

Interval Branching

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

Jason Liu
*School of Computing and Information Sciences
Florida International University
Miami, Florida 33199
liux@cis.fiu.edu*

Abstract

We propose a new simulation method, called interval branching, which aims to facilitate efficient simulation studies of systems that involve temporal uncertainty. The method uses interval timestamps for events and explores different execution orders of overlapping events by branching the simulation. We first present a sequential version of interval branching, and then extend it to parallel simulation using the logical process paradigm. The parallel interval branching algorithm uses the logical-process cloning technique for efficient computation of branches. Our preliminary experiments show its potential as an efficient method for discrete-event simulation studies.

1 Introduction

Discrete-event simulation is a widely used technique for analyzing the behavior of complex systems. Commonly, discrete-event simulation studies involve uncertainty about input parameters, and aim to evaluate its effect on simulation output. This is done by running multiple replications of the simulation until the desired statistical accuracy is reached. This may be a time-consuming process, which motivates the search for alternative simulation methods.

In this paper, we propose interval branching, a new discrete-event simulation method that aims to facilitate efficient simulation studies in the presence of *temporal* input uncertainty, i.e., uncertainty about the time of event occurrences. Instead of sampling precise event timestamps from time intervals and running several replications, interval branching includes time intervals into the simulation itself: events carry interval timestamps which represent the time span in which they may possibly occur.

When two or more event timestamps overlap at some point in the simulation, multiple execution orders of the events are possible. Interval branching creates simulation

branches for all possible execution orders, and continues with the simulation of all branches. In the process, it also calculates probabilities for the branches based on the event timestamps. Thus, interval branching is able to compute the complete set of simulation results that may emerge from the given temporal uncertainty in a single simulation run.

The concepts of interval timestamps and branching do not depend on a particular discrete-event simulation technique. However, interval branching uses parallel simulation techniques [4, 14] based on the logical process (LP) paradigm for efficiency. With parallel simulation, the system state is partitioned into LPs, each of which handles events related to its state partition. While parallel LP execution on different processors enables potential speedup, interval branching benefits from LPs in two additional respects. First, new branches are only created if overlapping intervals occur at the same LP. Only then the events can affect the same state variables. Second, LPs allow us to use cloning techniques for efficient computation of branches.

In the following, we establish the fundamental concepts of interval branching, and discuss its branching algorithms in detail. Our experiments with a prototype implementation of interval branching show its potential as an efficient simulation method for studies involving temporal uncertainty.

The paper is organized as follows: in section 2 we give an overview of related work. Section 3 establishes the foundations of interval branching and describes the branching and probability calculation algorithms. In section 4 we extend interval branching to a parallel simulation method. We present preliminary performance results in section 5, followed by a discussion in section 6. Section 7 summarizes the paper and gives an outlook on possible future work.

2 Related work

2.1 Interval timestamps

Interval timestamps have been recognized as a representation of temporal uncertainty in discrete-event simulation.

In [3], Fujimoto introduces *approximate time* as a means of increasing concurrency in parallel simulations. Approximate time uses interval timestamps for events, and enables synchronization mechanisms to execute events with overlapping timestamps in any order. Approximate time synchronization has been shown to achieve noticeable speedup compared to simulations with precise timestamps [3].

Qualitative discrete event simulation [10], developed by Ingalls et al., is a sequential method for simulation studies where temporal uncertainty cannot be quantified. Like our method, it combines interval timestamps with a branching mechanism and calculates a set of results. Interval branching, however, takes advantage of LPs and parallel simulation techniques. Furthermore, we aim to support simulation studies where probability distributions are known.

Time intervals are also used in *fuzzy discrete-event simulation* [18], which represents uncertainty about event occurrences by fuzzy sets. Whenever two fuzzy sets overlap, the event execution order is selected by a *ranking scheme*, in order to find the most probable path of execution. Note that a combination of fuzzy discrete-event simulation and our interval branching method would be able to examine the influence of fuzziness on simulation model behavior.

Other representations of temporal uncertainty have also been used in simulations. For a survey of these approaches, we refer to the related work section of [12].

2.2 Simulation branching and cloning

The traditional way of exploring alternative model behavior in discrete-event simulation is to run independent replications using different input parameters [6]. Branching approaches instead examine alternatives at *decision points* in the simulation, and run different simulation *branches* from the decision points onwards. Different strategies for simulation branching are discussed in [5].

In the context of parallel simulation with logical processes, Hybinette and Fujimoto have introduced *cloning* as an efficient computation technique for branches [8]. Cloning replicates only those parts of the system state that differ among branches. At a decision point, the simulation creates clones of LPs that show alternative behavior. All other LPs become *shared LPs* which are associated with multiple branches and perform common computations for them. An event message sent by a shared LP contains all IDs of its associated branches may thus have multiple receivers. In the course of the simulation, shared LPs inspect the branch IDs of event messages they receive. Whenever a shared LP finds that it receives different event messages for different branches, it has to clone itself. This way the differences among branches spread among LPs.

It has been shown that cloning can speed up parallel simulations dramatically when compared to independent repli-

cations. Therefore, we use cloning for the computation of branches in interval branching as well. For a comprehensive summary of cloning, see [9]. Recent extensions to the original cloning are described in [7, 1, 2].

2.3 Simultaneous events

In discrete-event simulation, two or more events with identical timestamps are called *simultaneous events*. Commonly, simulation tools use tie-breaking rules to order simultaneous events [16, 11]. However, since tie-breaking rules may bias simulation results [17], another approach is to examine different execution orders of simultaneous events. Recently, a branching mechanism for simultaneous events in parallel and distributed simulation has been presented [15]. It uses LP cloning for efficient computation of branches.

While interval branching focuses on exploring the effects of temporal uncertainty specified by interval timestamps, this includes precise timestamps as a special case. Thus, our method can also be used to analyze simultaneous events.

3 Interval branching

In this section, we present the sequential interval branching algorithm. In section 4, we introduce the parallel algorithm. We first introduce interval timestamps and relations among events, after which we describe the overall structure of the algorithm and discuss the procedure for branching and the calculation of branching probabilities.

We define the timestamp of an event e as a closed bounded set of real numbers

$$e = [E(e), L(e)]$$

with $E(e) \leq L(e)$. We call $E(e)$ the earliest time (or *E-time*) and $L(e)$ the latest time (or *L-time*) of event e . A similar definition can be found in [12]. Events with $E(e) < L(e)$ are called *interval events*, and events with $E(e) = L(e)$ are called *precise events*. Furthermore, $\text{size}(e) = L(e) - E(e)$.

We define a total order “ \preceq ” among events. Consider two events $e_1 = [E(e_1), L(e_1)]$ and $e_2 = [E(e_2), L(e_2)]$. Then

$$e_1 \preceq e_2 \iff E(e_1) < E(e_2) \vee (E(e_1) = E(e_2) \wedge L(e_1) \leq L(e_2))$$

That is, events are sorted by their E-times, and only then by their L-times. Interval branching assumes all FELs to use “ \preceq ” as their sorting criterion for events.

We define a binary relation “ \parallel ” among events:

$$e_1 \parallel e_2 \iff \neg(L(e_1) < E(e_2) \vee L(e_2) < E(e_1))$$

That is, $e_1 \parallel e_2$ is true if their timestamp intervals overlap (with at least one common point). If $e_1 \parallel e_2$, we call e_1 and e_2 *overlapping events*. Note that \parallel is reflexive and symmetric, but not transitive.

Algorithm 1 : sequential interval branching

```
initialize simulation:  $V \leftarrow V_0, E \leftarrow E_0, t \leftarrow t_0, p \leftarrow 1.0$   
 $B \leftarrow \{\langle V, E, t, p \rangle\}$   
WHILE  $|B| > 0$  DO  
   $b = \langle V, E, t, p \rangle \leftarrow \text{getAndRemoveNextBranch}(B)$   
  WHILE  $b$ 's termination condition is not fulfilled DO  
     $E_p \leftarrow \text{getNextEvents}(E, \emptyset)$  // see Algorithm 2  
    IF  $|E_p| = 1$  THEN  
      execute the event in  $E_p$  (update  $t$ ,  $V$ , and  $E$ )  
    ELSE  
       $B' \leftarrow \text{branching}(b, E_p)$  // see Algorithm 3  
       $B \leftarrow B \cup B'$   
      break from the inner loop
```

For the sake of brevity, we make two extensions to the definition of \parallel . Let e be an event and t a point in simulation time. Then $e \parallel t$ if e overlaps with the induced event $[t, t]$. Let E be a set of events. $e \parallel E$ is true, if $E = \emptyset$. Otherwise, let e' be the event with the smallest L-time of all events in E . We define $e \parallel E$ if $e \parallel e'$.

We define another binary relation “ \subseteq ” among events:

$$e_1 \subseteq e_2 \iff (E(e_1) \geq E(e_2) \wedge L(e_1) \leq L(e_2)).$$

We say e_1 is a subinterval of e_2 .

3.1 Basic structure of the algorithm

The simulation consists of a set B of branches. Each branch is represented by a tuple $\langle V, E, t, p \rangle$, where V is the set of state variables, E is a list of future events sorted according to the total order “ \preceq ” described above, t is the simulation clock, and p is the probability of the branch. The simulation starts with a single branch. Additional branches may be added due to the difference in execution order of pending events. The probabilities of the branches are determined by the likelihood of their respective event orderings. The simulation finishes when all branches are completed.

The sequential interval branching (shown in Algorithm 1) is similar to the classical discrete-event simulation algorithm, which features a single future event list for each branch. We start the simulation by initializing state variables V_0 , adding initial events E_0 , and setting the simulation clock t_0 . The initial branch is given a probability of 1. The outer loop is used to process all unfinished branches in B —one at a time, selected by the `getAndRemoveNextBranch` method following a certain branch selection criterion (e.g., with the largest branch probability or smallest simulation clock). The inner loop iteratively executes events in E according to the total order “ \preceq ”. If the set of pending events E_p as returned by the `getNextEvents` method (shown in Algorithm 2) contains only a single event, the event is simply executed. How-

Algorithm 2 : getNextEvents

```
INPUT:  $E$ : the set of future events, and  $E_p$ : the set of pending events  
WHILE  $|E| > 0$  and  $E.\text{first} \parallel E_p$  DO  
   $e \leftarrow \text{removeFirstEvent}(E)$   
   $E_p \leftarrow E_p \cup \{e\}$   
RETURN  $E_p$ 
```

ever, if more than one pending event is discovered, a branching procedure must be initiated. The branching procedure examines all possible execution orders of the pending events in E_p and calculates their resulting states together with their probabilities. If branching results in additional branches, they are returned by the `branching` method and subsequently added to the set of branches B for later processing.

3.2 The branching procedure

Different execution orders of pending events form an *execution tree* [15]. Each node in the tree is represented as a tuple $\langle b, E_p \rangle$, where $b = \langle V, E, t, p \rangle$ is the branch, and E_p is the set of pending events that remain to be sorted. The root of the tree is the simulation state at the beginning of branching. All other nodes correspond to states reached in the course of the branching procedure. A leaf node is defined as a node of the tree with $E_p = \emptyset$. Every path from the root to a leaf node corresponds to a different event execution order.

The branching procedure (shown in Algorithm 3) starts with the current branch b and a set of pending events E_p . All nodes of the tree are then traversed one at a time starting from the root $\langle b, E_p \rangle$. R is the set of branches that represent the outcome of the possible event orderings. These branches are stored at the leaf nodes of the execution tree. The algorithm starts with an empty R and gradually populates the set as leaf nodes are discovered. R is returned eventually once all tree nodes are explored.

N is the set of nodes that are yet to be explored. Each iteration of the loop takes an unexplored tree node from N , returned by the `getAndRemoveNextNode` method. The algorithm then derives a subset of events of E_p that can be executed next with nonzero probabilities and calculates their probabilities accordingly. In the next section, we describe in detail the method `calcProbabilities` which returns the executable events along with their probabilities.

For every event e with the calculated probability p_e , we continue the branch represented by the tree node $\langle b^0, E_p^0 \rangle$ currently being explored; we update the branch probability p , remove the event e from the set of pending events E_p^0 , and adjust the timestamps of the events, before executing the event. Event timestamps need to be adjusted (with the `adjustTimeStamps` method) because a particular ordering of the events may lead to the shrinking of times-

Algorithm 3 : branching

INPUT: $b = \langle V, E, t, \rho \rangle$: the current branch, and E_p : the set of pending events

$R \leftarrow \emptyset$; $N \leftarrow \{ \langle b, E_p \rangle \}$

WHILE $|N| > 0$ **DO**

$\langle b^0, E_p^0 \rangle \leftarrow \text{getAndRemoveNextNode}(N)$

$P_b \leftarrow \text{calcProbabilities}(E_p^0)$

FOR ALL $\langle e, p_e \rangle \in P_b$ **DO**

$b \leftarrow b^0$; $p \leftarrow p \cdot p_e$; $E_p \leftarrow E_p^0 \setminus \{e\}$

adjustTimeStamps(e, E_p)

execute event e (update t, V , and E)

$E_p \leftarrow \text{getNextEvents}(E, E_p)$

IF $|E_p| > 0$ **THEN**

$N \leftarrow N \cup \{ \langle b, E_p \rangle \}$

ELSE

$R \leftarrow R \cup \{b\}$

RETURN R

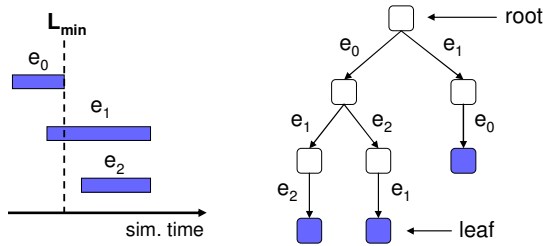


Figure 1. Execution tree for three events.

tamp intervals. For example, in Figure 1, if e_1 happens before e_0 , $L(e_1)$ should be changed to L_{\min} and $E(e_0)$ should be changed to $E(e_1)$. In general,

$$L(e) \leftarrow \min_{e_o \in E_p \cup \{e\}} \{L(e_o)\}, \text{ and}$$

$$E(e_o) \leftarrow \max\{E(e_o), E(e)\}, \forall e_o \in E_p.$$

Removing event e from the set of pending events may cause more events (including those generated by e) from the event list E to be included. This is because the set of overlapping events is determined by the smallest L -time of all events (see the definition of “||”). We therefore update E_p by invoking the `getNextEvents` method again. A new tree node $\langle b, E_p \rangle$ is added to N , unless it is a leaf node, in which case the resulting branch is added to R .

Figure 1 shows an example set of pending events and the resulting execution tree. The algorithm starts with an initial set of pending events $E_p = \{e_0, e_1\}$. If e_0 is executed first, e_2 overlaps with the remaining event e_1 and is removed from E and included into E_p . This leads to the execution orders (e_0, e_1, e_2) and (e_0, e_2, e_1) . If, however, e_1 is executed first, only e_0 remains while e_2 stays in E of the corresponding branch. This results in a total of three leaf states.

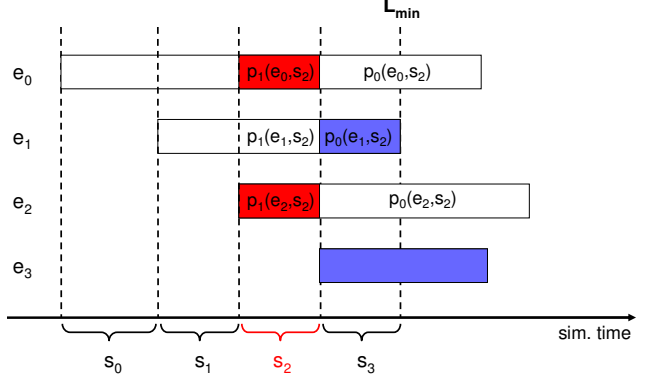


Figure 2. Event occurrences in subintervals.

3.3 Calculation of probabilities

When events overlap, the probability that an event is executed next depends on two factors: the way the events overlap, and the probability distributions associated with their timestamps. In the following, we assume uniform distribution for interval events. The probability calculation algorithm (the `calcProbabilities` method) receives the set of pending events E_p and first of all computes the subset of events E_o that can be executed next with nonzero probabilities: $E_o \leftarrow \text{getNextEvents}(E_p, \emptyset)$. In the following discussion, we assume all events in E_o to be interval events. We include precise events as special cases afterwards.

To calculate the probabilities of selecting the events to execute next, we first sort the events in E_o according to “ \preceq ”:

$$e_0 \preceq e_1 \preceq \dots \preceq e_{n-1}, \text{ where } n = |E_o|.$$

Next, we construct the intervals s_0, s_1, \dots, s_{n-1} , such that $E(s_i) = E(e_i)$, and $L(s_i) = E(e_{i+1})$, if $i < n - 1$; $E(s_{n-1}) = E(e_{n-1})$, and $L(s_{n-1}) = L_{\min}$, where L_{\min} is the smallest L -time of all events in E_o . Figure 2 shows an example that consists of four overlapping events (e_0, e_1, e_2 , and e_3) and the corresponding intervals (s_0, s_1, s_2 , and s_3).

It is easy to verify that $s_j \subseteq e_i$, if $0 \leq i \leq j < n$. Let $p_1(e_i, s_j)$ be the probability that event e_i is executed in the interval s_j . Let $p_0(e_i, s_j)$ be the probability that e_i is executed after $L(s_j)$. Assuming uniform distribution, $p_1(e_i, s_j) = \text{size}(s_j) / \text{size}(e_i)$; similarly, $p_0(e_i, s_j) = (L(e_i) - L(s_j)) / \text{size}(e_i)$. If $i > j$, $p_1(e_i, s_j) = 0$ and $p_0(e_i, s_j) = 1$.

We need to examine the conditional probability of executing event e_i before other events, given that e_i is executed in the interval s_j , which is denoted by $p(e_i, s_j)$. Such a probability can be calculated by considering all combinations of whether other events are executed in s_j or not. Using the example shown in Figure 2, let us examine $p(e_0, s_2)$. Suppose we consider the combination that e_2 is executed in s_2 and e_1 is executed after s_2 (we know e_3 must be executed after s_2 , since $p_0(e_3, s_2) = 1$). The probability of this configu-

ration is $(p_0(e_1, s_2) \cdot p_1(e_2, s_2))$. Since both e_0 and e_2 are executed in s_2 , the probability of the combination is shared equally between the two events. The probability considering all combinations is:

$$p(e_0, s_2) = p_0(e_1, s_2) \cdot p_0(e_2, s_2) + p_0(e_1, s_2) \cdot p_1(e_2, s_2)/2 + p_1(e_1, s_2) \cdot p_0(e_2, s_2)/2 + p_1(e_1, s_2) \cdot p_1(e_2, s_2)/3.$$

In general, we have

$$p(e_i, s_j) = \sum_{0 \leq k < 2^j} \frac{1}{1 + \sum_{0 \leq b < j} \beta(k, b)} \prod_{0 \leq b < j} p_{\beta(k, b)}(e_{\omega(b, i)}, s_j),$$

if $0 \leq i \leq j < n$, where $\beta(k, b)$ is the b^{th} least significant bit of number k in binary, and

$$\omega(b, i) = \begin{cases} b & \text{if } b < i \\ b+1 & \text{otherwise} \end{cases}$$

If $i > j$, $p(e_i, s_j) = 0$. We can now calculate the probability of event e_i being executed first:

$$p_{e_i} = \sum_{i \leq j < n} p_1(e_i, s_j) \cdot p(e_i, s_j).$$

The `calcProbabilities` method thus returns the set $\{(e, p_e) | e \in E_o\}$.

Now consider precise events that may be included in E_o . Let $E_x = \{e \in E_o | E(e) = L(e)\}$ be the subset of precise events in E_o . It must be true that these precise events have the same timestamp as L_{\min} . We only need to remove the precise events from E_o ($E_o \leftarrow E_o \setminus E_x$) to allow the algorithm above to work properly. To calculate the probability p_x that these precise events are executed first, we only need to consider the probability that all events in E_o are executed after L_{\min} , i.e., $p_x = \prod_{e \in E_o} p_0(e, s_{n-1})$.

If more than one precise event is present, this is the case commonly referred to as simultaneous events (cf. section 2.3), where tie-breaking rules may be available. In the absence of tie-breaking information, however, we assume all execution orders of simultaneous events are equally probable; we divide p_x by $|E_x|$.

The `calcProbabilities` method calculates exact probabilities for a given set of pending events. It is important to notice, however, that the repeated application of the method, as done by Algorithm 3, only provides *estimates* of the branch probabilities. Recall that the `calcProbabilities` method assumes uniform distribution for all intervals every time it is called. This, however, does not always hold, even if the simulation starts out with uniform distributions. Consider the two overlapping events e_0 and e_1 shown in Figure 3(a). The probability that e_0 is executed first is obviously $p_{e_0} = 0.5$. Now assume the simulation continues the branch (e_0, e_1) , i.e., e_0 is executed first.

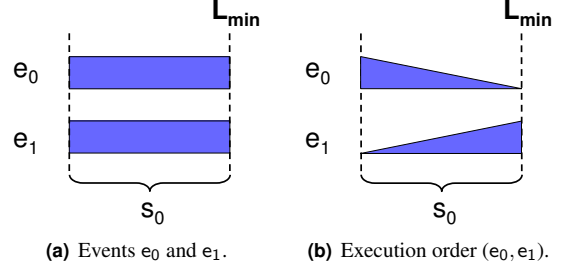


Figure 3. Changing probability distribution.

Figure 3(b) shows the resulting probability distributions, given that e_0 is executed first. These are not uniform distributions anymore, yet the `calcProbabilities` method will assume uniform distribution as soon as e_0 (or another event generated by e_0) is subject to branching again.

The example shows that, if the simulation requires the calculation of exact probabilities, interval branching has to support non-uniform distributions (which is desirable anyway, since non-uniform distributions are commonly used in simulations). In order to support non-uniform probability distributions, two requirements have to be fulfilled. First, distributions with unlimited support have to be approximated by double-bounded distributions (i.e., a finite interval). Second, the cumulative distribution function should have a closed form so that probabilities may be calculated efficiently. We will investigate this in future work.

4 The parallel branching algorithm

In this section, we propose a parallel interval branching algorithm based on logical processes. The use of parallel simulation techniques has two main advantages in addition to the potential gain in speedup. First, events are considered as overlapping only when they occur at the same LP. Thus, the branching procedure is invoked only if there is a reasonable chance that different execution orders lead to different states. Second, parallel simulation enables the use of LP cloning techniques, which may significantly reduce the time necessary to explore all branches.

4.1 Branching and cloning

The simulation consists of a set of LPs, managed by a synchronization mechanism. Initially, all LPs belong to the same branch with branch ID 0. When an LP_A encounters overlapping events, it invokes the branching procedure and calculates the leaf states of the execution tree. Each leaf state represents an alternative simulation behavior. Following the principles of cloning (cf. section 2.2), LP_A then creates a clone for each leaf state and assigns globally unique

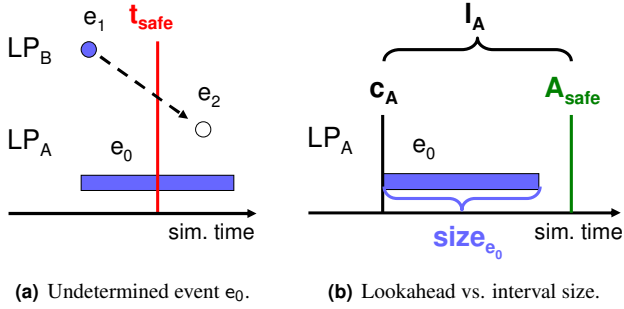


Figure 4. Undetermined events.

IDs to the branches. Since the branching at LP_A has not influenced the state of other LPs, they are shared clones until the state diverges. As simulation continues, all branches are computed in parallel, and all LPs (whether shared or not) participate in the same synchronization mechanism.

As an example, assume that the simulation initially consists of 100 LPs, and branching at LP_A results in two leaf states. Then, LP_A initializes two clones with branch ID 0 and branch ID 1, each corresponding to a leaf state. All other LPs are set as shared clones for both branches. As a result, the simulation virtually consists of 200 LPs, but only 101 of them (99 shared clones plus LP_A⁰ and LP_A¹) perform computations. Now assume that, later on, branching at a shared LP_B^{0,1} results in two leaf states. Then, it initializes two clones which are also shared LPs: LP_B^{0,1} and LP_B^{2,3}.

It is possible that branching takes place at different LPs at the same time. In order to ensure globally unique branch IDs and set shared clones correctly, we use a central *Branch Manager* (BM). An LP_A that completes the branching procedure reports the resulting leaf states to the BM. The BM in turn propagates the new branch IDs to all LPs at a global synchronization point. LP_A then completes the creation of clones, and other LPs are set as shared clones accordingly.

4.2 Synchronization

We assume a conservative synchronization mechanism, which calculates safe times with respect to the E-times of events: an LP with safe time t_{safe} is guaranteed not to receive any event e with $E(e) < t_{\text{safe}}$. Further, we assume that all LPs have a nonzero lookahead.

Let us first make a few definitions. Let LP_A be an LP with safe time t_{safe} . Every event e scheduled at LP_A belongs to one of three categories. If $L(e) < t_{\text{safe}}$, or $E(e) < t_{\text{safe}}$ and $L(e) = t_{\text{safe}}$, e is *safe*. That is, the timestamp of a safe precise event must be strictly less than t_{safe} ; the L-time of a safe interval event must be less than or equal to t_{safe} . If $E(e) \geq t_{\text{safe}}$, e is *unsafe*. If, however, $E(e) < t_{\text{safe}}$ and $L(e) > t_{\text{safe}}$, e is an *undetermined* event. Note that all undetermined events are interval events.

The parallel algorithm requires modifications to the sequential interval branching method to cope with undetermined events. Although there are several possible ways of handling undetermined events, all have their drawbacks. Consider an undetermined event e_0 at LP_A, as shown in Figure 4(a). If LP_A simply executes e_0 , it may miss the possible execution order (e_2, e_0) in case LP_B schedules event e_2 at LP_A. This clearly has to be avoided. Therefore, LP_A may consider to *aggressively* handle e_0 by creating two simulation branches: on one branch, e_0 is executed and its timestamp is modified to $[E(e_0), t_{\text{safe}}]$, while on the other branch e_0 is not executed and receives the timestamp $[t_{\text{safe}}, L(e_0)]$. This, however, may result in an unnecessary branch if no LP schedules an overlapping event at LP_A. To avoid these drawbacks, LP_A may choose a *cautious* approach, waiting for a higher guarantee t_{safe} that makes e_0 safe. Blocking, however, leads to potential deadlock situations.

Our implementation of interval branching uses a window-based synchronization protocol. In each window, all LPs receive a global safe time t_{safe} , execute all their safe events, and finally use a barrier synchronization to calculate the global next safe time from all guarantees of the LPs. We define the current guarantee A_{safe} of an LP_A as $A_{\text{safe}} = c_A + l_A$, where c_A is the smallest E-time of all events currently scheduled at LP_A, and l_A the *lookahead* of LP_A, a lower bound on the timestamps of events that LP_A may possibly schedule at other LPs. The global safe time as calculated at each barrier is $t_{\text{safe}} = \min(L_{\text{safe}} : L \text{ is LP})$.

With the described protocol, we can ensure progress of the cautious approach: in the presence of undetermined events, LP_A may continue waiting for higher guarantees as long as $t_{\text{safe}} < A_{\text{safe}}$. If, however, $t_{\text{safe}} = A_{\text{safe}}$ and event e is still undetermined, LP_A has to proceed, e.g., by creating two branches after all. Note that it is impossible for LP_A to receive a higher guarantee than A_{safe} .

We now show that with the cautious approach and the described protocol, an undetermined event e_0 eventually becomes safe if the condition $\text{size}(e_0) < l_A$ is fulfilled. Let $\text{size}(e_0) < l_A$. Then

$$L(e_0) \stackrel{\text{def.}}{=} E(e_0) + \text{size}(e_0) = c_A + \text{size}(e_0)$$

$$\stackrel{\text{def.}}{=} A_{\text{safe}} - l_A + \text{size}(e_0) < A_{\text{safe}}.$$

Note that $c_A = E(e_0)$ because e_0 is the first event at LP_A. Now let LP_A wait. Then, all other LPs eventually have executed their events up to A_{safe} . It follows $t_{\text{safe}} = A_{\text{safe}}$, and thus $L(e_0) < t_{\text{safe}}$. \square

As a conclusion, an LP taking the cautious approach is *never* required to create branches for an undetermined event if its interval size is smaller than the LP's lookahead. The complete procedure of the LP execution session within a synchronization window is given in Algorithm 4.

Algorithm 4 : LP execution

INPUT: $b = \langle V, E, t, p \rangle$: the current LP state, l : the current lookahead, and t_{safe} : LP safe time

WHILE $E.\text{first} < t_{\text{safe}}$ **DO**

$E_p \leftarrow \text{getNextEvents}(E, \emptyset, t_{\text{safe}})$ // see Algorithm 5

IF $|E_p| = 1$ **THEN**

$e \leftarrow$ the event in E_p

IF e is undetermined **THEN**

IF $\text{size}(e) < l$ and LP is cautious **THEN**

reinsert e into E

RETURN

ELSE

$B \leftarrow \text{branching}(b, E_p, t_{\text{safe}})$ // see Algorithm 6

informBranchManager(B)

ELSE

execute e (update t , V , and E)

ELSE

$B \leftarrow \text{branching}(b, E_p, t_{\text{safe}})$ // see Algorithm 6

informBranchManager(B)

Algorithm 5 : getNextEvents (parallel version)

INPUT: E : the set of future events, E_p : the set of pending events, and t_{safe} : LP safe time

WHILE $|E| > 0$ and $E.\text{first} < t_{\text{safe}}$ and $E.\text{first} \parallel E_p$ **DO**

$e \leftarrow \text{removeFirstEvent}(E)$

$E_p \leftarrow E_p \cup \{e\}$

RETURN E_p

4.3 Branching undetermined events

The branching procedure for overlapping events has to support the processing of undetermined events if there is possibility they may occur. We start with the sequential branching algorithm given in section 3.1. We first need to redefine the set of pending events E_p to filter out unsafe events. This can be achieved simply by selecting only those events with E-time smaller than t_{safe} . Algorithm 5 shows the modified `getNextEvents` method, which is used by the parallel branching procedure described next.

In the process of branching, we create an additional “pseudo” leaf node in the execution tree every time the set of pending events consists entirely of undetermined events. Figure 5(a) shows an example of an execution tree with three overlapping events: e_0 , e_1 , and e_2 . Pseudo leaves are created on branches where all safe events have been executed. For example, upon executing e_0 , the branching procedure is left with two undetermined events: e_1 and e_2 . However, if the execution takes e_1 and then e_0 , we would reach a pseudo leaf node with one undetermined event e_2 . Similarly, if we follow the path e_2 and then e_0 , we would reach a pseudo leaf node with e_1 as the sole remaining undetermined event. The parallel version of the branching pro-

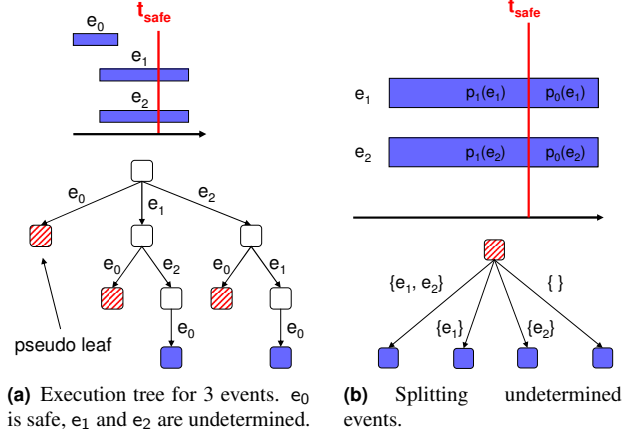


Figure 5. Branching undetermined events.

cedure is shown in Algorithm 6. Similar to the sequential algorithm, it iteratively processes the nodes of the execution tree as it unfolds. Upon reaching a pseudo leaf, which is determined by L_{min} (the smallest L-time of all events in E_p) becoming larger than t_{safe} , the branching procedure invokes the subroutine `processUndetermined` to create branches for the remaining undetermined events.

In the `processUndetermined` method, we consider all possible branches resulting from different combinations of whether the undetermined events are executed within the same synchronization window or they should wait until the next window. That is, we need to split the undetermined events and calculate the probabilities of different event orders in different branches. Consider the example shown in Figure 5(b), which consists of two undetermined events: e_1 and e_2 . We first calculate the probability of e_1 being executed within the current synchronization window, denoted as $p_1(e_1)$. Such probability can be calculated easily as $(t_{\text{safe}} - E(e_1))/\text{size}(e_1)$, owing to the uniform distribution assumption. The probability that e_1 is executed beyond the current window is $p_0(e_1) = 1 - p_1(e_1)$. We can calculate the probabilities $p_0(e_2)$ and $p_1(e_2)$ for e_2 in the same fashion.

Then, we consider four cases resulting four different branches: 1) both e_1 and e_2 happen before t_{safe} ; 2) e_1 happens before t_{safe} and e_2 happens after t_{safe} ; 3) e_1 happens after t_{safe} and e_2 happens before t_{safe} ; and 4) both e_1 and e_2 happen after t_{safe} . Each branch is then assigned with the correct probability accordingly. For example, the probability of the second branch is $(p_1(e_1) \cdot p_0(e_2) \cdot p)$, where p is the probability of the branch reaching the pseudo leaf node (when the `processUndetermined` method is invoked).

We then follow each possible branch and readjust the timestamps of the pending events: if an event is to happen before t_{safe} , we change its L-time to be t_{safe} ; otherwise, we change its E-time to t_{safe} . In the latter case, we remove the event from the pending event set E_p and restore it to the

Algorithm 6 : branching (parallel version)

INPUT: $b = \langle V, E, t, p \rangle$: the current branch, E_p : the set of pending events, and t_{safe} : LP safe time
 $R \leftarrow \emptyset$; $N \leftarrow \{\langle b, E_p \rangle\}$
WHILE $|N| > 0$ **DO**
 $\langle b^0, E_p^0 \rangle \leftarrow \text{getAndRemoveNextNode}(N)$
 $L_{\min} = \min_{e \in E_p^0} L(e)$
 IF $L_{\min} > t_{\text{safe}}$ **THEN**
 processUndetermined($b^0, E_p^0, t_{\text{safe}}, N, R$)
 continue with the while-loop
 $P_b \leftarrow \text{calcProbabilities}(E_p^0)$
 FOR ALL $\langle e, p_e \rangle \in P_b$ **DO**
 $b \leftarrow b^0$; $p \leftarrow p \cdot p_e$; $E_p \leftarrow E_p^0 \setminus \{e\}$
 adjustTimeStamps(e, E_p)
 execute event e (update t, V , and E)
 $E_p \leftarrow \text{getNextEvents}(E, E_p, t_{\text{safe}})$
 IF $|E_p| > 0$ **THEN**
 $N \leftarrow N \cup \{\langle b, E_p \rangle\}$
 ELSE
 $R \leftarrow R \cup \{b\}$
RETURN R

FEL, so that it can be processed in the next synchronization window. This step removes some of the undetermined events and converts the rest to safe events. Algorithm 7 shows the algorithm in detail. As defined in section 3.1, $\beta(k, i)$ is the i^{th} least significant bit of binary number k .

The procedure finishes when all possible combinations are considered. In the branch that all events are selected to be executed beyond t_{safe} , the branching procedure reaches a tree leaf and therefore the resulting branch shall be recorded in R . For others, a new tree node $\langle b, E_p \rangle$ is created and stored in N . These tree nodes will continue to be processed by the branching procedure when the `processUndetermined` method returns.

Just like in case of a single undetermined event, LPs may follow a cautious strategy (cf. section 4.2) instead of branching undetermined events. This would involve an interruption of the branching procedure when, for all unexplored tree nodes, E_p only consists of undetermined events.

5 Experiments and results

In this section, we present performance results of a prototype implementation of interval branching from a series of experiments with a simple air traffic simulation model. Every LP simulates a single airport. Each airplane either waits on the ground at an airport or flies between airports. We use a fixed ground time of 120 minutes and introduce uncertainty about flight times. For every pair of airports i and j , we specify a minimum flight time $d_{i,j}^{\min}$ and a maximum

Algorithm 7 : processUndetermined

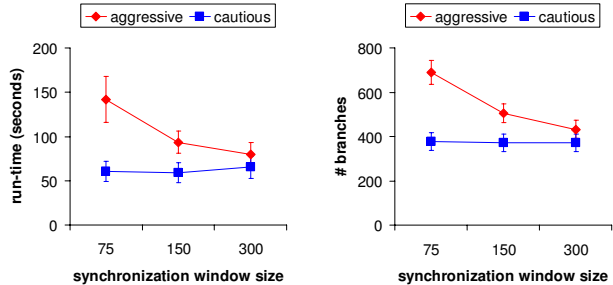
INPUT: $b^0 = \langle V, E, t, p \rangle$: the current branch, $E_p^0 = \{e_0, e_1, \dots, e_{n-1}\}$: the set of pending (undetermined) events (where $n = |E_p^0|$), t_{safe} : LP safe time, N : the set of unexplored tree nodes, and R : the set of resulting branches
FOR ALL $i = 0$ to $n - 1$ **DO**
 $p_1(e_i) = (t_{\text{safe}} - E(e_i)) / \text{size}(e_i)$;
 $p_0(e_i) = 1 - p_1(e_i)$
FOR $k = 0$ to $2^n - 1$ **DO**
 $q \leftarrow \prod_{i=0}^{n-1} p_{\beta(k,i)}(e_i)$ // probability of combination k
 IF $q > 0$ **THEN**
 $E_p \leftarrow \emptyset$; $b \leftarrow b^0$; $p \leftarrow p \cdot q$
 FOR $i = 0$ to $n - 1$ **DO**
 IF $\beta(k, i) = 1$ **THEN**
 $L(e_i) \leftarrow t_{\text{safe}}$; $E_p \leftarrow E_p \cup \{e_i\}$
 ELSE
 $E(e_i) \leftarrow t_{\text{safe}}$; $E \leftarrow E \cup \{e_i\}$
 IF $|E_p| > 0$ **THEN**
 $N \leftarrow N \cup \{\langle b, E_p \rangle\}$
 ELSE
 $R \leftarrow R \cup \{b\}$

flight time $d_{i,j}^{\max}$. That is, each “flight arrival” event has an interval timestamp of the form $[t + d_{i,j}^{\min}, t + d_{i,j}^{\max}]$, where t is the flight’s departure time. All minimum flight times are constants chosen randomly between 300 and 500 minutes. Thus, the simulation has a lookahead of 300 minutes.

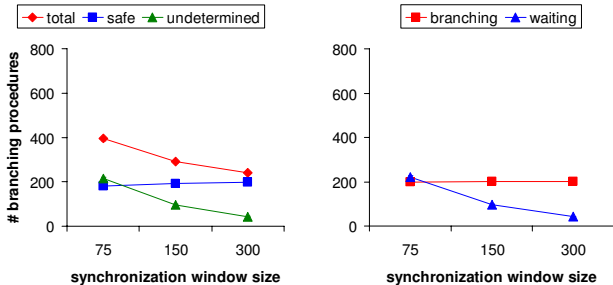
All simulations were run on a machine with two 2.8 GHz Intel Xeon Dual-Core processors, so four processor cores were available for parallel simulation. All figures contain averages of 30 runs and 95% confidence intervals. All simulations are run for 100,000 minutes (i.e., 10 weeks).

Our first evaluation scenario, *scenario 1*, consists of 100 airports and a total of 300 airplanes. Due to bad weather conditions at airport A, flight times to airport A are uncertain: we set $d_{i,A}^{\max} = d_{i,A}^{\min} + 30$ for all airports i . Note that in this case the interval size (30) is smaller than the lookahead. For the rest of the flight times, we use precise timestamps, i.e., $d_{i,j}^{\max} = d_{i,j}^{\min}$ for all airports $j \neq A$.

We first evaluate the performance of processing undetermined events in the branching procedure between the cautious strategy (LPs block and wait for higher guarantees) and the aggressive strategy (LPs create branches immediately). We vary the synchronization window size to be 75, 150, and 300 minutes, respectively. Figure 6(a) shows the execution times. The cautious strategy consistently outperforms the aggressive strategy. This can be explained by the number of created branches (shown in Figure 6(b)). Smaller lookahead increases the frequency of undetermined events; the aggressive strategy immediately creates extra branches, while the cautious strategy blocks the LP.



(a) Simulation run-time. (b) Total number of branches.

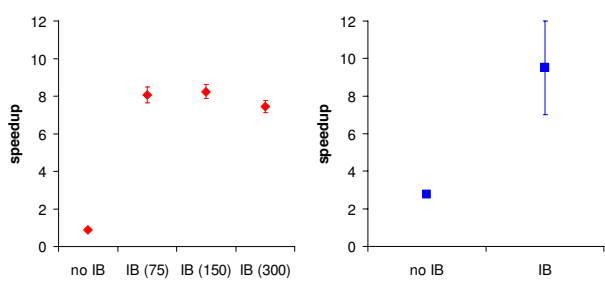


(c) Number of branching procedures (aggressive). (d) Number of branching procedures and waiting (cautious).

Figure 6. Scenario 1: influence of lookahead.

Figure 6(c) breaks down the number of times the branching procedure is invoked in the aggressive strategy. While almost all invocations are due to safe overlapping events when the synchronization window size is 300 minutes, more than 50% are initiated because of undetermined events in the case of 75 minutes. For the cautious strategy, Figure 6(d) shows that the number of times the LPs waited because of undetermined events (“waiting”) is almost identical to the number of branching procedures due to undetermined events in Figure 6(c). Furthermore, the total number of branches (“branching”) is equal to the number of safe branches created with the aggressive strategy. This means that the cautious strategy can successfully avoid all unnecessary branches due to undetermined events. We conclude that the cautious strategy is clearly preferable.

Next we examine the performance of interval branching compared to the traditional approach running independent replications. We define the *normalized run-time* t_{IB} as the execution time of interval branching divided by the number of explored branches. We then define *speedup* as the execution time of a sequential simulation divided by t_{IB} . Note that the speedup defined here only suggests performance improvement over an equivalent number of independent simulation runs. Recall that interval branching includes probability calculation for all branches. To provide a good *estimate* of the probability of a branch using the traditional discrete-event simulation method, it would require



(a) Speedup in scenario 1. (b) Speedup in scenario 2.

Figure 7. Speedup of interval branching.

many independent trials.

Figure 7(a) shows the speedup obtained with interval branching in scenario 1. For reference, we also include the speedup (0.9) achieved by a parallel simulation without branching (“no IB”). Interval branching runs 8 times faster than sequential simulation with independent replications. Even if we consider that, with four processor cores available, we may run four sequential simulations simultaneously, interval branching still completes twice as fast.

To increase parallelism, we examined a larger scenario, *scenario 2*, with 400 airports and 12,000 airplanes. The only difference from the previous scenario is that we randomly choose an airport during the simulation and use interval timestamps for flights departing from the airport. Figure 7(b) shows the speedup with the cautious strategy. Again, we include the speedup (2.8) of a parallel simulation without branching for reference. Interval branching leads to a much higher speedup, almost 10 compared to sequential replications. Note that the larger confidence interval is due to the larger variation in the number of branches created in the different simulation runs. The fluctuation mirrors reality in the sense that we usually do not know in advance how much temporal uncertainty affects simulation behavior.

The results presented in this section are only a first step towards a performance evaluation; yet they already provide us valuable insight to the potential benefit of interval branching. It is important to point out that interval branching should not be applied blindly without consideration of the simulation scenario at hand. For example, we also ran a set of simulations with uncertain flight times between *all* airports. This led to an enormous number of branches which could no longer be handled. We discuss possible ways to tackle the branch explosion problem in the next section.

6 Discussion

In this section we discuss two potential limitations of interval branching that we have not addressed so far: branch explosion, and interdependency of events.

As indicated at the end of section 5, interval branching may require prohibitive computation time or exhaust memory space when the number of branches is large. Therefore, it is important to keep the number of branches as small as possible. First of all, the unnecessary creation of branches should be avoided. Interval branching uses state comparisons to detect duplicates during the branching procedure. This is costly, however, if states consist of many variables. One way to reduce these costs is to utilize *a priori knowledge* about event interactions, if available. This enables LPs to decide in advance whether different execution orders lead to the same state. If so, they refrain from branching [15].

Even if branches differ at first, they may eventually reach identical states again. Therefore, a merging mechanism for such branches is desirable [1]. A variation of this approach is to have user-defined equivalence classes in order to recombine *almost* identical branches. Some simulations may also benefit from a trade of information for efficiency by discarding branches which are not “interesting”. One example is to drop a branch if its probability is below a certain threshold [13]. For other simulation studies, however, it may be advisable to focus on these “rare branches”.

Another possible limitation of interval branching is that the branching procedure assumes that overlapping events are independent. However, this is not always the case. As an example, consider *broadcast events*, which have more than one receiving LP. Assume that two copies of a broadcast event, e_A and e_B , are received by LP_A and LP_B respectively. Further assume that branching procedures including e_A and e_B are invoked at both LPs. Then, on some branches, the timestamps of e_A and e_B might be modified such that $L(e_A) < E(e_B)$. Since there is no global knowledge of the interdependency of e_A and e_B , inconsistent branches would be created. A similar problem has been discussed in [3].

7 Conclusions and future work

In this paper, we have introduced interval branching, a new parallel discrete-event simulation method which represents temporal uncertainty with interval timestamps and branches the simulation in case of overlapping events. We have described the simulation methodology and the algorithms used by interval branching. Our preliminary performance evaluation of interval branching shows its potential for efficient simulations much faster than the traditional method.

In future work, we will pursue several open issues that we have pointed out in the paper, e.g., non-uniform probability distributions, and interdependency of events. Furthermore, we will perform case studies in different application scenarios. In particular, we are going to investigate applications that allow for the specification of pruning criteria in order to cope with the branch explosion problem.

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] 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.
- [3] R. M. Fujimoto. Exploiting temporal uncertainty in parallel and distributed simulations. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, 1999.
- [4] R. M. Fujimoto. *Parallel and Distributed Simulation Systems*. Wiley Interscience, 2000.
- [5] J. B. J. Gilmer and F. J. Sullivan. Alternative implementations of multitrajectory simulation. In *Proceedings of the 30th Conference on Winter Simulation*, 1998.
- [6] S. G. Henderson. Input model uncertainty: why do we care and what should we do about it? In *Proceedings of the 35th Conference on Winter Simulation*, 2003.
- [7] M. Hybinette. Just-in-time cloning. In *Proceedings of the 18th Workshop on Parallel and Distributed Simulation*, 2004.
- [8] M. Hybinette and R. Fujimoto. Cloning: A novel method for interactive parallel simulation. In *Proceedings of the 29th Conference on Winter Simulation*, 1997.
- [9] M. Hybinette and R. M. Fujimoto. Cloning parallel simulations. *ACM Transactions on Modeling and Computer Simulation*, 11(4):378–407, 2001.
- [10] R. G. Ingalls, D. J. Morrice, and A. B. Whinston. The implementation of temporal intervals in qualitative simulation graphs. *ACM Transactions on Modeling and Computer Simulation*, 10(3):215–240, 2000.
- [11] V. Jha and R. Bagrodia. Simultaneous events and lookahead in simulation protocols. *ACM Transactions on Modeling and Computer Simulation*, 10(3):241–267, 2000.
- [12] M. L. Loper and R. M. Fujimoto. A case study in exploiting temporal uncertainty in parallel simulations. In *Proceedings of the 2004 International Conference on Parallel Processing*, 2004.
- [13] T. Ono-Tesfaye and P. Gburzynski. A discrete event simulation approach to protocol validation. In *Proceedings of the 13th European Simulation Multiconference*, 1999.
- [14] K. S. Perumalla. Parallel and distributed simulation: Traditional techniques and recent advances. In *Proceedings of the 37th Conference on Winter Simulation*, 2006.
- [15] P. Peschlow and P. Martini. Efficient analysis of simultaneous events in distributed simulation. In *Proceedings of the 11th IEEE International Symposium on Distributed Simulation and Real Time Applications*, 2007.
- [16] 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.
- [17] F. Wieland. The threshold of event simultaneity. *Transactions of the Society for Computer Simulation International*, 16(1):23–31, 1999.
- [18] H. Zhang, H. Li, and C. M. Tam. Fuzzy discrete-event simulation for modeling uncertain activity duration. *Engineering, Construction and Architectural Management*, 11(6):426–437, 2004.