Proceedings of the 2004 Winter Simulation Conference
R .G. Ingalls, M. D. Rossetti, J. S. Smith, and B. A. Peters, eds.

INCREMENTAL HLA-BASED DISTRIBUTED SIMULATION CLONING

Dan Chen
Stephen John Turner
Wentong Cai

School of Computer Engineering
Nanyang Technological University
639798, SINGAPORE

Boon Ping Gan
Malcolm Yoke Hean Low

Singapore Institute of Manufacturing Technology
638075, SINGAPORE

ABSTRACT

Distributed simulation cloning technology is designed to analyze alternative scenarios of a distributed simulation concurrently within the same execution session. The goal is to optimize the execution time for evaluating different scenarios by avoiding repeated computation. In terms of High Level Architecture (HLA) based simulations, a federate may make clones to explore different scenarios at decision points. It is desirable to use an incremental cloning mechanism to replicate only those federates whose states will be affected. This paper discusses the theory and issues involved in incremental distributed simulation cloning, which employs an event checking algorithm to ensure accurate sharing and initiates cloning only when absolutely necessary. Experiments have been performed to compare the performance of entire cloning and incremental cloning mechanisms. The experimental results indicate that cloning technologies can effectively reduce the time of executing multiple scenarios, and the incremental cloning mechanism significantly surpasses entire cloning in execution efficiency.

1 INTRODUCTION

Distributed simulation technology facilitates the construction of a large-scale simulation with simulation models (federates) distributed geographically. The High Level Architecture (HLA) defines the rules and interface specification to support reusability and interoperability amongst the simulation federates. The Runtime Infrastructure (RTI) software supports and synchronizes the interactions amongst different federates conforming to the standard HLA specification (Dahmann, Kuhl, and Weatherly 1998).

Using traditional simulation technology, in order to examine alternative decision policies, an analyst has to repeat executing a simulation to collect multiple sets of results for analysis. Basically this task is time-consuming and onerous in which a lot of computation is repeated unnecessarily. Especially for a large-scale distributed simulation, it can be costly to reconfigure and execute the overall simulation again and again due to the complexity and distribution of the individual simulation federates.

When reaching a decision point, a federate has different choices to examine. Instead of simulating each choice from the start in a conventional manner, distributed simulation cloning technology can be used to replicate the federate and allow the replicas to examine these choices concurrently from the decision point onwards. Thus the execution time can be reduced and the analyst may quickly obtain multiple sets of results that represent the impacts of alternative decisions. One important goal of cloning technology is to optimize execution by avoiding repeated computation amongst independent scenarios.

In this project, we have enabled cloning of HLA-based distributed simulations using a decoupled federate architecture (Chen et al. 2003a, 2004). When a federate makes clones on its own initiative and creates new scenarios, other federates in the original scenario have to interact with each of these clones properly in the new scenarios. One direct solution is to clone all other federates immediately, thus each independent set of clones form a new standalone scenario. Therefore a full set of independent federates are exploited to examine each scenario after cloning. However, when performing distributed simulation cloning, it is desirable to replicate only those federates whose states will alter at a decision point. The remaining federates may keep intact and become shared between the original scenario and the new ones; only when absolutely necessary will those shared federates be cloned. Hence such an incremental simulation cloning mechanism is expected to further share computation amongst scenarios (Chen et al. 2003b).

Sharing federates in different scenarios needs accurate control to achieve correctness and efficiency. Clones developed from the same federate (sibling clones) may have different impacts on those shared federates. An event checking algorithm is designed for shared federates to deal with events from multiple scenarios, which checks whether the events from sibling clones are identical or not. The

checking determines whether a shared clone remains shared or requires cloning. It guarantees the simulation results of alternative scenarios obtained using distributed simulation cloning technology are the same as those obtained by repeating simulation executions.

The rest of this paper is organized as follows: Section 2 addresses related work and introduces some basic concepts as well as the theory and issues involved in incremental cloning. Section 3 gives an overview of the proposed cloning technology. Section 4 details the algorithms for managing shared clones. A distributed simulation example is presented in section 5, which compares the performance of using entire cloning technology with incremental cloning technology. In section 6, we conclude with a summary and proposals on future work.

2 DISTRIBUTED SIMULATION CLONING

Hybinette and Fujimoto (2001) proposed using simulation cloning technology as a concurrent evaluation mechanism in the parallel simulation domain. This technique aimed to develop a parallel model that supports an efficient, simple, and effective way to evaluate and compare alternative scenarios. The method was targeted for parallel discrete event simulators that provide the simulation application developer with a logical process (LP) execution model. In Hybinette and Fujimoto (2004), they suggested a just-in-time cloning mechanism to avoid unnecessary cloning of a LP as long as it keeps receiving identical messages from other replicated LPs.

Schulze, Straßburger, and Klein (2000) introduced a cloning approach to extend the flexibility of system composition to run-time. Their approach included the parallel management of different time axes in order to provide forecast functionality. Internal cloning and external cloning techniques were suggested to clone federates at run-time.

Our design targets users who may have their own existing complex simulation models; thereby we have the additional aim to provide reusability and transparency while enabling simulation cloning. Our research and discussion are based on HLA-compliant distributed simulations. We also need to support easy utilization and deployment. Our approach focuses on optimizing and controlling a large-scale distributed simulation using the cloning technology.

2.1 Concepts and Definition

A federate may make clones on its own initiative to explore different scenarios when it reaches a decision point, and such a federate is said to perform **active cloning**. An active cloning results in the creation of new scenarios. Other federates who interact with this federate may have to spawn clones to perform proper interaction with each of the replicas, and those federates are said to perform **passive cloning**.

We can perform passive cloning on all other federates immediately in the scenario created as a result of active cloning, this approach is known as **entire cloning**. Alternatively **incremental cloning** only requires cloning those federates whose states will alter at this decision point. As for those federates whose states are not affected, the incremental cloning mechanism allows them to operate in the new scenarios in addition to the original one as **shared federates** (clones).

The clones created from the same root federate are referred to as **sibling clones**. Those federates that interact within the same scenario are known as **partner federates**. Figure 1 illustrates the different effects of an active cloning using entire cloning versus incremental cloning.

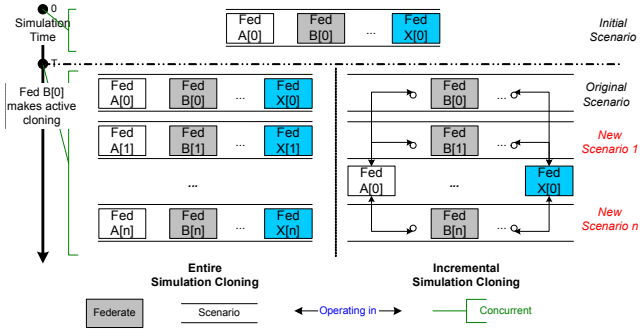

Figure 1: Entire Cloning vs. Incremental Cloning

In order to manage concurrent scenarios within a single federation, we use Data Distribution Management (DDM) to partition scenarios (Chen et al. 2003b). Each scenario is specified with an exclusive **characteristic point region** which is associated to the clones that operate in the respective scenario. To provide reusability to existing simulation federates, a **middleware** approach is adopted to hide the implementation of any cloning related modules. To tackle the problems involved in replicating running federates, a **decoupled federate architecture** is used to separate the simulation model from the local RTI component (Chen et al. 2003a). A **virtual federate** is built up with the same code as the original federate, while a **physical federate** associates itself with a real local RTI component serving the virtual federate with RTI services.

A clone needs to inherit identical states from the original federate, including the RTI entities known to the simulation model, for example the registered object instances (Kuhl, Weatherly, and Dahmann 1999). We name the object instances registered by the original federate prior to cloning as **Original Object Instances** whereas we use **Image Object Instances** to denote those object instances re-registered (representing the original ones) by the clones of this federate in the state replicating procedure.

2.2 Theory and Issues in Incremental Cloning

The incremental cloning mechanism enables a shared clone to execute in multiple scenarios as long as it keeps receiving identical events from corresponding federates in all scenarios in which it participates. This design aims to avoid repeating identical computation amongst scenarios as much as possible. The shared clone persists in this mode until the condition for triggering passive cloning is met.

A typical shared clone is shown in Figure 2. The shared clone (*SC*) executes in *n* concurrent scenarios, and those scenarios are said to be *SC*'s **related scenarios** (written as $RELASCEN = \{relaScen[i] | i = 1,2, \ldots, n\}$). Let $X = \{x[i] | i = 1,2, \ldots, n\}$ denote the set of sibling clones that are created from the same simulation federate x, with $x[i]$ operating in $relaScen[i]$. *SC* may receive events from $x[i]$ and generate events for each related scenario. It is unnecessary to perform extra processing on the events generated by *SC*, as those events must be identical in any scenario. However, the events received by *SC* have to be checked.

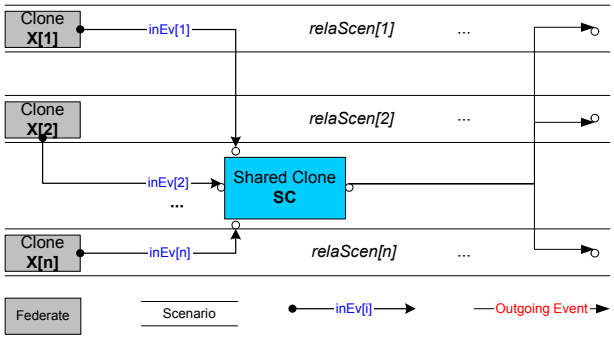

Figure 2: A Typical Shared Clone

Definition 1 (Sensitive Update) *If an object instance ObjX registered by federate x has been discovered by SC, then SC treats ObjX and its image objects (see section 3.2) as a set of sensitive object instances. Obviously, the object class to which ObjX belongs must be published by x and subscribed by SC. Let inEv[i] represent an update of ObjX (or its image objects) issued by any* $x[i] \in X$*, then inEv[i] is defined as a* ***sensitive update*** *for the shared clone SC.*

Definition 2 (Sensitive Interaction) *Any interaction class published by x and subscribed by the shared clone SC is regarded as a sensitive interaction class. Let inEv[i] represent an interaction of any sensitive interaction class sent by any* $x[i] \in X$*, then inEv[i] is defined as a* ***sensitive interaction*** *for the shared clone SC.*

A **sensitive event** is defined as a sensitive update or interaction. A shared clone may present non-sensitive events straightforwardly to its simulation model without extra checking, whereas it has to check each sensitive event before conveying it to the simulation model. A non-sensitive event can be an event sent by another shared clone executing in all related scenarios of the receiver. A sensitive event needs to be compared with corresponding counterpart events. In each round of event comparison, the first received sensitive event is referred to as the **target event** by subsequent counterpart events.

Definition 3 (Comparable Updates) *Any two sensitive updates for a shared clone are* ***comparable*** *to each other only when following conditions are satisfied:*

- *They carry equivalent timestamps.*
- *They are updates of two individual image objects (or an original object and one of its image objects) representing the same original object.*

Definition 4 (Comparable Interactions) *Any two sensitive interactions are* ***comparable*** *only when following conditions are satisfied:*

- *They carry equivalent timestamps.*
- *They belong to the same sensitive interaction class.*
- *They originate from two individual sibling clones.*

A shared clone should not compare received interactions that are not sent by sibling clones even if they belong to the same interaction class. According to definition 3 and the definition of original and image object instance, it is obvious that comparable events must originate from sibling clones.

Definition 5 (Identical Events) *Comparable events are called* ***identical*** *if they have the same associated attributes/parameters and the values of all attributes/parameters are identical.*

Comparable events need to be checked to verify whether they are identical. If a shared clone detects any two comparable events are not identical, the shared clone has to perform passive cloning to handle this situation. On the other hand, the shared clone may remain intact if:

- All received comparable events are identical.
- The shared clone receives comparable events from all the sibling clones in the related scenarios before it is granted a simulation time greater than (or equal to) the target event's timestamp.

If the second condition is not met, it means that the shared clone has obtained different behaviors from related scenarios and requires passive cloning. As a consequence, the federate previously shared and each of its clones created in this passive cloning, operate as a normal clone in only one individual scenario (at least until the next decision point). **Normal clone**s are those clones that operate in a single scenario (e.g. *x[i]* in Figure 2); this term is used in this paper to distinguish them from shared clones.

2.3 Example of Incremental Cloning

Figure 3 illustrates a simple supply chain simulation comprising three federates, namely *simAgent (SA)*, *simFactory(SF)* and *simTransportation(ST)*. Two object classes "*Order*", "*Products*" and one interaction class "*deliveryReport*" are defined to represent the types of events exchanged amongst the federates. A cloning trigger is predefined for federate *simFactory*, which contains a cloning condition "*OrderSize* > *MAX*?" and several candidate policies. The simulation emulates the supply chain operation of one-year duration. The *simFactory* reports the cost incurred in each order and in the whole year at the end of simulation.

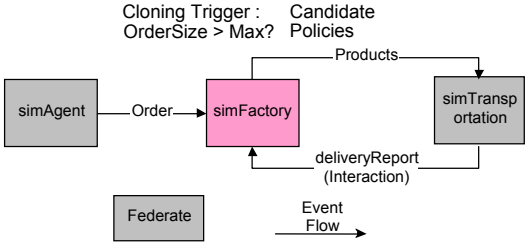

Figure 3: A Distributed Simulation Example

Figure 4 depicts the simulation execution using incremental simulation cloning to examine three candidate policies. Each scenario is marked as *Scen[i]* (i = 0, 1, 2), in which *Scen[0]* denotes the initial scenario. The incremental simulation cloning occurs along the time axis as follows:

At time 0, the simulation initializes with a single scenario *Scen[0]*. When simulation progresses to time T_1, *SF[0]* performs active cloning due to an order with extra large volume, which results in the creation of clones *SF[1]* & *SF[2]*, and new scenarios *Scen[1]* & *Scen[2]*. The remaining federates do not need to be cloned immediately, and they only need to expand their associated region to enable them to continue interacting with *SF[1]* & *SF[2]*. Thus *SA[0]* and *ST[0]* become shared clones in both scenarios. The event flow from *SF[i]* (i = 0,1,2) to *C[0]* is named as *ev_F[i]* (i = 0,1,2). *ST[0]* keeps intact as long as *ev_F[i]*(i = 0,1,2) remain identical.

At simulation time T_2, *ev_F[0]* deviates from *ev_F[1]* and *ev_F[2]*, this triggers a passive cloning of *ST[0]* and results in the creation of clones *ST[1]* & *ST[2]*. This passive cloning does not trigger any change of existing scenarios. *SA[0]* persists as a shared clone after that.

Clones are created incrementally according to the changing external conditions. We always have: *Total no. of federates* $\leq \Sigma$*No. of federates executing in each scenario*. For example, from simulation time T_0 to T_1 there exists only 5 federates simulating 3 scenarios whereas there has to be 9 federates examining the same scenarios in the context of traditional distributed simulations or using the entire cloning approach. Both cloning approaches avoid repeating the computation of the original scenario before cloning in the new scenarios. Moreover, the incremental cloning approach enables sharing computation amongst independent co-existing scenarios after cloning.

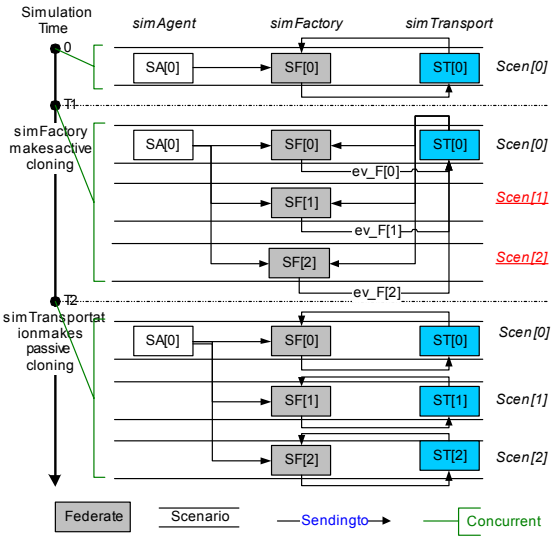

Figure 4: Execution with Incremental Cloning

3 OVERVIEW OF THE DISTRIBUTED SIMULATION CLONING ARCHITECTURE

Cloning of HLA-based distributed simulations has been enabled using a decoupled federate architecture. This section gives an overview of the design of the modules supporting cloning and introduces the cloning algorithm including the entity mapping approach.

3.1 Modules

A RTI++ library to enable simulation cloning is built as the middleware between the simulation model and the real RTI, and performs the necessary functionalities related to simulation cloning. The user can specify the conditions according to which the cloning should be triggered and the different actions to be taken. Figure 5 gives an overview of the RTI++'s structure and internal modules. The **Control Module** monitors the states in which the user is interested and evaluates the conditions for cloning the federate at a decision point. The **Cloning Manager** module **(CMM)** creates new clones for the request issued by the Control Module, and it initiates the creation and update of the scenarios (Chen et al. 2004). The **Scenario Manager** module creates and stores the scenario tree. The **Region Manager** module creates DDM regions and manages the regions. The RTI++ services invoked by a federate are eventually executed by the physical federate that calls the real RTI services and conveys callbacks to the RTI++ middleware.

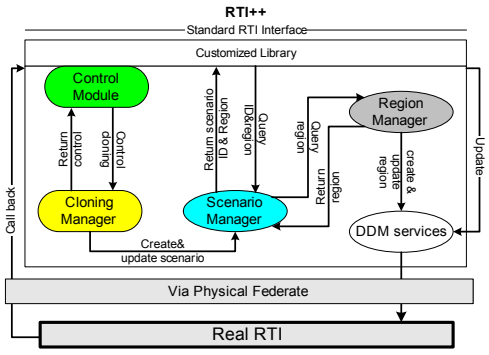

Figure 5: RTI++ and Internal Modules

The internal components inside the CMM are highlighted in Figure 6. The **Cloning Executor** answers the cloning request issued by the Control Module, and makes replicas of the simulation model and initiates new physical federate instances. The **RTI States Manipulator** saves RTI states and replicates them on simulation cloning. The Cloning Executor directly replicates static states for new clones, while the **Buffer Manager** takes charge in copying dynamic states such as managing the replicated RTI entities. During cloning of a federate, the **Federation Coordinator** should synchronize other federates within the whole simulation run including the sibling clones.

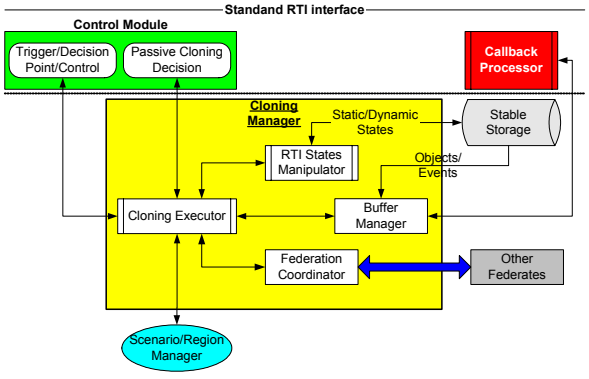

Figure 6: Cloning Manager Module

3.2 Mapping Entities

A federate simulation model obtains the information in the HLA object model via RTI services using handles assigned by the RTI. For example, when an object instance is registered, a federation unique handle is returned to identify that object instance. This handle is used to represent an entity known to the model as well as other federates who have discovered this object instance.

A clone inherits identical states from the original federate, including the RTI entities known to the simulation model. In order to keep the state consistent and federate code transparent, our cloning approach needs to ensure that the clones of a federate use the same reference to the original entities at the RTI level as before cloning. The approach should correctly manage the interactions related to these entities within the overall federation, for example a shared clone may receive updates of different object instances even though they refer to the same object in the simulation model.

The consistency can be achieved using a mapping approach in the middleware. The middleware maps the original handles with the image object handles to ensure user transparency and consistency. For one original object instance referred to by the simulation models of all clones, there can be different image object instances accessed by the physical federates. The middleware keeps transparency of image objects in the simulation model. The same principle is applied in processing other entities at the RTI level.

4 MANAGING SHARED CLONES

A shared clone is capable of operating in multiple scenarios as long as it keeps receiving identical events from all scenarios in which it participates. The shared clone persists in this mode until the condition of triggering passive cloning is met. Thus during this time, the computation of this clone can be shared by different scenarios. The incremental cloning mechanism aims to make full use of the interdependencies amongst related scenarios, which is supported by a sensitive event checking algorithm.

Sensitive events are checked by the Callback Processor which is one part of the RTI++ middleware built upon the decoupled federate architecture (Chen et al. 2003a). Figure 7 illustrates the primary elements inside the Callback Processor designed for checking events. The Sensitive Event Checker checks events and invokes the external control module to trigger passive cloning when necessary. Mapping Tables maintain the relationships amongst scenarios and federates and object instances (original/image) registered by related sibling clones. These tables are established and updated during the state replicating procedure on cloning. The checker can identify the source clone and scenario of each event via the tables, thus it can verify which events are comparable.

A queue *Pending_Sensitive_Events_Queue* (*PSEQ*) stores the **target** sensitive events with which other incoming sensitive events must be compared. The queue can be either empty or contain events with the same timestamp at any point in the simulation, this timestamp is referred to as the characteristic timestamp of *PSEQ*. A set of TSO event queues, *TSO_Queue_Scen[i]* or *TQS[i]* ($i = 1, 2, \ldots, n$), are established to buffer the events from each scenario in the corresponding queue. Events in those queues can be presented to the simulation model if and when necessary.

The event checking algorithm determines which events and how these events should be conveyed to the simulation model. The event checking decides whether or

not a passive cloning is required and at which point the cloning should be triggered. A shared clone is said in to be in **pending-passive-cloning** mode during the interval from deciding that a passive cloning is required to carrying out the cloning. Event checking is performed when control of a federate process is still with the RTI. Thus cloning will only be carried out when the RTI returns control to avoid potential problems incurred by replicating a federate while the RTI invokes callbacks.

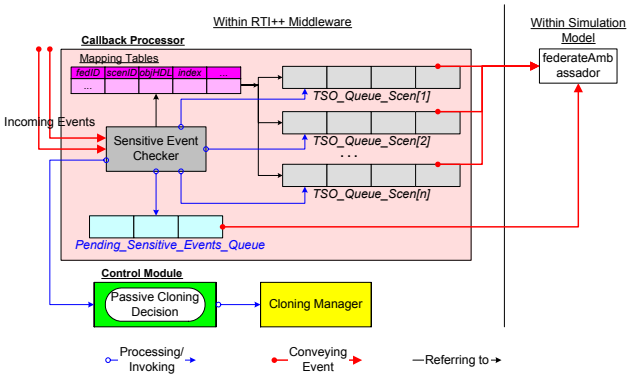

Figure 7: Internal Design of the Callback Processor

Figure 8 illustrates the algorithm of checking sensitive events by the Callback Processor which are as follows:

- **Testing sensitive event**. A received update/interaction is tested according to definition 1 and 2. In case the event is a sensitive one, the checking continues. The processing of non-sensitive events will be covered later.
- **Identifying event source**. Mapping Tables are referenced to locate the source of this event. Hence the event checker can enqueue this event into the corresponding *TQS* queue.
- **Checking pending sensitive event queue**. If *PSEQ* is empty, the event checker pushes this event into *PSEQ* and sets its timestamp equal to the event's, after which current processing ends. When *PSEQ* is not empty, the event checker compares its characteristic timestamp with the event's. In case they are not equal (event's > *PSEQ*'s), the shared clone will enter pending-passive-cloning mode, otherwise the processing continues.
- **Locating target comparable event**. The event processor searches *PSEQ* to locate the comparable event to the event in processing. If *PSEQ* does not contain any comparable event, the event will be pushed into *PSEQ* and the processing ends. Otherwise, the event processor checks if the received event and the target event in *PSEQ* are identical. If they are not identical, the shared clone will also enter pending-passive-cloning mode, otherwise the processing continues.
- **Checking the progress status of processing**. The event checker examines whether the shared clone has received identical comparable events from all other scenarios. If so, the event processor removes the targets event from *PSEQ* and flushes the comparable events inside the *TQS* queue set. If not, then the event checker waits for the next event.

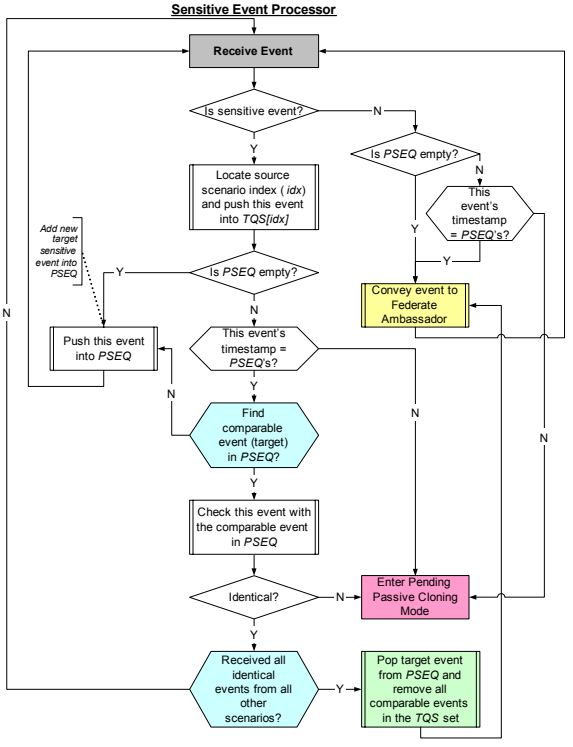

Figure 8: Processing Sensitive Events

The last step ensures the Federate Ambassador reflects only one single event for one full set of identical events obtained from all related scenarios. This design hides the complexity of checking events from multiple scenarios. As a result, shared clones operate in multiple scenarios as if they only interact with one single scenario independently.

In case a non-sensitive event is received, the *PSEQ*'s characteristic timestamp also needs to be compared when *PSEQ* is not empty. If the event's timestamp is greater than the characteristic timestamp, the shared clone requires passive cloning. This is because the shared clone will no longer receive identical events from any of the related scenarios to the target events in *PSEQ*. In the case that their timestamps are identical or *PSEQ* is empty, the event processor delivers this event to the Federate Ambassador directly.

If *PSEQ* contains target sensitive events, the decision of triggering a passive cloning depends on both the incoming events and the next granted time. Once the Callback Processor gets a granted time greater than *PSEQ*'s time-

stamp, the shared clone enters pending-passive-cloning mode. If the granted time is equal to PSEQ's timestamp, setting pending-passive-cloning or not depends on whether the shared clone requests the last time advance by calling timeAdvanceRequest (TAR) / nextEventRequest (NER) or timeAdvanceRequestAvailable (TARA) / nextEventRequestAvailable (NERA).

When a shared clone is in pending-passive-cloning mode, the Callback Processor buffers the incoming events as illustrated in Figure 9. Sensitive events should be enqueued to the corresponding *TQS* queue, whereas non-sensitive events should be inserted into all *TQS* queues unselectively. The Callback Processor logs the *timeAdvanceGranted* callback and the granted time. Preparations for the coming passive cloning are made in this procedure; for example, the received events are sorted according to their source scenarios. All callbacks are retained and not delivered to the simulation model until the pending cloning has been completed. Such a design aims to keep the semantics of the HLA specification and minimize the complexity of dealing with potential callbacks during pending-passive-cloning.

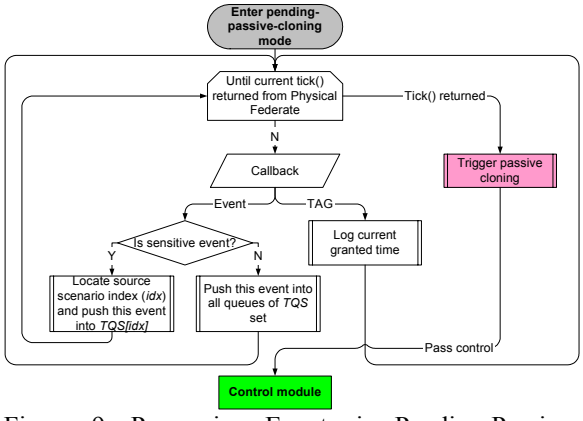

Figure 9: Processing Events in Pending-Passive-Cloning

5 EXPERIMENTS AND RESULTS

In order to investigate the performance of the proposed incremental cloning mechanism, a series of experiments have been carried out to compare the execution time of running different scenarios of a simple distributed simulation example using conventional federates and cloning-enabled federates.

The experiments adopt the simulation example shown in Figure 3, which starts at simulation time 0 and ends at 361 with one time unit representing one real day. The experiments use four computers in total (three workstations and one server), which are interlinked via a 100Mbps-based backbone. Each federate (together with its clones if any) occupies one individual computer, with the RTIEXEC and FEDEX processes running on another computer. We have adopted DMSO RTI NG 1.3 V6 in building the federates and executing experiments.

The experiment architecture and platform specification are listed in Table 1. Using the same codes, the federates are built into three different versions by linking to: (1) the DMSO RTI library directly (Traditional), (2) an RTI++ middleware library supporting entire cloning (Cloning_Entire) and (3) an RTI++ library supporting incremental cloning (Cloing_Incremental). Experiments are carried out to collect the overall execution time using different types of federates to analyze different policies (listed in Table 2).

Table 1: Configuration of Experiment Test Bed

Specification	**Computers**		
	Work-station1~2	Server1	Work-station3
Operating System	Sun Solaris OS 5.8	Sun Solaris OS 5.8	Win2000 Pro
CPU	Sparcv9 CPU, at 900 MHz	Sparcv9 CPU * 6, at 248 MHz	Intel 1700 MHz Pentium IV
RAM	1024M	2048M	256M
Compiler	GCC 2.95.3	GCC 2.95.3	MS VC++ 6.0
Processes Running On	*SimAgent, simTrans-portation*	*simFactory*	RTIEXEC & FEDEX

Table 2: Experiments for Studying the Efficiency of Cloning Technology

Type of Federates	**Experiments**		
Cloning Stage	Start (Middle, End)		
Number of Policies	2	3	4
Cloning_ Entire	*Ec_s(m, e)_2*	*Ec_s(m, e)_3*	*Ec_s(m, e)_4*
Cloning_ Incremental	*Ic_s(m ,e)_2*	*Ic_s(m, e)_3*	*Ic_s(m, e)_4*

For traditional federates, we execute the policies one by one, after which the sum of the execution times of the runs are calculated. As for experiments with cloning-enabled federates, we let federate *simAgent* generate different orders. We select three runs in which cloning of *simFactory* occurs at time 80, 203 and 320, thus federate *simFactory* may trigger active cloning at different stages in each run. These points represent cloning at the start, middle and end stages respectively. Furthermore we also specify *simFactory* to make different numbers of clones to examine alternative policies in different experiments (2, 3 or 4 policies in each run). For example (see Figure 2 and 4), in experiment *Ec_m_3* federate *simFactory* creates 2 clones at time

203 and *simAgent* together with *simTransportation* also create 2 clones immediately to explore 3 scenarios with 9 federates in total (entire cloning); whereas in experiment *Ic_m_3* federate *simAgent* keeps intact all the time and *simTransportation* remains shared until simulation time 224 and performs passive cloning to create 2 clones (incremental cloning).

The CPU utilization of a single federate (in workstation 1 and 2) is reported as ~80%. In the case of enabling simulation cloning, the CPU utilization of each clone is reported as ~44%, ~30% or ~21% respectively when there are 2, 3 or 4 clones running on a single workstation. Physical federates have a CPU utilization as low as ~1%. Experimental results are recorded in seconds in Figure 10. The average time of executing one single scenario per run is 561 seconds using traditional federates. The percentage of saved execution time using cloning technology is shown in Figure 11 which use the execution times reported by traditional federates running scenarios sequentially as a reference.

Figure 10: Execution Time for Examining Multiple Scenarios

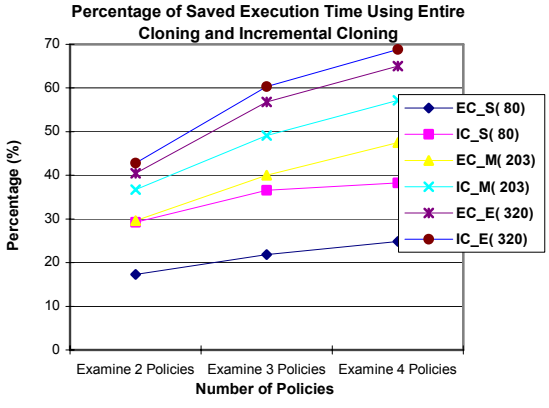

Figure 11: Percentage of Saved Execution Time using Entire and Incremental Cloning

Figure 10 shows that the cloning enabled federates can significantly reduce execution time compared with conventional federates. The experimental results indicate that the more computation there is in common amongst different scenarios the more execution time can be reduced using simulation cloning. It also shows that the larger the number of scenarios to be examined using cloning the more execution time can be reduced. The results in Figures 10 and 11 indicate that the incremental cloning approach has an obvious advantage over the entire cloning approach in terms of execution efficiency under all given configurations. This is because the incremental cloning approach can further save computation by supporting federate sharing amongst scenarios.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we have presented an in-depth study on the incremental cloning mechanism for HLA-based distributed simulations. The theory of incremental simulation cloning is detailed, and the design is also introduced. Our design enables distributed simulation cloning using a decoupled federate architecture. The cloning management module is developed to ensure correct distributed simulation cloning when preset conditions are met at decision points. The incremental cloning mechanism initiates cloning only when strictly necessary. Federate sharing amongst multiple scenarios is supported by a sensitive event checking algorithm. The algorithm facilitates accurate sharing of federates and delays the passive cloning as long as possible. The proposed mechanism supports correct HLA semantics and user transparency, and it optimizes the simulation execution for analyzing different scenarios.

A series of experiments has been performed to investigate the performance of two alternative cloning approaches. The experimental results are compared for entire cloning and incremental cloning in terms of execution efficiency. The experimental results show that the cloning technology can reduce the time of executing multiple scenarios of existing distributed simulations. Furthermore, the incremental cloning technology can further save computation of distributed simulations compared with an approach using entire cloning.

For our future work, we need to further minimize the overhead incurred in the cloning procedure, which includes investigating an efficient approach to dealing with in-transit events on cloning. Another issue is to facilitate fault tolerance using cloning technology.

REFERENCES

Chen, D., S. J. Turner, B. P. Gan, W. Cai, J. Wei. 2003a. A Decoupled Federate Architecture for Distributed Simulation Cloning. In *Proceedings of the 15th European Simulation Symposium*, 131-140. Delft, the Netherlands.

Chen, D., S. J. Turner, B. P. Gan, W. Cai, J. Wei, N. Julka. 2003b. Alternative Solutions for Distributed Simulation Cloning. *Simulation: Transactions of the Society*

for Modeling and Simulation International 79 (5-6): 299-315.

Chen, D., S. J. Turner, B. P. Gan, W. Cai, J. Wei. 2004. Management of Simulation Cloning for HLA-based Distributed Simulations. In *European Simulation Interoperability Workshop 2004*, 04E-SIW-010, Edinburgh, UK.

Dahmann, J. S., F. Kuhl, R. Weatherly. 1998. Standards for Simulation: As Simple As Possible But Not Simpler, The High Level Architecture for Simulation. *Simulation: Transactions of the Society for Modeling and Simulation International* 71 (6): 378-387.

Hybinette, M. and R. M. Fujimoto. 2001. Cloning parallel simulations. *ACM Transactions on Modeling and Computer Simulation*, 11: 378-407.

Hybinette, M. and R. M. Fujimoto. 2004. Just-in-time Cloning. In *Proceedings of 18th Workshop on Parallel and Distributed Simulation,* 45-51. Kufstein, Austrian.

Kuhl, F., R. Weatherly, J. Dahmann. 1999. *Creating Computer Simulation Systems: An Introduction to HLA*. ISBN 13-022511-8, Prentice Hall, USA.

Schulze, T., S. Straßburger, U. Klein. 2000. HLA-federate Reproduction Procedures In Public Transportation Federations. In *Proceedings of the 2000 Summer Computer Simulation Conference,* Vancouver, Canada.

AUTHOR BIOGRAPHIES

DAN CHEN was a research engineer with the Singapore Institute of Manufacturing Technology. In China, he received a BSc from Wuhan University and a MEng from Huazhong University of Science and Technology in 1994 and 1999 respectively. He received a M.Eng. and is currently a PhD student at Nanyang Technological University. His research interests are distributed simulation, networking and other related technologies. His email address is <chendan@pmail.ntu.edu.sg>.

STEPHEN JOHN TURNER is Director of the Parallel and Distributed Computing Centre in the School of Computer Engineering, Nanyang Technological University (Singapore). He received his MA in Mathematics and Computer Science from Cambridge University (UK) and his MSc and PhD in Computer Science from Manchester University (UK). His current research interests include: parallel and distributed simulation, distributed virtual environments, grid computing and multi-agent systems. His email address is <assjturner@ntu.edu.sg>.

WENTONG CAI is currently an Associate Professor at School of Computer Engineering (SCE), Nanyang Technological University (Singapore). He received his B.Sc. in Computer Science from Nankai University (P. R. China) and Ph.D. also in Computer Science from University of Exeter (U.K.). His current research interests are mainly in the areas of Parallel & Distributed Simulation and Grid Computing. His email address is <aswtcai@ntu.edu.sg>.

BOON PING GAN is a Research Engineer at Singapore Institute of Manufacturing Technology in Singapore. He received a BSc and MSc from Nanyang Technological University of Singapore in 1995 and 1998 respectively. His research interests are parallel and distributed simulation, parallel programs scheduling, and application of genetic algorithms. His email address is <bpgan@simtech.a-star.edu.sg>.

MALCOLM YOKE HEAN LOW is a Research Engineer at the Singapore Institute of Manufacturing Technology. He received his Bachelor and Master of Applied Science in Computer Engineering from Nanyang Technological University, Singapore in 1997 and 1999 respectively, and a D.Phil. in Computer Science from Oxford University in 2002. His research interests include adaptive tuning and load-balancing for parallel and distributed simulations, and the application of multi-agent technology in supply chain logistics coordination. His email address is <yhlow@simtech.a-star.edu.sg>.