Merging Parallel Simulations

by

Abhishek Agarwal

(Under the Direction of Maria Hybinette)

Abstract

In earlier work cloning is proposed as a means for efficiently splitting a running simulation midway through its execution into multiple parallel simulations. In simulation cloning, clones usually are able to share computations that occur early in the simulation, but as their states diverge individual logical processes (LP’s) are replicated as necessary so that their computations proceed independently. Over time the state of the clones (or their constituent LPs) may converge. Traditionally, these converged LPs would continue to execute identical events. We address this inefficiency by merging of previously cloned LPs. We show that such merging can further increase efficiency beyond that obtained through cloning only. We discuss our implementation of merging, and illustrate its effectiveness in several example simulation scenarios.

Index Words: Parallel and Distributed Event Simulation (PDES), Cloning, Merging Simulations.
MERGING PARALLEL SIMULATIONS

by

ABHISHEK AGARWAL

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment
of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2004
MERGING PARALLEL SIMULATIONS

by

ABHISHEK AGARWAL

Major Advisor:  Maria Hybinette
Committee:
    Thiab Taha
    Eileen T. Kraemer

Electronic Version Approved:
Maureen Grasso
Dean of the Graduate School
The University of Georgia
December 2004
DEDICATION

To my family
ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Maria Hybinette for all her help and encouragement. Without her guidance this project would not have been possible. I would also like to thank Dr. Thiab Taha and Dr. Eileen T. Kraemer for their valuable comments and suggestions. Their feedback has helped make this work much better than its initial draft.
# TABLE OF CONTENTS

Page

ACKNOWLEDGEMENTS.............................................................................................................v

LIST OF FIGURES ...................................................................................................................... vii

CHAPTER

1  BACKGROUND .........................................................................................................................1

2  INTRODUCTION ....................................................................................................................3

3  PROBLEM STATEMENT .........................................................................................................5

4  IMPLEMENTATION .................................................................................................................8

5  PERFORMANCE .....................................................................................................................12

6  RELATED WORK ....................................................................................................................17

7  CONCLUSION ........................................................................................................................19

REFERENCES ..............................................................................................................................21
LIST OF FIGURES

Figure 1: Sequence of Events in Simulation Cloning and Simulation Cloning with Merging……12
Figure 2: Performance of P-Hold...................................................................................................14
Figure 3: Speedup of Cloning with Merging over Pure Cloning...................................................15
Figure 4: Scalability of Merging....................................................................................................16
CHAPTER 1
BACKGROUND

Simulation is the means of modeling the behavior of some entity. A computer simulation models the behavior of an entity with the help of a computer program. Simulations can serve as effective decision making tools and to create virtual environments. Examples of areas in which simulations aid as decision making tools are air and ground transportation, modeling network characteristics and manufacturing. Virtual environments are used for training purposes such as military training and pilot training.

In discrete simulation, the simulation state changes only at discrete intervals of time. When the time-interval is constant in a discrete simulation, the simulation is known as time stepped. As opposed to changing state at constant time increments, discrete event simulations change state only on the occurrence of an event. Here a time stamp is associated with each event. When a simulation receives an event, it advances its time to the time-stamp of the current event and marks the message as processed. Processing a message may or may not lead to generation of new messages.

When more than one computer is used to execute a discrete event simulation, the system is referred to as a Parallel Discrete Event simulation (PDES). PDES can be implemented on either one computer having multiple processors or from different computers distributed across the globe. Parallelism has the effect of making the simulations run faster as well as making it more fault-tolerant. In the Parallel Discrete Event Simulation paradigm, a sequential simulation is referred to as a Logical Process (LP). Each LP models an entity (physical process) in the physical system. A physical system may consist of many entities. Thus many LP’s are used to model a physical system with each LP representing a physical process.
Though parallel simulations offer the benefit of lower execution time and fault tolerance, it also adds new complexities in order to keep the state consistent with that of a sequential simulation. In the case of a sequential computer it is straightforward to keep the state consistent because it is easy to process events in a time stamped order. This is not the case when the simulation is a PDES. The process of ensuring that events are processed in time stamped order is called the synchronization problem. A mechanism is required to solve this synchronization problem. There are two well-known protocols that solve this problem. One is known as the conservative synchronization protocol and the other is the optimistic synchronization protocol.

Conservative synchronization protocols strictly avoid processing events out of time-stamp order. It waits until it can make sure than the event it is processing is the one with the smallest time-stamp. This is the general principle of the conservative synchronization protocols. However, conservative synchronization protocols are prone to deadlock. The null message algorithm (Chandy and Misra 1978, Bryant 1977) essentially solves this problem.

Optimistic synchronization protocols allow an event to be processed out of timestamp order, but provide an additional mechanism for rollback (Jefferson and Sowizral, 1982) of that event. Rollback of an event has the effect of undoing the processing of that event. The events are then again processed in timestamp order. The net effect is as if the events have been processed in timestamp order.

Our research emphasizes on the performance enhancement of PDES. Though we have used an optimistic simulator for our work, our approach is equally applicable to both optimistic and conservative simulators.
CHAPTER 2
INTRODUCTION

In this work we introduce simulation merging as the converse of cloning and we show that for efficient parallel simulations merging is as important as cloning. To put merging in context, we must first briefly review simulation cloning.

In earlier works simulation cloning (Hybinette and Fujimoto 2001) was introduced as a means of gaining efficiency in the execution of parallel simulations. Simulations are cloned at decision points. Cloning dynamically replicates simulations at decision points. This allows computations before the decision point to be shared. As simulation time progresses, the decision points are analogous to forks in the time-line where the state of the executing simulations begins to diverge. Cloning is efficient because all simulations after a fork benefit from shared computations made before the fork. In fact, because our cloning scheme is incremental, clones continue to benefit from shared computations after a decision point, depending on how quickly the change in state propagates among the constituent LPs.

We refer to a simulation clone as fully diverged when all of its LPs have been cloned. Even though computations were saved before diverging, after diverging the continuing simulation is as expensive as a separate standalone simulation. However, as simulations continue to run, it is possible, perhaps likely, that the state of separate clones that were once distinct may converge. This is possible, for instance, in simulations of stable systems that tend to converge to an equilibrium, or of cyclic systems that “reset” each day (e.g. the air traffic network).

It is natural to wonder, if once clones have converged to similar states, might they be recombined or merged? In other words, can forked time lines be recombined? In this way a simulation could benefit again from shared computations in the same way that it did before the fork.
Consider that a single simulation clone may comprise hundreds or thousands of LPs distributed over dozens of processors. At first it would seem that merging simulations is a bit like putting the proverbial genie back in the bottle. How could we efficiently evaluate the state of such a vast process, and then compress it back together with its sibling? In fact, the answer is simple, flexible and dynamic. Our approach is to recombine simulations gradually, in the same way that they are cloned in the first place: one LP at a time.

Specifically, LPs occasionally compare their state with the state of their peers. If the states are identical (or “close enough”), the LPs recombine. This enables a clone to be fully diverged or fully re-converged with another, or anywhere between these two extremes.
CHAPTER 3
PROBLEM STATEMENT

In order to provide context for the problem, we must first review our approach to simulation cloning in more detail. After that, we describe the need for and challenge of simulation merging.

For cloning and merging we assume that simulations consist of multiple logical processes (LPs) that communicate exclusively by time stamped messages. Conceptually, one may consider that the simulation is composed of virtual logical processes, physical logical processes, and messages. The relationship between physical and virtual LPs is similar to that between physical and logical (virtual) address space in modern virtual memory systems. The cloning software maps the simulation programmer’s virtual logical processes and messages to their corresponding “real” physical logical processes and messages. This approach was introduced in (Hybinette and Fujimoto 2001). Virtual LPs and messages do not have a physical realization. Instead, each virtual LP maps to one physical LP that has a physical realization in the parallel simulation system. Similarly, each virtual message maps to a single physical message.

Computation common between two or more clones are completed by a physical LP and shared by the virtual LPs in the clones that are mapped to the physical LP. Similarly, virtual messages among different clones are mapped to a single physical message to avoid duplication. Sharing of common computations may proceed until the state of one of the virtual LPs diverges from the others. At this point the LP must be replicated and it becomes a physical LP. If replication is “one way” only, all cloned simulations will eventually end up in a fully replicated state. Once in this state, efficiency from shared computations is no longer feasible. However, it may be the case that cloned simulations converge back to
similar or identical states. It may be possible to reclaim efficiency by merging (inverse of cloning) simulations back together.

Our problem is twofold:

1. How can we identify clones that may be merged safely? and
2. How can we accomplish the merging?

Our approach is to consider merging at the LP level—in much the same way that cloning is accomplished at the LP level. Accordingly a clone, composed of many LPs, is merged with another one LP at a time.

As an example, consider an air traffic simulation. Suppose that cloning was being used to evaluate whether to restrict traffic at the Sacramento airport (which is represented as an LP in our simulation). Later, the restriction is lifted. After the restriction is lifted, flights may gradually return to a normal pattern, and be similar to the original simulation without a restriction. In this case, during the period of the restriction, the states of simulated airports (LPs) near the Sacramento airport may diverge. However, after the restriction is lifted they may gradually converge to the state of a simulation without a restriction.

A number of sub-problems arise as we consider how merging should be implemented. For example, there are situations in which unexpected thrashing may occur. Suppose we have a fully replicated air traffic simulation and the cloned simulations are now beginning to converge. Specifically, suppose that the multiple LPs representing the Sacramento airport can now be merged and the merge is performed. After the merge, the Sacramento airport can accept arrivals of aircraft from airports that are still diverged, which would lead to an immediate re-cloning of the LP. However if that aircraft arrival did not cause the airport to diverge, or differ, it becomes an immediate candidate for merging again. We can quickly wind up in a merge/clone cycle, wasting enormous amounts of CPU effort.

Another consideration is when or how frequently we should check for re-convergence. One solution is to consider merging a cloned logical process at any time that it receives a message, this has the benefit that after a merge, LPs will avoid being cloned again as in our previous example. However, this
may result in too much overhead. A more practical approach is to check en masse at specific checkpoint times. The frequency of checks could be specified by the simulation application designer.

To reduce the cost of full LP state comparisons we envision the merging system to use heuristics or statistical methods that gauge the most likely elements to be different, and to compare those elements first. In our initial implementation however we assume that LPs have a limited amount of state to compare. This is a reasonable assumption for optimistic simulations; otherwise system memory would be quickly exhausted by the simulation’s state saving features.

The several problems we must address then are:

• How and when can we detect that logical processes re-converge?

• How can we avoid clone-merge thrashing?

• How frequently should we test for merging?

• How can we implement merging so that the resulting simulation outcome is correct?

We address these questions in the sections that follow. Our earlier work showed that cloning can provide a significant application performance advantage for a wide class of simulation executions. In this work our initial results indicate that clone-merging can provide similar performance improvement.
CHAPTER 4
IMPLEMENTATION

Merging is implemented together with cloning in a layered fashion with the operating system at the bottom, the simulation executive in the middle, and the simulation application at the top. The code that implements merging and cloning is layered between the simulation application and the simulation executive. This approach allows cloning and merging to run on top of various underlying simulation executives. From the point of view of the executive, cloning and merging software is simply part of the simulation application. Cloning is independent of the synchronization mechanism (optimistic or conservative); however, our implementation is using an optimistic simulation kernel.

An efficient implementation of merging requires us to pay careful attention to two issues: comparisons and thrashing. Merging, of course, requires state comparisons, but state comparisons are expensive, so our implementation must consider how to minimize them. Additionally, we must avoid merge/clone thrashing cycles, in which an LP is merged and then immediately re-cloned due to receiving a message from an LP that has not yet been merged. We address these issues at the application level and within the cloning/merging library.

We observe that it is difficult to estimate how long it might take for cloned LPs to converge without knowledge of the application. Accordingly, in order to reduce the number of comparisons, we rely on the application programmer to provide a recommendation for how frequently LPs should be checked to see if they have converged. This information is provided by the application programmer using an API given later in this section.

Thrashing occurs when a cloned LP is merged then quickly re-cloned. This happens when the cloned LP receives a message from another LP that is still cloned, even though the sender’s LP state may
not have diverged from it’s corresponding cloned physical logical process. Thrashing may be further aggravated when merging runs on an optimistic simulation engine. Our solution to the first part of the problem is to delay replication (cloning) until it can be determined that the sender’s virtual logical processes are indeed diverged. Observe that identical sender virtual logical processes send identical messages. Thus one can rely on message checking to detect situations where replication is necessary. This issue was addressed Just-In-time cloning (Hybinette, M. 2004). Just-In-time cloning delays replication of cloned logical processes until it is absolutely necessary, thus avoiding unnecessary replication. Merging uses Justin-time cloning to avoid unnecessary re-cloning.

When merging is implemented on an optimistic executive we may find situations in which LPs are merged, but then the merging event is rolled back. Clearly, this behavior could contribute to merge/clone/merge thrashing. We address the problem by scheduling merging events conservatively at physical logical processes at global virtual time.

Now we introduce the API for merging that is provided to the simulation application programmer. The API also includes calls to support cloning, which are described in CITATION. Three merging functions are available. We list the function names below, and then describe them in detail later. The functions are prefixed with the letters CM to indicate that they are functions from the clone merge library:

```c
CM_Enable (check-frequency);
CM_Disable ();
CM_RegisterFunction (mergefunc);
```

**CM_Enable()** and **CM_Disable()** enable and disable merging. **CloneMerge_Enable()** turns on a timer (in simulated time) in the simulation executive, and each time the timer expires, a clone merge check is performed. When the simulation is completely merged **CloneMerge_Disable** is called automatically. The check-frequency parameter defines how regularly the merging library checks for merging after the simulation is cloned.
CM RegisterFunction(mergefunc) registers a user-defined function. This function compares two LPs so that it can be determined if they are ready for merging. The function ‘mergefunc’ returns either True or False depending on whether the LPs should be merged or not. We require this function from the user because it is possible that the state of two LPs may be bit-wise different, but still equivalent. The kernel cannot determine this without help from the application level. Also, to allow the user to define the function the constraint on how similar the state must be may be relaxed and can consequently increase the efficiency of merging.

The merging library uses the merge_time [LP][clone] data structure to store the time to make a comparison to test for merging for each physical logical process. The matrix is indexed by logical process number and clonenumber. This is set by the merge library. It is initialized by the time given by the parameter check-frequency in CM_Enable (check-frequency) and is reset every time the timer expires. In general the comparison time of clones of a particular logical process should be set to the same time in order to synchronize the merging event. Observe that different logical processes may schedule merging asynchronously, especially if they do not communicate, without any impact on performance.

The merging library also uses two functions internally (transparent to the simulation application):

CM_ScheduleMergeEvent ()
CM_Merge (parent_lp, child_lp);

CM_ScheduleMergeEvent () is scheduled by a logical process on itself by the merging library when the timer for state checking expires. This is done by sending a blocking message to itself. In general, parent and descendant virtual logical processes should schedule comparisons simultaneously to that their states are synchronized. In order to prevent indefinite blocking, priority is given to the parent process.

If two LPs are equivalent they are merged by CM_Merge (parent_lp, child_lp). This function changes the mapping of the virtual logical processes so that their corresponding physical logical process
are now equivalent. The child's old physical logical process is set to inactive and returned to the pool of physical logical processes.
CHAPTER 5
PERFORMANCE

In this section we experimentally evaluate the performance of merging simulations. We observed in earlier experiments with cloning that the performance depends significantly on when in simulated time a simulation is cloned (or in this, case merged). Performance is maximized when cloning occurs late in a simulation because all computations up to that point can be shared. Conversely, we expect that merging will have the best impact on performance when merging occurs early in the simulation.

A cloned simulation passes through three distinct phases: (1) before cloning (2) diffusion (before the simulation is fully replicated) and (3) fully replicated. When merging is enabled, the simulation transitions back from fully replicated to partly replicated and finally, fully reconverged. Merging is most beneficial the earlier the merging point.

Figure 1: Sequence of Events in Simulation Cloning and Simulation Cloning with Merging: Top: A simulation that can be merged benefits from sharing computations after the simulation converges and merges into one single simulation. The LPs are incrementally merged. Bottom: A simulation that can be merged but that remains cloned cannot benefit from shared computations.
The performance of merging was evaluated using the benchmark application P-Hold. P-Hold provides synthetic workloads using a fixed message population (Fujimoto 1990). Each LP is instantiated by an event. Upon instantiation, the LP schedules a new event with a specified time-stamp increment and destination LP. In all our experiments P-Hold is configured with 512 logical processes and a message population of 4,096. The sender is configured to choose a recipient LPs from a uniform distribution. We conducted the experiments on an SGI Origin 2000 with 24 X 300 MHz MIPS R12000 processors with 4MB cache memory and 8 GB of system main memory. All experiments used two processors.

Performance improvement due to merging over pure cloning can be measured in terms of the progress of virtual time as a function of real time, i.e. the independent variable for this experiment is execution time and the metric is simulation progress (virtual time). Larger values indicate better performance. For this experiment the simulation is run for a total of 2,000 simulated seconds. The simulation is cloned when approximately 25% of the simulated time is complete (i.e. at simulation time 500). We manipulated the simulation application so that LPs could merge at a specific point in simulated time. We ran three experiments, with three different merging times. Performance in these simulations is illustrated in Figure 2.

The plot in Figure 2 shows three curves: the left-most curve shows a run where merging occurs at simulated time 600, the middle curve shows a run where merging occurs at simulated time 1,200 and the right-most curve merging occurs at simulated time 1,600.

We observe that performance degrades upon instantiation of new clones. This is evident by a shallower slope of the curves at the cloning time. The simulations then continue with linear progress after simulated time 500. All three curves show similar performance until simulated time 600, when the left-most curve accelerates. This is a result of merging starting at time 600 and completing at time 800. The acceleration of simulated time for the middle curve is noticeable at simulated time 1,200 and the right-most curve at simulated time 1,600.
Figure 2: Performance of P-Hold. Smaller numbers indicate better performance.

In these experiments, in the best case, merging improves performance by about 24% over a non-merged simulation (which ran for 418 seconds, but is not shown in the plot).

In a second experiment, we investigate the impact of merging on speedup in the performance on P-Hold. For comparison, a pure cloning scheme (i.e. no merging), is compared against merging. Performance is evaluated as speedup of the run time of merging versus pure cloning. Speedup is the total running time of pure cloning divided by the running time of merging.

The independent variable in this experiment is the merging point. The plot of this experiment is shown in pure cloning for all merging points. As expected, merging is more beneficial the earlier the
merging point. The results show that merging can achieve a speed up of up to 2.5 (when the merging point is instantiated at the time of cloning).

![Graph showing speedup over simulation time](image)

**Figure 3:** Speedup of cloning with merging over pure cloning with P-Hold. Larger numbers indicate better performance.

The plot in Figure 4 shows the scalability of merging with respect to the number of clones. Just as in the second experiment, the performance of pure cloning is compared with merging. The number of clones was varied from one to six and the speed up was measured in each case. We can clearly see an increased performance benefit as the number of clones is increased. Thus we can conclude that merging is scalable with respect to the number of clones.
Figure 4: Scalability of Merging. Figure shows speed up of merging over pure cloning with increasing number of clones. Larger numbers indicate better performance.

The benefit of merging depends on the frequency of state comparisons and the size of the state. It should be noted that in our initial implementation there is an overhead associated with merging that we measured in the worst case. Merging overhead averages 4% of the application run time. The worst case occurs when the application test for merging fails frequently because the state is still diverged.
Simulation cloning has been shown to improve performance of interactive applications such as simulations of the ground transportation (Schulze et al. 1999; Schulze et al. 2000) and air-traffic control (Hybinette and Fujimoto 2001). Cloning and extensions of cloning has been implemented both in conservative (Chen et al. 2003; Schulze et al. 1999; Schulze et al. 2000) and optimistic (Hybinette and Fujimoto 2001) simulation executives.

With respect to cloning in general, related work in interactive parallel simulation include (Franks et al. 1997) and (Ferenci et al. 2002). The approach of (Franks et al. 1997) allows for the testing of what-if scenarios provided they are interjected before a deadline. Alternatives are examined sequentially using a rollback mechanism to erase exploration of one path in order to begin exploration of another. (Ferenci et al. 2002) create a new simulation from a previous base-line simulation by executing forward. They can determine if a computation can be reused by using a predicate function that is tested on the baseline simulation, however as in (Franks et al. 1997) only the testing of one alternative at a time is allowed. A drawback of this latter approach is that one must manage the entire state-space of the baseline simulation.

Cloning has been used to improve accuracy of simulation results, i.e., to run multiple independent replications then average their results at the end of the runs. For example, in (Glynn and Heidelberger 1991) research focuses on how to achieve statistical efficiency of replications spread on different machines. In (Vakili 1992) replications are synchronized via a shared clock. In this way the same event occurs at the same time at all replicas. Cloning is also used in sequential simulations to propagate faults in digital circuits (Lentz et al. 1999), modeling of flexible manufacturing systems (Davis 1998) or to fork
transactions in simulation languages (Henriksen 1997). Sequential simulation languages typically clone
dummies ("shadows") to do time-based waiting (Schriber and Brunner 1997).

Our approach that avoids redundant computations is similar to approaches (but in the reverse) that
reduce the cost of rollbacks: Lazy cancellation (Gafni 1988), Lazy re-evaluation (West 1988) the
aggressive no-risk (ANR) (Dickens and Reynolds 1990) protocol or exploiting "lookback" (Chen and
Szymanski 2003) for example. Lazy cancellation uses message comparisons and consequently delays
sending anti-messages until the event that originally scheduled them is guaranteed not to be regenerated.
Lazy re-evaluation is a technique that avoids state recomputation when process state is un-altered after a
rollback. ANR delays delivering messages until the event that created them is guaranteed not to be rolled
back (the sender event), this technique avoids cascading rollbacks. Lookback is a technique that identifies
situations where one can avoid rollback.

Our approach is applicable both to optimistic and conservative protocols. It uses state and
message comparison as in the lazy cancellation and revaluation protocols. However, the number of state
comparisons is minimal and only occurs infrequently. After merging, it uses just-in-time cloning
(Hybinette 2004), which uses message comparisons, that was proposed earlier, to avoid cloning
immediately after an LP is merged, but it only compares messages if the LP is again subject to replication.
Merging parallel simulations enables cloned simulations to recapture computational time wasted by unnecessary clones in a cloned or replicated simulation. The efficiency of cloning is provided by shared computations before the LPs are replicated. Merging enables simulations to recapture the efficiency of shared computations after LPs merge.

We demonstrated the potential benefit of merging using the parallel simulation benchmark, P-Hold. The impact of merging on performance depends on when in the simulation merging occurs. Results indicate that merging can improve the performance of simulation cloning by a factor of 2.5 when merging occurs at an advantageous time. At best, when a clone is immediately followed by a merge operation the overhead is insignificant. Over all run time is virtually the same as that for a single simulation run. These initial performance results indicate that merging may significantly reduce the time required to compute multiple simulations.

At present we have only evaluated the approach in a synthetic application. In future work we will investigate performance in more typical simulation tasks, such as ground and air transportation (e.g. D-PAT (Wieland 1998)). Additionally, we assume that the cost of state comparison is low. This may be reasonable since we only need to compare state at irregular, user defined intervals. In future work we will examine the impact of varying the state size and also explore heuristic methods, such as hierarchical comparison of data specified by the simulation application designer. The idea is that the designer could define an ordered comparison of the LP state data in order to make the comparison run very efficiently (by efficiently, we mean able to come to a negative decision quickly, since this will be presumed to be the
most common case). One adaptive approach we are considering is to record the locations within the LP state which caused the most recent failures, and to check those locations first.

A final point of interest for future work concerns the issue of how similar simulations must be in order for them to be considered “equivalent” and thus eligible for merging. It may be that relaxing the requirement that LP state must be identical will provide even more speedup. It may be that portions of the state have significant tolerance for variation that will have no significant impact on the simulation outcome. If so, a designer could specify such tolerances through an API.
REFERENCES


completion time constraint. *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 1,
1, 3–23.

Conference* (December 1997), 559–566.


environments for testing. *Journal of Electronic Testing: Theory and Applications* 11, 3 (December),
247–262.

22.

models: New approaches and possibilities through hla. In *Proceedings of the 1999 Winter Simulation
Conference* (December 1999), 1602–1609.

SCHULZE, T., STRASSBURGER, S., AND KLEIN, U. 2000. Hla-federate reproduction procedures in
public transportation federations. In *Proceedings of the 2000 Summer Computer Simulation
Conference* (July 2000).

VAKILI, P. 1992. Massively parallel and distributed simulation of a class of discrete event systems: a
different perspective. *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 2, 3,
214–238.

of Calgary.