Practical Tie-Breaking for Parallel/Distributed Simulations

Andrea Piccione
Huawei Munich Research Center
andrea.piccione@huawei.com

Alessandro Pellegrini
Tor Vergata University of Rome
a.pellegrini@ing.uniroma2.it

***Abstract*—In this paper, we discuss a tie-breaking strategy based on a bitwise comparison of event payload that allows parallel and distributed discrete-event simulations to observe a deterministic order in the execution of events, even in the presence of event ties. This approach provides practical usability whenever model-assisted tie-breaking is unavailable, thus ensuring that multiple simulation executions provide deterministic behaviour and repeatable results. Moreover, it ensures that the selected order of events is also consistent with sequential executions. We discuss the theory behind this strategy and experimentally show that the performance drop is imputable to event queue management when relying on tie-breaking strategies like the ones discussed in this work.**

1. Introduction

Simultaneous events [6] are a relevant problem for discrete-event simulation (DES). In the physical world, the occurrence of simultaneous events is something that the physical system can handle autonomously by its very nature. When attempting to translate the workings of the physical world into a simulation model, the modeller must explicitly handle simultaneous events. If simultaneous events are not additive and commutative, processing them in a different order may lead to different simulation results. In some cases, choosing an incorrect order may lead to error conditions that cause the simulation to fail. This concept is well understood in the parallel/distributed simulation community [16], [21], but in the case of simultaneous events, such error phenomena can also occur in purely sequential simulations.

The only way to ensure that a simulation is correct in the case of simultaneous events is to entrust the modeller, through the use of a *tie-breaking* function, with the task of finding a correct ordering for the invariants of the portion of the world one is trying to simulate. For example, the works in [3], [8] theorise the need to handle *sets of simultaneous events* at the model level. Providing the event dispatcher with a set makes it possible to ask the model to deal with them in a manner consistent with its characteristics.

In the case of optimistic parallel/distributed simulations [7], this approach may be hard to implement. The speculative nature of forward execution may still lead to handling sets of events when they are not fully formed because an antimessage could be received before the corresponding positive one. However, in the case of sequential simulation, this method may be sufficient to handle correctness, except when some events in a set with the same timestamp generate a new simultaneous event. From a practical point of view, this interaction pattern is important, as it allows for the implementation of *sensing capabilities* in the models [5], which are relevant, e.g., for agent-based simulation [1]. This is so relevant that, in seminal Time-Warp works, Jefferson et al. [9] devised the concept of *query messages* to solve the dichotomy between event sets and the need to access portions of the simulation state in read-only.

The scenario in which the modeller has provided a tie-breaking function capable of correctly handling simultaneous events may be ideal. We may assume that it is available only when the simulation model is stable, complete, and correct. Indeed, in the life cycle of a simulation model [18], [22], the modeller may prioritise developing the core dynamics of the model and delay implementing the tie-breaking function. Similarly, if the model undergoes evolutionary maintenance or is integrated into a simulation of simulations to reuse existing models in larger models, the tie-breaking function may need to be modified to restore correctness properties, which may be time-consuming. In this case, the modeller may decide to suspend the use of the current tie-breaking function and redesign it later.

In this dynamic perspective on models, a significant contrast emerges between parallel and sequential executions of the same model. Sequential models, despite the occurrence of simultaneous events, can possess the unique quality of exhibiting *deterministic* executions. On the contrary, when the model is executed on parallel or distributed architectures, two executions may yield different outcomes if simultaneous events are not properly managed. This holds true even when employing the same seed to configure pseudo-random number generators for stochastic simulations.

This situation presents a clear problem. The modeller may find it necessary to execute numerous runs with identical configurations in order to compare simulation results for purposes such as debugging or performance evaluation. In the case of parallel or distributed simulations, runtime environment developers are thus compelled to explicitly manage simultaneous events, even if the modeller has not specified a model-oriented tie-breaking function.

In this paper, we explore theoretically and experimentally a practical tie-breaking function based on a bitwise comparison of the event payload that ensures the reproducibility and repeatability of executions. Unlike existing solutions in the literature, this approach enables the incorporation of domain information directly into the simulation model if required by the modeller. Consequently, the established ordering ensures that ambiguity between two simultaneous events arises only when the model itself cannot differentiate between these distinct events, even at the application level.

Specifically, our analysis reveals that, in parallel/distributed execution, any uncertainty regarding the ordering of two events only occurs when the sequential model is also faced with the task of determining the order between two perfectly identical events. Notably, the discussed tie-breaking strategy maintains the same event ordering regardless of whether the model is executed sequentially or in parallel, even when employing optimistic approaches. Therefore, deterministic executions can be achieved even in the case of parallel/distributed executions in the absence of tie-breaking functions at the model level. This strategy may suffer from incomplete information related to the delay in receiving certain events. However, this phenomenon is inherent in the concept of speculative simulation *aggressiveness* and can therefore be solved by techniques already present in the literature [9].

The remainder of the paper is structured as follows. In Section 2, we formalise the problem we are dealing with and provide useful insights into our methodology. Related work is discussed in Section 3. We present the tie-breaking strategy in Section 4 and report the results of our experimental assessment in Section 5.

2. Problem Statement

In a DES, whether sequential or parallel, the simulation state is updated by the execution of events that are the *atomic unit of processing*. An event has an all-or-nothing nature: it must be executed entirely or not at all. Therefore, the simulation state undergoes a set of updates each time an event is processed. Given a simulation that begins with an initial state S_0, the execution of an event e_1 at simulation time t_1 can be seen as the application of a transition function f that produces a state update, i.e.:

$$S_1 = f(e_1, S_0) \tag{1}$$

The succession of events leads to the final simulation state S^* through an iterative application of f:

$$S^* = f(e_n, \ldots, f(e_2, f(e_1, f(e_0, S_0)))) \tag{2}$$

The order in which events are executed is determining to attain a terminal state of the simulation of interest. Given a sequence of events $e_1, e_2, \ldots, e_n$, reversing the execution of two events e_i, e_j with $i \neq j$ can cause an alteration in the trajectory of the simulation. Indeed, if the events e_i and e_j are not *commutative with respect to the function* f, i.e. if $f(e_j, f(e_i, S_h)) \neq f(e_i, f(e_j, S_h))$, it is easy to observe that after a sequence of m events we have that:

$$\begin{aligned} S_m = f(e_m, \ldots, f(e_j, f(e_i, \ldots, f(e_1, f(e_0, S_0)))))) \neq \\ S'_m = f(e_m, \ldots, f(e_i, f(e_j, \ldots, f(e_1, f(e_0, S_0))))) \end{aligned} \tag{3}$$

In the classical DES approach, the presence of a data structure known as the *future event set* (FES) is essential. The FES is commonly implemented as a priority queue,

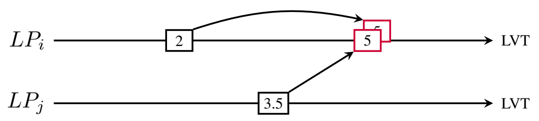


Figure 1: Lack of strict total ordering. The two events at logical time 5 should "happen" at the same time: it is unspecified which of the two to dequeue first from the FES.

allowing queries to extract the event with the highest priority. Due to the temporal nature of simulations, priority is typically associated with simulation time.

The concept of event simultaneity refers to a situation in which, at a particular timestamp $\bar{t}$, there exist two (or more) events that need to be processed. This presents a challenge when it comes to extracting events from the FES since two events, e_1 and e_2, are associated with identical timestamps $t_1 = t_2 = \bar{t}$. In such cases, there is no strict total ordering between e_1 and e_2 based on logical time alone. This scenario is illustrated in Figure 1. Here, we consider the problem of *tie-breaking* as the need to reconstruct a deterministic order over all events, even when two or more are associated with the same timestamp, formally:

Definition 2.1 (Tie-Breaking Oracle)**.** The operation of extracting the next event from the Future Event Set is entrusted to an oracle O which determines the next event e to be executed to obtain a final simulation state S^* consistent with the behaviour of the real-world system being simulated.

It is evident that the model's dynamics cannot be disregarded when defining the oracle O. As a result, it has been acknowledged in the literature [8] that the modeller must provide a tie-breaking function to ensure the proper implementation of O, especially when dealing with noncommutative events. However, there may be situations where the oracle O is unavailable due to factors related to the simulation development cycle or the evolution/composition of the model. From the perspective of model execution support, it is still necessary to establish a deterministic total order for the events in the FES.

We note that the output of a correct simulation must be necessarily deterministic, even in the case of ties. Sequential simulations suffer less than parallel/distributed simulations. Indeed, determinism is a common property of FES implementations; however, if the data structure employed for the FES is unstable with respect to sorting, then even a serial simulation may be nondeterministic. However, the deterministic ordering that occurs during a sequential simulation may not necessarily represent the characteristics of the physical system being investigated. Therefore, for sequential simulations, Definition 2.1 can be relaxed by introducing the concept of *partial oracle* P according to the following

Definition 2.2 (Partial Tie-Breaking Oracle)**.** The operation of extracting the next event from the Future Event Set is entrusted to a partial oracle P, which determines the next event e to be executed so as to obtain a sequence of events $\{e_0, e_1, \ldots, e_n\}$ to reach a final state of simulation $\widetilde{S}^*$ such that $e_0 \prec e_2 \prec \ldots \prec e_n$

where $\prec$ is some total order defined by the oracle on the sequence of events. For what we have discussed above, even if the FES is stable, it is possible that $\widetilde{S}^* \neq S^*$—hence the partiality of the oracle P. Anyhow, Definition 2.2 has an important property related to the reproducibility and repeatability of executions in sequential simulations with a stable FES. Since the oracle P chooses a sequence of events in a deterministic manner, two different executions configured exactly the same way will lead to the same final state of simulation $\widetilde{S}^*$. Any pseudo-random number generator seeds in the case of stochastic models must also be included in the model configuration.

We argue that, in the case of parallel/distributed simulations, emulating the partial oracle P, according to Definition 2.2, is complicated and also not useful. In fact, the behaviour of P depends on the characteristics of the sequential simulation *implementation*; in other words, it is biased in a way that is not representative of the model dynamics. Still, we are left with the problem of parallel nondeterministic simulations. In fact, two different parallel executions may lead to the scenario shown in Figure 2. Here, we are depicting an extremely simple scenario in which two users (modelled as two distinct LPs) contact the same service over a network. Both requests arrive at the same timestamp ($t = 2$). Assuming that the server has a delay of 1 virtual second to process a request, the order in which they are received affects the response time. Imagine that user A can tolerate a delay of up to 2 seconds, while user B can tolerate a delay of up to 3 seconds. In the case of the scenario in Figure 2b, the result of the simulation would be that the configuration does not respect the model's invariants, while in the execution in Figure 2c, the solution would be acceptable. To illustrate the bias of the partial oracle P, it is worth noting that *one* serial execution engine might consistently and deterministically generate the output shown in Figure 2b, while *another* serial engine implementation could consistently and deterministically replicate the situation depicted in Figure 2c.

The processing and commit order of the events depend on many factors *external to* the simulation, such as scheduling dynamics at the operating system level or network latencies. Therefore, to be able to guarantee the reproducibility of executions even in parallel/distributed simulations, it is necessary to find a strategy that does not rely on local implementation properties such as the partial oracle P of Definition 2.2.

3. Related Work

The handling of simultaneous events is an important topic for DES that has received a lot of attention from the community. Interestingly, the seminal contribution [11] does not consider the problem of simultaneous events from the point of view of models that may suffer from non-commutativity in updating states. In fact, concerning simultaneous events, Lamport merely states that *two contemporaneous events do not have an impact on mutual causality*. However, while not creating causality problems, two simultaneous events may lead to correctness issues with respect to model characteristics, possibly leading to errors in results or crashes in simulation [16]. In [7], the problem of simultaneous events is already made apparent in the context of optimistic simulation. Indeed, the contribution showed how, for concurrency control simulations in distributed databases, it is possible to use the event source to resolve ties between events.

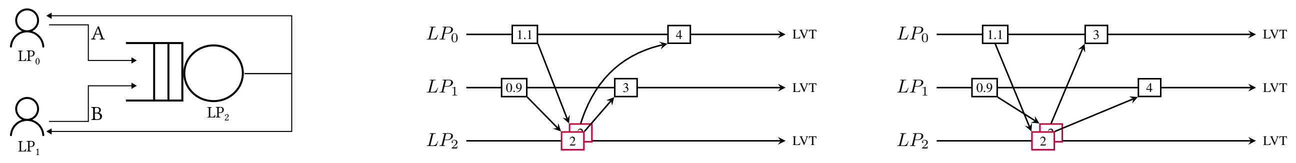
Several proposals have addressed the construction of a deterministic ordering based on additional bits used alongside the timestamp [2], [10], [12], [13], [23], [24]. The main difference between these proposals is *from whence* these bits were taken. In [13], two fields are appended to the application-defined time stamp, called the age and id; in [10], a simulator priority and an event level (similar to the age above) are used to add an ε-delay to the timestamp[1]; in [23] the additional bits are based on user rankings and a Lamport clock [11]; in [2] Hyperreal numbers are used to determine infinitesimal time instants that can be used to extend the standard notion of time; in [24] the timestamp is coupled with a data structure that allows the modeller to control the event order; in [12] controllable and deterministic random-number generators are used. All these solutions, independently of the level of transparency provided to the modeller, can suffer (with different probability and under different scenarios) from *collisions* that can make the ordering of events nondeterministic. Conversely, in our proposal, such collisions are only observed in scenarios where the simulation model trajectory would be totally unaffected. Therefore, our solution remains deterministic also in the case of collisions unless the model requires to explicitly deal with *superposition effects*, in which case any form of transparent platform-level tie-breaking cannot serve as a solution to the problem.

Since the approach in [12] exploits controllable random-number generators, it also allows exploring multiple ordering of events, thus enabling a broader statistical analysis of models. This approach was also envisaged in previous work [19], where the importance of studying the outcome of multiple ordering of simultaneous events was highlighted. Another relevant contribution from [12] is a variant of the PHold benchmark [4] to study the behaviour of ties. We leverage this model to test our implementation. In general, we share the reproducibility and repeatability goals of [12], although we follow a different path to tie-breaking.

The work in [23] also discusses the implications of ties broken at the platform level when there are zero-look-ahead messages. It identifies possible sources of causality incorrectness. The problems of zero-lookahead in the context of the High-Level Architecture (HLA) are discussed in [5]; we take care of them in Section 4.1.

In [25], the author claims that, in the case of simultaneous events, it would be correct to present averaged results over all possible orderings. This kind of exploration can be difficult or significantly expensive for large-scale simulations. Moreover, the goal differs from ours, as we

1. We note that, while ensuring deterministic executions in parallel scenarios, changing the timestamp of an event could deviate the results from the sequential simulation.



(a) Simulated Scenario. (b) First Possible Execution. (c) Second Possible Execution.

Figure 2: Inconsistent Runs in a Parallel/Distributed Simulation.

strive to enforce reproducible and repeatable simulations rather than exploring the space of possibilities.

Some proposals [8], [20] explicitly demand from the modeller a solution to ties by delivering to the model sets or bundles of events. The model must explicitly handle such sets and process the events according to the model's logic. Similar approaches can be found in [3], where *transition collisions* are considered a model's responsibility and are handled using the user-defined *confluent transition function*. While completely non-transparent, these approaches also allow accounting for the cumulated effects of tying events, which is impossible when ties are broken at the simulator level. In practise, these works advocate for the need for the Oracle O as defined in Definition 2.1. Clearly, this approach contrasts the practical goals of our proposal, especially when the implementation of a model is unstable.

In [21], the authors discuss the tie-breaking functions in the Time Warp Operating System (TWOS). Although this descriptive work hints that the authors perform a bit-wise comparison between event payloads to construct event sets, few methodological and no implementation details are provided. This lack of detail has led the work in [23] to conclude that the approach is not correct. With the theoretical framework that we present in this paper, it becomes clear how and when practical tie-breaking schemes, such as the one we discuss, are correct and viable. Moreover, we clarify why the event class (or type) is not necessary for tie-breaking, differently from [21].

4. The Practical Tie-Breaking Technique

The tie-breaking mechanism that we propose and study in this paper is based on the intuition that if two events are *indistinguishable* by the simulation model, they can be executed in any order without affecting the final state of the simulation. The same concept also holds for sequences of indistinguishable events. To better formulate our intuition, we provide a definition of ties that is compatible with the logical framework discussed in Section 2. Let e be a generic event injected into the system, associated with the timestamp t_e at which the event must be executed. A timestamp-based tie follows the definition below.

Definition 4.1 (Timestamp-based Scheduling Equivalence)**.** Two events e_1 and e_2 are *scheduling equivalent*, namely $e_1 \sim e_2$, if $t_{e_1} = t_{e_2}$.

The goal of the Partial Oracle P discussed in Definition 2.2 is therefore to *enhance* the scheduling equivalence from Definition 4.1 by enforcing an ordering $\prec_P$ that comes from the rules enforced by P. Part of the body of work discussed in Section 3 has effectively tried to extend this notion of timestamp-based scheduling equivalence by defining an ordering $\prec_P$ that can be applied to a set of events e_i such that $t_{e_i} = t_{e_j} \forall i \neq j$. The simplest definition of $\prec_P$ that has been considered in the early literature and in various early implementations of PDES runtime environments extends the previous definition of events, in various forms. In particular, we can consider an event e as a tuple $e = \langle t_e, c_e, s_e, d_e \rangle$, where c_e is the *event class* (also referred to as event type), s_e is the sender of the event and d_e is its destination. We can practically assume that we can build some (lexicographic) ordering on c_e, s_e, and d_e. Indeed, in many implementations, these elements of the tuple are already numbers, but it is straightforward to define some mapping to $\mathbb{N}$ or $\mathbb{R}$ that can define an ordering over the values. Then, we can enhance Definition 4.1 as follows:

Definition 4.2 (Enhanced Scheduling Equivalence)**.** Two events e_1 and e_2 are *scheduling equivalent*, namely $e_1 \sim e_2$, if $t_{e_1} = t_{e_2}$, $c_{e_1} = c_{e_2}$, $s_{e_1} = s_{e_2}$, $d_{e_1} = d_{e_2}$.

There are two important implications of Definition 4.2. First, such a definition of equivalence may create a bias with respect to the choices that the tie-breaking Oracle O of Definition 2.1 might make. Indeed, regardless of the ordering of the elements, P could choose differently from O since, by definition, P is unaware of the dynamics of the modelled system. For our practical purposes, this bias might be tolerable: P is not intended to be a *correct* replacement of O, but an *acceptable approximation* if O is not available.

The strongest implication, however, is that $\prec_P$ defined in accordance with Definition 4.2 is a weak total order. Therefore, using $\prec_P$ defined in this way does not solve the problem at all since it is still possible to find two events $e_1 \sim e_2$ such that the scheduler of the runtime environment is unable to make a deterministic choice, even if their processing order could impact the simulation trajectory. Since the goal of our approach is to support reproducibility and replicability, Definition 4.2 is not sufficient.

As mentioned, several of the works discussed in Section 3 have addressed this problem by implicitly providing an extension of Definition 4.2 that allows this problem to be addressed. In particular, it is possible to construct an ordering $\prec_P$ that imposes a deterministic ordering by extending the definition of the event e. Indeed, we can consider an event e as a tuple $e = \langle t_e, c_e, s_e, d_e, b_e \rangle$, where b_e are *arbitrary bits* provided by the model developer that implicitly describe the priority of e over other tying events. Abiding by this definition, we can construct an improved ordering $\prec_P$ leveraging the following

Definition 4.3 (User-Defined Enhanced Scheduling Equivalence)**.** Two events e_1 and e_2 are *scheduling equivalent*, namely $e_1 \sim e_2$, if $t_{e_1} = t_{e_2}$, $c_{e_1} = c_{e_2}$, $s_{e_1} = s_{e_2}$, $d_{e_1} = d_{e_2}$, $b_{e_1} = b_{e_2}$.

At first sight, this definition should solve the problem of ties. Indeed, the modeller can set the value of b_e arbitrarily, thus deciding what is the precedence between events. However, in our reference scenario, this strategy is highly objectionable for two reasons. The first concerns transparency towards the modeller: if they are asked to provide additional information to define an Oracle P that succeeds in resolving any remaining ties, the authors of this work then wonder why this strategy is better than explicitly requesting that the Oracle O be realized directly. Indeed, as discussed in [8], the quality of O is clearly superior to that of P, from a modelling perspective. Reasoning on this aspect, a solution defining an event e as the simpler tuple $e = \langle t_e, b_e \rangle$ may also be a better solution, as it would eliminate the artificial bias introduced by Definition 4.2. Moreover, in our reference scenario, we consider the model as an evolving object, so the properties according to which b_e should be valued could easily change. Therefore, this is not a practical, life-cycle-oriented approach to models.

As mentioned, this problem related to transparency towards the modeller was addressed in [12] by deciding to entrust the values of b_e to a pseudo-random number generator with special properties. This way, it is both possible to explore alternative simulation trajectories and obtain reproducible executions without bothering the modeller. We follow an alternative path in our proposal as we reason about indistinguishability between events.

Two events are actually indistinguishable if they expose the exact same information to the simulation model. To better formulate this concept, we redefine an event e as the tuple $e = \langle t_e, p_e \rangle$, where p_e is the event's payload[2]. We can then define indistinguishable events as follows.

Definition 4.4 (Indistinguishable Events)**.** Two events e_1 and e_2 are indistinguishable, namely $e_1 \mathrel{\overset{?}{=}} e_2$, if $t_{e_1} = t_{e_2}$, and $p_{e_1} = p_{e_2}$.

According to this definition, two events are indistinguishable when their timestamps and payloads are bitwise identical. We do not consider the pair of LPs involved and the event's class as a necessary part of the event payload, as, in the general case, they may not be needed by the model logic. Moreover, if we take into account indistinguishable events, it is not important to make a deterministic choice as to which events execute first on different LPs: indeed, given the concurrent nature of Time Warp, it is sufficient to provide local guarantees—we discuss aspects of cross-LP causality in Section 4.1.

It is interesting to ask what effects at the model level such indistinguishability may entail. We note that the following property must hold.

2. We consider the event payload as the information that is included in an event message, i.e. the collection of all the event properties observable by the model.

Property 4.1. *Regardless of the order in which a sequence of indistinguishable events is processed, the simulation result is unchanged.*

If, for a sequence of events, changing their processing order changes the result of the simulation, that implies that at least a pair of events $e1, e2$ in the sequence are distinguishable by the model. In other words, a model could use the difference in the observed information in $e1, e2$ to order them deterministically. This choice would be part of the perfect oracle O, but in its absence, we will show that a simulation engine can always take a deterministic, although biased, choice.

Finally, relying on Definition 4.4, we can build a total ordering $\prec_L$ in the following way:

Definition 4.5 (Lexicographic Tie-breaking)**.** Given two events e_1 and e_2, $e_1 \prec_L e_2 \Leftrightarrow t_{e_1} < t_{e_2} \vee (t_{e_1} = t_{e_2} \wedge p_{e_1} \leq p_{e_2}$

From the point of view of the simulation engine, the payload comparison does not need to understand the semantics of its content, i.e., a simple bitwise comparison is sufficient to always break ties[3]. We observe that, $e_1 \prec_L e_2 \wedge e_2 \prec_L e_1$ if and only if $e_1 \mathrel{\overset{?}{=}} e_2$; in other words, our total order is unable to order two events only if they are indistinguishable. This means that $\prec_L$ defines an order of events that deterministically induces the same simulation trajectory. Our approach also has the valuable property of being independent of the initialisation order of the LPs. On the contrary, employing an ordering based on any scheduling equivalence defined above can result in a scenario where the initialisation order determines the order of tied events scheduled by LPs during initialisation. This makes the outcome of the whole simulation dependent on the order of evaluation of LPs.

Model developers sometimes unknowingly rely on the ordering implicitly guaranteed by some scheduling equivalence property, which can lead to models that subtly depend on this behaviour to function correctly. For instance, in an agent-based model, an agent departure and return may be scheduled at two randomly-sampled times t and $t + \delta t$, respectively. If $\delta t = 0$, this would mean that the logical dependence of the two events is encoded only in their scheduling order. More complex interactions can lead to difficult-to-debug issues where the model works correctly in a sequential simulation but crashes in parallel execution.

By using our tie-breaking strategy, we can ensure that a model functions consistently in both sequential and parallel execution modes. If the sequential simulation that uses our tie-breaking strategy runs correctly, then—assuming that speculative simulation trajectories do not cause irreparable side effects [16]—the parallel execution will also be correct.

4.1. Handling Cross-LP Causality

To understand the implications of our tie-breaking strategy, we have to discuss the implications of our approach when cross-LP interactions are observed. There are two

3. If the modeller wants to enforce a particular ordering, it is still possible by tweaking the content of the payload.

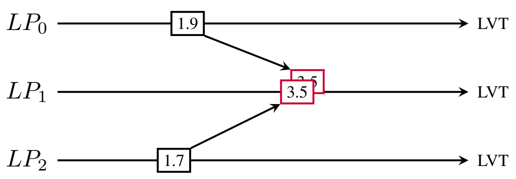


Figure 3: Simultaneous events by different LPs.

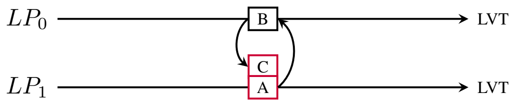


Figure 4: Zero-Lookahead Cycle.

cases of interest here. The first is depicted in Figure 3, where one LP is the target of two simultaneous events generated by two different LPs. Our tie-breaking strategy leaves no doubt in this case. If differences between events can be identified, the ordering will be well-defined and reproducible. If the events are indistinguishable, then Property 4.1 holds, and the order in which these events are executed is irrelevant: the simulation results will be unchanged.

A more interesting implication of our practical tie-breaking methodology relates to *zero-lookahead cycles*. Let us consider the scenario depicted in Figure 4. Here there is a circular causal dependence A $\rightarrow$ B $\rightarrow$ C, which involves two different LPs. Let us consider the most difficult problem, i.e. when events A and C have empty payloads and the other properties are the same. We observe that for a correct simulation, we must have that $A \prec_L B$ and $B \prec_L C$. Also, our ordering $\prec_L$ defined according to Definition 4.5 is not able to make a choice, i.e.: $A \doteqdot C$, so we must also have $B \doteqdot C$ and $B \doteqdot A$.

According to our defined total order, C would not be a straggler for A. This property can be generalized, thus stating that, if an LP has executed an event e_1 and subsequently receives an event $e_2 \doteqdot e_1$, the reception of e_2 must not cancel the execution of e_1. Apparently, this can be regarded as contrary to the concept of reproducible and repeatable execution. Once again, thanks to Property 4.1, we consider the two executions (e_1 before e_2 and e_2 before e_1) to be perfectly equivalent with respect to the final result of the simulation.

Interestingly, Property 4.1 can also be applied to relax the implementations of the data structures used for the FES. Indeed, should A and C be reprocessed silently due to the receipt of a straggler, it is irrelevant whether they are reprocessed in the order A, C or in the order C, A. This is a strong implication of Property 4.1: indeed, should the events be reprocessed in the order C, A we would be faced with a *causality violation that can be ignored*.

If, on the other hand, the two events are distinguishable, then a deterministic order can be imposed to decide if the execution is consistent or not. The reasoning we have just set out applies to zero-lookahead cycles, and by extension, it can be applied to any form of zero-lookahead events. Therefore, we can exploit the definition of indistinguishability between events to solve the management of zero-lookahead events locally as well.

4.2. Implementation Details

One advantage of our proposed tie-breaking strategy is that it can be evaluated completely locally to the LP, therefore making it suitable for parallel and distributed simulations. Implementing this approach in an existing simulation engine only requires a few changes. In conservative simulations, the scheduling policy needs to be expanded so as to consider the bitwise comparison of the payloads in case of event ties. In optimistic simulations, it is also necessary to include the extended tie-breaking logic in the straggler detection mechanism. No other significant changes are needed.

One important caveat to note with this tie-breaking method is that in practical implementations, especially in low-level languages like C, the determinism of bitwise comparisons can be skewed by uninitialized data contained in the event payload. This issue can arise due to several reasons, such as the model failing to initialize some members of the event payload, the presence of padding bytes in the event payload structure, or the model writing uninitialized bytes in the event payload.

Although addressing these issues may demand effort from the model developer, there are several straightforward strategies that can be useful for their mitigation. For example, padded data structures can be detected at compile time using GCC's `-Wpadded` flag. During debugging, tools such as Valgrind [15] can be employed to verify whether a tie has been broken due to uninitialised bytes. Programming languages may offer additional alternative solutions to some of the issues mentioned earlier. For example, in C++, a padding-free comparison can be implemented using template meta-programming techniques such as those provided by the Boost.PFR library.

4.3. Relations with Other Tie-Breaking Schemes

Breaking ties by comparing event payloads bitwise is effective but may result in an event order that is not meaningful model-wise. In addition, as discussed above, modellers are accustomed to the use of event classes to handle potential ties. However, this problem can be solved easily. Indeed, it is sufficient for the modeller to insert into the model payload the additional information (such as the event class) that it intends to be used to break the ties. By doing so, we can elegantly solve the problem outlined earlier with the agent-based model by assigning two different classes to *depart* and *return* events. Our experience with model development suggests that prioritizing specific event classes over others can already solve most of the issues related to tied events.

From this discussion, it is therefore clear that our practical tie-breaking approach is not at odds with other system-level solutions to solve this problem. In fact, it can be viewed as a generalisation of those approaches. Any of the strategies based on Properties 4.1–4.3 (and proposals based on additional bits discussed in Section 3) can be re-implemented within an ordering defined according to our strategy.

However, there are two ways to materialise these relations. The simplest involves realising the change at the runtime environment level: the environment developer can transparently insert within events any information they wish to be considered by the tie-breaking approach we have presented. With this strategy, different properties can be guaranteed transparently to the modeller.

Nevertheless, this possibility poses a problem as it introduces a bias that the modeller may not be aware of, similar to many methods discussed in Section 3. In contrast, our strategy avoids introducing biases stemming from simulation engine details. Therefore, the second way is to let the modeller explicitly include in the event payload as much information as they deem useful for ordering the events, with proper encoding. In this way, it is possible to create simulation environments that are anyhow correct and support repeatable and replicable executions. The model developer still has control over the choices made by the simulator.

Conversely, if a model is unaware of the attributes of our tie-breaking policy, it will not enhance the events with tie-breaking information and will not establish a meaningful order among the members of its event payloads. In such a scenario, tie-breaking is biased, but this stems solely from the way the model generates events, and the resulting ordering still remains deterministic. In this sense, the practicality of our approach is greater, as it allows transparent and non-transparent approaches to be combined in a single implementation.

5. Experimental Assessment

For our experimental assessment, we implemented the proposed tie-breaking strategy in ROOT-Sim [17], a parallel/distributed discrete event simulator. Our evaluation focuses on two aspects: the performance impact and the effect on model accuracy when compared to a version of the simulator that treats tied events as part of the same equivalence class, resulting in their execution in any order they are delivered. The experimental evaluation was run on a machine equipped with two Intel® Xeon™ e5-2699v4 processors @2.0 GHz, each consisting of 22 physical cores and 44 hyperthreads, for a total of 44 physical cores and 88 hyperthreads. It has 256 GB of RAM, even though the maximum size of the resident set utilized by the runs is approximately 42GB.

5.1. Testbed Applications

We evaluated three models and aimed to use configurations similar to [12]. Table 1 provides an overview of the four model configurations. Results are averaged over 20 runs, with the highest observed coefficient of variation being approximately 0.05, indicating reasonably reliable results.

The first considered model is the well-known PHold synthetic benchmark [4]. It models a simple communication pattern where LPs send messages to other LPs at random

TABLE 1: Model configurations.

Model	#LPs	Remote events	Committed events
PHOLD	2^{21}	10%	$\sim 527M$
ETIES-easy	65536	50%	$\sim 393M$
ETIES-hard	131072	10%	$\sim 1100M$
TBC	16384	$\sim 80\%$	$\sim 739M$

times. The model is characterized by its simplicity and scalability, making it a popular choice for testing parallel simulation frameworks.

The *event-ties* (E-TIES) model is a synthetic benchmark that investigates how simulation engines handle large volumes of tied events. It operates in rounds that are triggered every unit of virtual time, where each LP starts one or more chains of tied events. If an LP receives an event as part of a chain, it can extend it or terminate it if a pre-configured length is reached. Additionally, if an LP A terminates a chain and happens to be the first chain initiated in that round by another LP B, then A sends an event to B to schedule the next round, effectively closing the chain. The logical dependence chains in this model are long and can stress-test tie-breaking implementations.

We examined two E-TIES configurations with distinct features, as presented in Table 1. The *easy* configuration was designed such that in each round, each LP only initiates a single chain of events with a maximum length of two. On the other hand, the *hard* configuration involves each LP producing three chains of length five, which results in far more demanding tie-breaking activities.

The tuberculosis (TBC) model [14] is a real-world agent-based Susceptible-Infected-Recovered (SIR) model that focuses on the spread of epidemics in large populations. Agents move around a geographical area consisting of several LPs, with each LP maintaining a record for each agent. The individuals in the model can be in the *healthy*, *infected*, *sick* (active TBC), *under treatment*, or *cured* state. The model also considers individual parameters like age, origin, risk factors (such as smoking), and immunosuppression, with the presence or absence of lung cavitation being considered after infection. We simulate two million agents, for a total logical time of one thousand simulated days.

For each configuration, we show four different plots. In order from left to right, we have:

1) the event processing time in seconds, which does not include initialisation and finalization costs;
2) the speedup computed over the corresponding serial configuration;
3) the efficiency, i.e., the percentage of committed events over the total number of executed events;
4) the cost per event in nanoseconds, divided between the actual event processing and event extraction costs. We measure those using the `rdtsc` instruction available on most *x86* CPUs.

We note that, in ROOT-Sim, both the serial and parallel engines eagerly insert scheduled events in the Future Event Set; therefore, event insertion operations are already included in the event processing costs.

5.2. Experimental Results

The results of the PHold model in Figure 5 show that there is no significant difference between the two configurations, as expected[4]. The tie-breaking logic is not stressed extensively, as timestamp deltas are drawn from an exponential distribution, and randomisation of timestamps across 64-bit floating point samples is sufficient to avoid event ties. Moreover, PHold may not be the best choice to evaluate the characteristic of our tie-breaking policy because, in its classic formulation, its events are all indistinguishable. The serial runtime shows unusually high event extraction costs, which is unsurprising given that the ROOT-Sim serial engine is optimised for smaller-scale models.

In contrast, the E-TIES model presents a more diverse trend, even in its *easy* configuration (Figure 6), highlighting the cost of using a tie-breaking strategy. Ignoring ties provides a significant advantage, especially at low core counts, yet both configurations exhibit good performance. However, the overall speedup indicates a favourable trend for the tie-breaking configuration. Possibly, the configuration without tie-breaks cannot scale further because the parallelism of the model is already being vastly exploited.

We can draw two notable observations. First, the serial execution of E-TIES *easy* exhibits lower event processing costs than PHold, despite its more complex logic. One reason for this is that when E-TIES is executed without tie-breaking logic, inserting events into the heap typically requires very few operations because the events are placed at the end of the heap. Likewise, in tie-breaking executions, the chain position identifier is used as the event type. Most newly-scheduled chain events do not happen before those in the queue, resulting in event insertions that require only a limited number of operations. Conversely, in PHold, timestamps are sampled from an exponential distribution: they are better distributed across an interval of logical time. As a result, logarithmic heap costs are observed since heap bubbling operations are necessary during insertions.

The other interesting fact is the vast difference in event extraction costs, which could explain the observed performance gap between the two parallel configurations. To further prove this, efficiency is mostly conserved between the two configurations, with only a slight increase in the number of rollbacks, possibly due to the stricter causality requirement imposed by our event ordering.

The results of the more demanding version of E-TIES, shown in Figure 7, reinforce these findings, showing that ignoring ties results in even better performance. As more events are tied, extraction costs increase significantly, strengthening the argument that queue management is the main factor that affects performance. Nevertheless, the speedup shows that both configurations scale well with a larger model. At the same time, the efficiency confirms that the lower performance of tie-breaking runs is not due to an increase in rollbacks.

Finally, we observe that the experimental results of the TBC model in Figure 8 suggest that tie-breaking may have a limited impact on performance for real-world models. While a noticeable difference can be observed at lower thread counts, the added cost of tie-breaking is effectively parallelised away with higher core count configurations. The evaluation of PHold and TBC shows that our proposal has a negligible cost with few or no event ties but a high cost when ties are prevalent. We have shown that this is not due to parallelisation inefficiencies, but rather because the FES of the simulation engine is effectively doing more work in the attempt to provide the correct event extraction ordering.

This increased cost is inevitable when using a priority queue that relies on element comparisons. Such a data structure requires at least one operation involving a logarithmic number of comparisons with respect to the events in the queue. Even if the modeller writes an efficient event comparison function, the event inspection cost would still be much higher than the timestamp comparison, often requiring only a few machine instructions. Furthermore, existing priority-queue data structures that do not involve direct timestamp comparisons are unsuitable for this purpose. For example, in a calendar queue, all tied events would end up in the same bucket, resulting in disastrous linear extraction times.

In other words, it seems that running a parallel simulation full of tied events with the requirement of deterministic executions may result in much more stress on queue management operations. As a solution, it may be possible to alleviate this burden by assigning the responsibility of re-ordering messages with the same timestamp to the straggler detection system. Although interesting, our initial investigation of this approach suggested that the associated cost of more frequent rollbacks trumps any other performance gain.

Moving to the second part of our evaluation, we consider how the lack of tie-breaking affects a model's dynamics. For this analysis, we used the output produced by the TBC model. To extract the relevant data from the simulation, each LP in TBC schedules an event at regular intervals, which saves the count of the five classes of agents to a buffer that belongs to the LP's state. Upon simulation completion, the agent counts from each LP are merged by timestamp to produce a comprehensive evolution timeline of agent states.

The outcomes depicted in Figure 9 demonstrate no notable variation in the results among the different types of executions[5]. This may lead one to believe that tie-breaking is insignificant in real-world scenarios. However, two factors should be considered. First, this model was chosen for its resistance to ties, enabling non-tie-broken executions to run without errors. Secondly, each model run produced slightly different trajectories, even with an identical pseudo-random generator seed. In contrast, runs with tie-breaks are entirely deterministic for the same pseudo-random seed. As mentioned, deterministic simulation runs are valuable, particularly during model debugging.

Second, as previously noted, models that do not heavily use tie-breaking logic do not experience a significant decrease in performance. Thus, it may be beneficial to maintain the enhanced tie-breaking feature enabled anyway.

4. The minimum at 14 worker threads is related to the double ring of the Intel Broadwell-EP CPU used in the experimentation

5. The number of healthy individuals is negligible, given the pandemic-like scenario, making them mostly invisible in the plots.

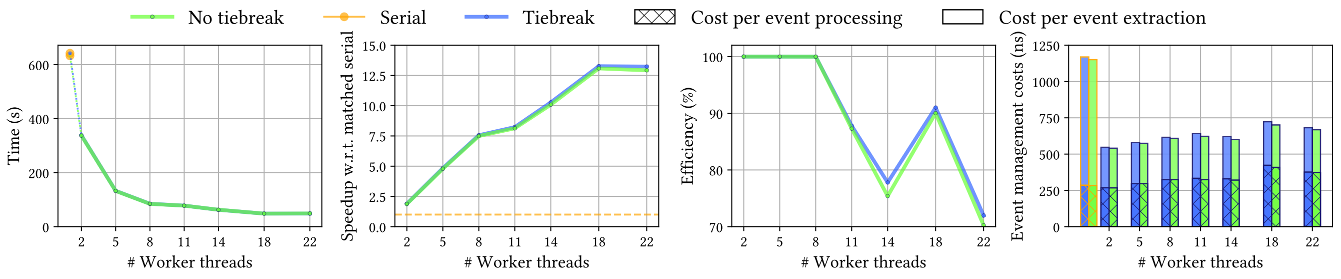


Figure 5: Results for *PHold* model

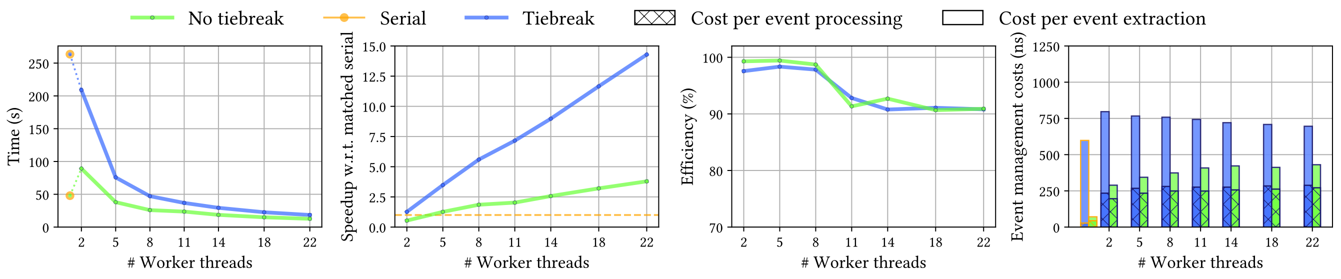


Figure 6: Results for *event-ties* model, easy configuration

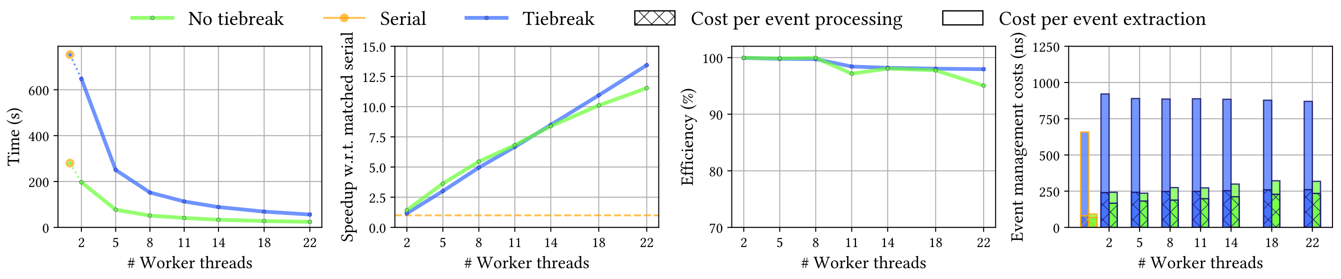


Figure 7: Results for *event-ties* model, hard configuration

6. Conclusions

We have explored a simple yet effective tie-breaking technique that enables replicable and repeatable executions of concurrent simulations, in a way similar to what would be observed in a sequential execution using the same technique. Our approach is easy to implement for both model developers and simulation engine designers. Although our strategy may not be the optimal solution from a modelling point of view, it still serves as a fundamental tool to implement a suitable model-specific tie-breaking strategy. Our experimental results showed that the cost of using our technique is negligible for models with infrequent event ties. We provide insights into the performance implications for models with many ties. Most importantly, we demonstrated that, in any case, our technique does not hinder parallel-scaling properties.

In conclusion, we have discovered that in certain situations, the tie-breaking processing may not be required, even for practical applications. Nevertheless, we believe that for these models, the expense of implementing tie-breaking is minimal, and as a result, we suggest keeping it turned on.

References

[1] S. Abar, G. K. Theodoropoulos *et al.*, “Agent based modelling and simulation tools: A review of the state-of-art software,” *Computer Science Review*, vol. 24, pp. 13–33, 2017.

[2] F. J. Barros, “On the representation of time in modeling & simulation,” in *2016 Winter Simulation Conference*, T. M. K. Roeder, P. I. Frazier *et al.*, Eds. Piscataway, NJ, USA: IEEE, Dec. 2016, pp. 1571–1582.

[3] A. C. Chow, “Parallel DEVS: A parallel, hierarchical, modular modeling formalism and its distributed simulator,” *Transactions of the Society for Computer Simulation*, vol. 13, no. 2, pp. 55–68, Jul. 1996.

[4] R. M. Fujimoto, “Performance of time warp under synthetic workloads,” in *Distributed Simulation*, ser. PADS ’90, D. Nicol, Ed. San Diego, CA, USA: Society for Computer Simulation International, 1990, pp. 23–28.

[5] ——, “Zero lookahead and repeatability in the high level architecture,” Georgia Institute of Technology, Tech. Rep., 1999.

[6] D. Jefferson and H. Sowizral, “Fast concurrent simulation using the time warp mechanism. part I. local control,” The Rand Corporation, Santa Monica, CA, USA, Tech. Rep. N-1906-AF, Dec. 1982.

[7] D. R. Jefferson, “Virtual time,” *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 3, pp. 404–425, Jul. 1985.

[8] D. R. Jefferson and P. D. Barnes, “Virtual time III, part 1: Unified virtual time synchronization for parallel discrete event simulation,”

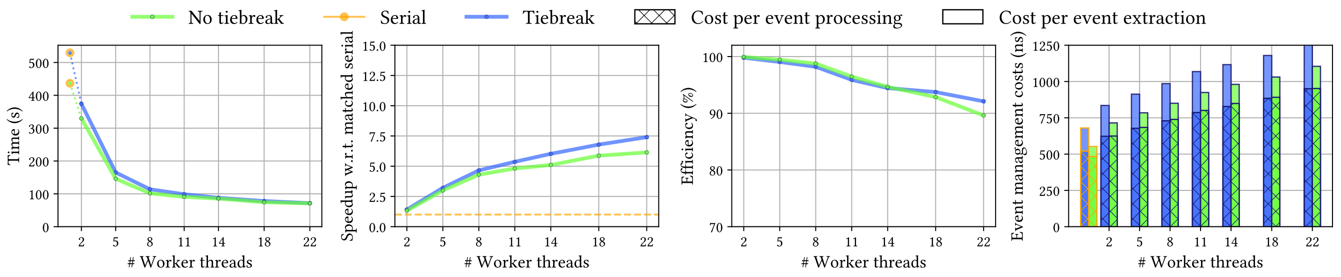


Figure 8: Results for *TBC* model

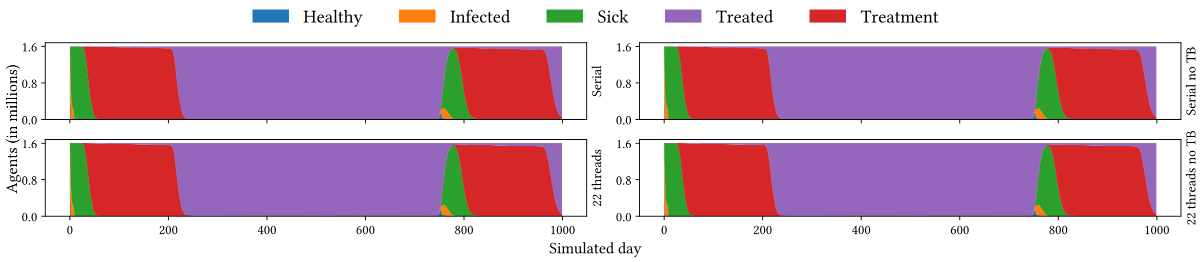


Figure 9: Evolution of the TBC model

ACM Transactions on Modeling and Computer Simulation, vol. 32, no. 4, pp. 1–29, Sep. 2022.

[9] D. R. Jefferson, B. Beckman *et al.*, "Time warp operating system," in *Proceedings of the eleventh ACM Symposium on Operating systems principles*, ser. SOSP '87. New York, NY, USA: Association for Computing Machinery, Nov. 1987, pp. 77–93.

[10] K. H. Kim, Y. R. Seong *et al.*, "Ordering of simultaneous events in distributed DEVS simulation," *Simulation practice and theory*, vol. 5, no. 3, pp. 253–268, Mar. 1997.

[11] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[12] N. McGlohon and C. D. Carothers, "Toward unbiased deterministic total orderings of parallel simulations with simultaneous events," in *Proceedings of the 2021 Winter Simulation Conference*, ser. WSC '21, S. Kim, B. Feng *et al.*, Eds. Piscataway, NJ, USA: IEEE, Dec. 2021, pp. 1–15.

[13] H. Mehl, "A deterministic tie-breaking scheme for sequential and distributed simulation," in *Proceedings of the Multiconference on Advances in Paralleland Distributed Simulation*, ser. PADS '91, V. K. Madisetti, D. Nicol, and R. M. Fujimoto, Eds. San Diego, CA, USA: Society for Computer Simulation, 1992, pp. 199–200.

[14] C. Montañola-Sales, J. F. Gilabert-Navarro *et al.*, "Modeling tuberculosis in barcelona. a solution to speed-up agent-based simulations," in *Proceedings of the 2015 Winter Simulation Conference*, ser. WSC, L. Yilmaz, W. K. V. Chan *et al.*, Eds. Piscataway, NJ, USA: IEEE, Dec. 2015, pp. 1295–1306.

[15] N. Nethercote and J. Seward, "Valgrind: A program supervision framework," *Electronic notes in theoretical computer science*, vol. 89, pp. 44–66, 2003.

[16] D. M. Nicol and X. Liu, "The dark side of risk (what your mother never told you about time warp)," in *Proceedings of the 11th Workshop on Parallel and distributed simulation*, ser. PADS '97. Washington, DC, USA: IEEE Computer Society, Jun. 1997, pp. 188–195.

[17] A. Pellegrini, R. Vitali, and F. Quaglia, "The ROme OpTimistic simulator: Core internals and programming model," in *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, ser. SIMUTOOLS. Brussels, Belgium: ICST, Apr. 2012, pp. 96–98.

[18] L. F. Perrone, "On the evolution toward computer-aided simulation," in *Modeling and Simulation-Based Systems Engineering Handbook*, D. Gianni, A. D'Ambrogio, and A. Tolk, Eds. Boca Raton, FL, USA: CRC Press, 2014, pp. 95–118.

[19] P. Peschlow and P. Martini, "Efficient analysis of simultaneous events in distributed simulation," in *Proceedings of the 11th International Symposium on Distributed Simulation and Real-Time Applications*, ser. DS-RT'07. Piscataway, NJ, USA: IEEE, Oct. 2007, pp. 244–251.

[20] B. Preiss, "The YADDES distributed discrete event simulation specification language and execution environment," in *Proceedings of the 1989 Multiconference on Distributed Simulation*, ser. PADS '89, B. Unger and R. M. Fujimoto, Eds. San Diego, CA, USA: Sociesty for Computer Simulation International, 1989, pp. 139–144.

[21] P. L. Reiher, F. Wieland, and P. Hontalas, "Providing determinism in the time warp operating system-costs, benefits, and implications," in *IEEE Workshop on Experimental Distributed Systems*. IEEE Comput. Soc, 1990, pp. 113–118.

[22] A. Ruscheinski and A. Uhrmacher, "Provenance in modeling and simulation studies — bridging gaps," in *Proceedings of the 2017 Winter Simulation Conference*, ser. WSC '17, Wai Kin (Victor), A. D'Ambrogio *et al.*, Eds. Piscataway, NJ, USA: IEEE, Dec. 2017, pp. 872–883.

[23] R. Rönngren and M. Liljenstam, "On event ordering in parallel discrete event simulation," in *Proceedings 13th Workshop on Parallel and Distributed Simulation*, ser. PADS '99. Piscataway, NJ, USA: IEEE Comput. Soc, 2003, pp. 1–8.

[24] M. Schordan, T. Oppelstrup *et al.*, "Reversible languages and incremental state saving in optimistic parallel discrete event simulation," in *Reversible Computation: Extending Horizons of Computing*, ser. Lecture notes in computer science. Cham: Springer International Publishing, 2020, pp. 187–207.

[25] F. Wieland, "The threshold of event simultaneity," in *Proceedings of the 11th workshop on Parallel and Distributed Simulation*, ser. PADS '97. USA: IEEE Computer Society, Jun. 1997, pp. 56–59.