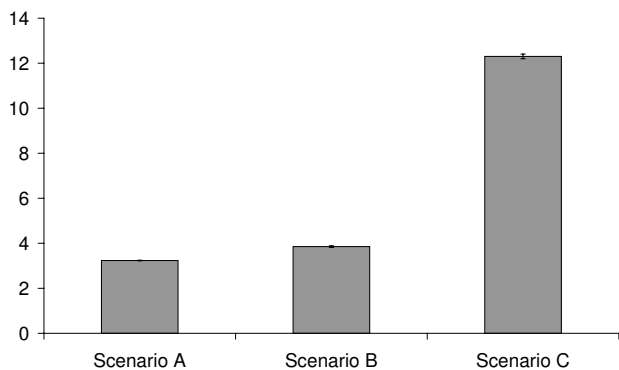


Figure 5. Speedup of cloning compared to classical branching (deadlock detection).

that execution trees had to be explored completely. All simulations were run for 1,000 units of virtual time. We made sure that this duration is long enough so that no shared LPs exist anymore at the end of the simulation. This avoids any bias in favor of the cloning mechanism. In the following, we present results for three simulation series:

Scenario A The model consists of 300 nodes with 3 initial events scheduled at every node. Furthermore, 3 simultaneous events are scheduled at one specific node at the beginning of the simulation. This results in 6 different branches which have to be computed.

Scenario B In addition to the configuration in scenario A, the model exhibits local behavior: The nodes are divided into 10 groups, and new events are scheduled at a node of the same group with probability 0.8, and with probability 0.2 at another node. Thus, the effects of simultaneous events spread slower among the nodes.

Scenario C In addition to the configuration in scenario A, another 3 simultaneous events occur at time 20.0. This leads to a total number of 36 branches.

Figure 5 shows the speedup obtained by cloning when a conservative deadlock detection scheme [4] is used. In scenario A, a speedup of 3.23 is obtained. Since the maximum possible speedup is equal to the number of branches, i.e. 6, this is an encouraging result. The cloning mechanism had already finished when classical branching was still computing the second branch. With local behavior, in scenario B, the performance of cloning is even better.

The most notable speedup, however, was reached in scenario C. Since the total number of branches is 36, a speedup of 12.3 means that classical branching was still computing the third branch when cloning had already finished. This is remarkable, especially if we take into account that no LPs were shared anymore at the end of the simulation runs.

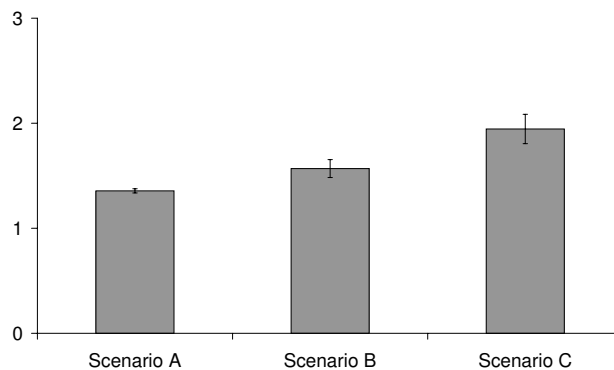


Figure 6. Speedup of cloning compared to classical branching (optimistic).

As figure 6 shows, with optimistic synchronization [13] the achieved speedup is not as large as with the deadlock detection scheme. Nevertheless, cloning performs clearly better than classical branching, and is approximately twice as fast as classical branching in scenario C. We expect the speedup to increase further when just-in-time cloning is used, which has been shown to increase the efficiency of cloning with optimistic synchronization (cf. section 2.2). With our current implementation, it is possible that passive cloning is performed prematurely and rolled back later.

We have also run the simulations with other synchronization mechanisms and noticed that, generally, cloning performs especially well when frequent global synchronization between the LPs is required. For example, with a completely synchronous mechanism, we achieved an average speedup of 28.1 in scenario C. Also, the conservative deadlock detection scheme (cf. figure 5) belongs to this category. We attribute this behavior to the fact that cloning increases the event population, and thus there are more events to execute between synchronization points than with classical branching. This conforms to the results obtained in [3] for entire cloning. In contrast to that, with an optimized conservative null message protocol [18], we have only achieved a marginal speedup of 1.04, since the increasing number of LPs also leads to a much larger number of null messages.

In summary, in our performance evaluation, cloning was always able to achieve speedup in comparison to classical branching. However, the achieved speedup varied strongly depending on the scenario and the synchronization mechanism. Nevertheless, our conclusion is that cloning should be used for the analysis of simultaneous events. Naturally, due to the higher number of LPs, cloning may require much more memory than classical branching. This holds especially for systems with many state variables. Therefore, combinations of both mechanisms should be examined.

4 Conclusions and future work

In this paper, we have proposed a branching mechanism for analyzing the effects of simultaneous events in distributed simulation. In particular, we have presented various techniques which increase the efficiency of branching. These include methods for reducing the number of branches as well as the general overhead of branching. Furthermore, we have incorporated cloning techniques in order to compute branches efficiently. Our performance evaluation has shown that cloning reduces simulation run-times strongly when compared to the classical method of computing branches as independent replications. Altogether, this makes branching a practical method for analyzing simultaneous events.

In future work, we are going to examine ways of combining cloning and classical branching. Although cloning generally reduces the run-times of simulations, classical branching should be used when the available memory gets low. Furthermore, we are going to examine the interaction of cloning with partitioning algorithms. Since the cloning of LPs may lead to load and communication imbalances, dynamic partitioning of an otherwise well-balanced system may be required. Here, the question is whether standard dynamic partitioning algorithms are able to deal with cloning successfully, or whether they have to be made aware of it.

An interesting question is whether the APK technique we developed for branching can be generalized and applied in other areas, too. Although it is easier to determine the interaction of events with identical timestamps, it is also possible to specify general APK tables, i.e. for events with different timestamps. One potential application area for such a generalized type of APK is optimistic synchronization, where knowledge about non-interacting events could be used to reduce the number of rollbacks.

References

- [1] A. Agarwal and M. Hybinette. Merging parallel simulation programs. In *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation*, 2005.
- [2] C. Barz, R. Göppfarth, P. Martini, and A. Wenzel. A new framework for the analysis of simultaneous events. In *Proceedings of the 2003 Summer Computer Simulation Conference*, 2003.
- [3] L. Bononi, M. Bracuto, G. D'Angelo, and L. Donatiello. Concurrent replication of parallel and distributed simulations. In *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation*, 2005.
- [4] K. M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, 24(4):198–206, 1981.
- [5] 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.
- [6] B. A. Cota and R. G. Sargent. Simultaneous events and distributed simulation. In *Proceedings of the 1990 Winter Simulation Conference*, 1990.
- [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] M. Hybinette. Just-in-time cloning. In *Proceedings of the 18th Workshop on Parallel and Distributed Simulation*, 2004.
- [10] M. Hybinette and R. Fujimoto. Cloning: A novel method for interactive parallel simulation. In *Proceedings of the 29th Conference on Winter Simulation*, 1997.
- [11] M. Hybinette and R. M. Fujimoto. Cloning parallel simulations. *ACM Transactions on Modeling and Computer Simulation*, 11(4):378–407, 2001.
- [12] *IEEE 1516-2000 - Standard for Modeling and Simulation High Level Architecture - Framework and Rules*, 2000.
- [13] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, 1985.
- [14] V. Jha and R. Bagrodia. Simultaneous events and lookahead in simulation protocols. *ACM Transactions on Modeling and Computer Simulation*, 10(3):241–267, 2000.
- [15] T. Kiesling, J. Lüthi, and R. E. A. Khayari. Bias in parallel and distributed simulation systems. In *Proceedings of the 37th Conference on Winter Simulation*, 2005.
- [16] K. H. Kim, Y. R. Seong, T. G. Kim, and K. H. Park. Ordering of simultaneous events in distributed DEVS simulation. *Simulation Practice and Theory*, 5(3):253–268, 1997.
- [17] H. Mehl. A deterministic tie-breaking scheme for sequential and distributed simulation. In *Proceedings of the 6th Workshop on Parallel and Distributed Simulation*, 1992.
- [18] J. Misra. Distributed discrete-event simulation. *ACM Computing Surveys (CSUR)*, 18(1):39–65, 1986.
- [19] 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*, 2007.
- [20] P. Peschlow and P. Martini. Analyzing simultaneous events in distributed simulation. In *Proceedings of the 19th European Modeling and Simulation Symposium*, October 2007.
- [21] P. L. Reiher, F. Wieland, and P. Hontalas. Providing determinism in the time warp operating system - costs, benefits, and implications. In *Proceedings of the 2nd IEEE Workshop on Experimental Distributed Systems*, 1990.
- [22] R. Rönnngren and M. Liljenstam. On event ordering in parallel discrete event simulation. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, 1999.
- [23] 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*, 2006.
- [24] A. Wenzel. Experiences with simultaneous events using discrete-event simulation. In *Proceedings of the IASTED International Conference on Modeling and Simulation*, 1999.
- [25] F. Wieland. The threshold of event simultaneity. In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, 1997.