Proceedings of the 1998 Winter Simulation Conference
D.J. Medeiros, E.F. Watson, J.S. Carson and M.S. Manivannan, eds.

BUILDING PARALLEL TIME-CONSTRAINED HLA FEDERATES: A CASE STUDY WITH THE PARSEC PARALLEL SIMULATION LANGUAGE

C. D. Pham
R. L. Bagrodia

University of California, Los Angeles
Department of Computer Science
Los Angeles, CA 90095, U.S.A.

ABSTRACT

Based on the DIS result, the HLA framework has been defined to achieve interoperability of independent simulators. Concurrently, and for the most part, independent of that effort, the parallel and distributed simulation community has attempted to define synchronization protocols for the correct execution of parallel simulation as-fast-as possible. Building parallel time-constrained federates within an HLA framework is not an easy task. We identify the potential difficulties: one or several federates, when and how to advance the federate's time, how to handle RTI notifications, etc., and present our experiences with adding HLA features into the PARSEC parallel simulation language.

1 INTRODUCTION

Simulation is becoming a common way to perform the evaluation and the study of a large variety of systems such as large scale computer networks or battlefield applications. As more powerful machines are available for this task, the level of attempted complexity is always increasing. Since high development costs are unavoidable, it is certainly better to maximize the reuse of models and provide interoperability instead of rebuilding new simulators for each new test case. Starting in March 1995, the U.S. Department of Defense (DoD) has devoted considerable efforts to define a common technical framework for all DoD simulation applications. This framework includes a High Level Architecture (HLA) to achieve reuse and interoperability of simulators. In the HLA terminology a single simulator is referred to as a "federate". A "federation" is then a set of federates working together to achieve a given goal. Any federate developed following the HLA guidelines, by using a common software interface, should be able to join an existing federation and its capabilities used by the other federates. HLA builds upon the results of the Distributed Interactive Simulation (DIS) experiments with the desire to incorporate advanced time management features that were lacking in the DIS approach.

Independent of interoperability efforts, the research community has also devoted considerable effort to develop parallel and distributed simulation concepts (PADS). Whereas DIS was concerned with real-time simulations, the main concern of PADS is to provide an "as fast-as-possible" execution while ensuring that the parallel simulation satisfies all time dependencies and causality constraints among events. The underlying synchronization mechanisms to achieve a correct execution can be denoted as conservative, optimistic, or hybrid. A comprehensive comparison between PADS and DIS can be found in (Fujimoto, 1995).

Interoperability of independent simulators can to a certain extent allow for the simulation of large scale systems. However, models that include a large number of objects, such as communication network models, can not be decomposed into a set of independent simulators because the cost of interoperability would be overwhelming. Also, such a large model may not be implementable on a single machine since the computation load is extremely high. The idea of bringing the previously different communities, HLA and PADS, into a closer cooperation is tempting. As a result, an attempt is being made to incorporate the PADS requirements into the definition of the time management facilities of HLA (Fujimoto, 1996). Such a cooperation would allow the creation of an HLA federation that contains a parallel federate simulating a large system that would not have been possibly simulated by a traditional sequential federate. This is fundamentally different from plugging DIS-based systems into an HLA federation because no time constraints are taken into account in DIS-based federates. Even if the HLA framework provides some services that can be useful for PADS, building time-constrained parallel federates with an environment whose primary concern is interoperability requires a careful look on how the HLA time management facilities can be handled by the parallel simulator.

The parallel and distributed simulation group at UCLA has defined the PARSEC language (Bagrodia, and al., 1998) that transparently supports several synchronization schemes on a variety of parallel architectures. The present paper describes our experiments in incorporating HLA features into the PARSEC environment to develop HLA compliant parallel simulations. In the process we identify lessons that were learned. The rest of the paper is organized as follows: Section 2 presents the HLA architecture, Section 3 discusses the process of building time-constrained parallel federates, and Section 4 presents our case study using the PARSEC language. Conclusions and future work are given in Section 5.

2 THE HLA FRAMEWORK

HLA consists of three components: (i) a set of rules for the federates and the federation, (ii) an Object Model Template (OMT) that defines how the capabilities of a federate can be described and (iii) a software component called the Run-Time Infrastructure (RTI) (DMSO RTI, 1998) that acts like a distributed operating system to provide a common programming interface. With HLA, a real system is modeled by a set of objects with a given number of attributes. Interoperability is achieved by using a subscription to or publication of object attributes. A federate that subscribes to a set of object's attributes declares its interest in receiving any updates performed by remote federates on remote instances of the given object class. Publication of attributes of a given object class means that the federate is capable of simulating the object and sending updated values for the object's attributes.

Programming interfaces of the RTI are proposed in C++, ADA and Corba IDL. We will use the C++ terminology and syntax in this paper. The RTI software is divided in two parts: one part is federate-initiated and the other is RTI-initiated. The first part is encapsulated in the `RTIambassador` class. A number of methods are available to create a federation or subscribe to some object's attributes for example. The second part is defined in an abstract `FederateAmbassador` class and the user has to define its own subclass of `FederateAmbassador`. The RTI services are divided in six categories: (i) Federation Management, (ii) Declaration Management, (iii) Object Management, (iv) Ownership Management, (v) Time Management and, (vi) Data Distribution Management. Our discussion on parallel time-constrained federates will mainly address the use of categories (i) and (v).

3 BUILDING PARALLEL FEDERATES

In the PADS terminology, a synchronization algorithm that only processes safe events is said to be conservative (Chandy and Misra, 1979). Safety is guaranteed by forcing the Logical Process (LP) to block if causality may be violated. Unfortunately, this scheme introduces deadlocks that must be avoided by sending *null-messages*. This approach usually requires input channels (ICs) between logical processes. The Earliest Output Time (EOT) for an LP is defined as the timestamp of the next possible message generated by the LP. Each IC_i has an Earliest Input Time (EIT_i) that is the EOT of the LP connected to it. The EIT for an LP is defined as the minimum of the EIT_i. Locally, the LP can process all events with timestamp lesser than its EIT. Null-messages are used in the conservative approach to propagate the EOT of LPs that do not send real messages. The value of the EOT usually includes the LP's lookahead. The larger the lookahead, the better the algorithm performs. On the other hand, optimistic approaches (Jefferson, 1985) do not search for safety but provisions are made to *roll back* to an earlier coherent state when causality is discovered to have been violated. Causality constraints are corrected on-the-fly, when the LP receives a message with a timestamp smaller than its logical time. In that case, it has to roll back, undo the false computation and use lazy or aggressive cancellations to cancel its outgone messages. Periodic check-pointing and *anti-messages* to cancel incorrect computations are then needed as a counterpart of more freedom. In addition, a *Global Virtual Time* (GVT) is required to monitor the simulation progress and to reclaim memory used by obsolete information.

3.1 Time Requirements

The RTI kernel uses a conservative approach to synchronize the different federates in the federation. Each federate may send its simulation time to the RTI so that it can compute a Lower Bound Time Stamp (LBTS) for the federation execution. This LBTS value is computed by taking into account the lookahead of all the time-regulating federates and represents "*the minimum time stamp so that it can be guaranteed that no federate will generate any more time-stamp-order events with a lower time stamp*" (DMSO RTI, 1998). In the HLA Interface Specification v1.3 and v1.2, a time-regulated federate must provide a lookahead value when it wants to become time-regulated. This was not the case in version 1.1. The definition of a federate's simulation time in a parallel federate is the minimum of the local simulation time of all the LPs in the parallel federate.

Each federate can be defined with respect to the way it manages the time with a combination of 2 booleans: the time-regulation boolean (ON/OFF) and the time-constrained boolean (ON/OFF). A time-regulated federate participates in the federation LBTS computation performed by the RTI and is allowed to send time stamp order (TSO) messages (TSO messages sent by a non time-regulated federate are transformed in receive order messages by the RTI). In other words, its EOT is taken into account. A time-constrained federate should process TSO messages in timestamp order, as opposed to receive order. The RTI will ensure that messages for a time-constrained federate will be delivered only when the federate's time has advanced sufficiently. Within the federation each federate may have to request a `timeAdvanceGrant` notification from the RTI. The purpose of this task is two-fold: (i) to inform the RTI of the federate's simulation time and, (ii) to indirectly indicate to the RTI that the federate is ready to receive TSO messages up to the requested time. With two booleans, four separate status indications for a federate can be defined. We show in Figure 1 that a federate has to request a `timeAdvanceGrant` in three cases, for different reasons.

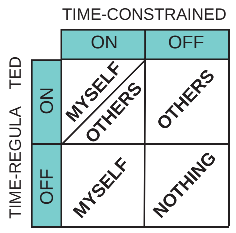

Figure 1: Federate status.

The terms MYSELF and OTHERS are defined as follows: MYSELF means that the federate needs to request a time advance grant in order to increase its simulation time as perceived by the RTI. This is needed for the federate to receive messages from the RTI when the federate is time-constrained. OTHERS means that the federate requests the time advance grant to allow other federates to advance. For instance, even if a federate is non-time-constrained but is time-regulated, it must inform the RTI of its advancement, otherwise, its non-increasing simulation time will cause the RTI to delay the delivery of the TSO messages to the other time-constrained federates. Usually, the time-regulated federate will request a time advance grant to the time of its next local event. Within the traditional definition of PADS, conservative federates must be time-regulating and time-constrained, whereas optimistic federates can be only time-regulating. However, in practice, an optimistic federate will also be time-constrained; otherwise it would encounter numerous time errors.

A given federate can obtain the value of the federation LBTS less its own EOT. In the HLA terminology, this value is called the federate's LBTS. Actually, this is the only service provided by the RTI v1.3 (`queryLBTS`), as opposed to v1.2 and v1.1 where a federate can obtain the federation LBTS (`requestFederationTime`). If we consider each federate in the federation as a conservative LP, the value provided by a given federate's call to `queryLBTS` is simply the federate's EIT from the RTI. For a time-constrained federate, it is not advisable to process events with timestamp greater than its LBTS. Exceptions may be made for federates that run an optimistic synchronization protocol.

3.2 The Need for an Accurate GVT

For a parallel federate a kind of Global Virtual Time (GVT) computation is required to determine the smallest timestamp in the federate. As opposed to a traditional optimistic approach where the GVT is mainly needed for fossil collection, the GVT computation needed for the RTI must be performed quite frequently, otherwise the federate's simulation time perceived by the RTI will not increase sufficiently. For a time-constrained federate this is a problem because TSO messages from the RTI may be delayed and never passed to the federate.

For a traditional conservative simulation without a Global Control Mechanism as described in (Jha and Bagrodia, 1994), this GVT computation must then be added. Optimistic federates that already have GVT computation features, nevertheless need to increase the rate of the GVT computation, otherwise they may block the other non-optimistic federates (conservative, or time-constrained sequential federates). The exact overhead of a frequent GVT computation is not known yet but previous studies have shown that an accurate GVT is costly. The traditional definition of the GVT for fossil collection in optimistic simulators is the minimum of all the LPs' simulation time and the timestamps of messages in transit. For an optimistic federate, its GVT_{fed} must now take into account the federate's LBTS and is then defined as $GVT_{fed} = min(GVT, LBTS)$.

3.3 One Federate or Sseveral Federates?

When incorporating a parallel simulation in a federation, an important question is whether to declare it as a single federate, or as a collection of federates. If each LP is declared as a separate federate, the advantage is that each LP can join the federation with an independent `RTIambassador` and `FederateAmbassador`. On a distributed memory machine, this is a simple way to enable the distribution of the callback mechanism—otherwise callback notifications that are beyond the boundaries of a physical processor must be translated in some way (the RTI may provide pointer arguments that are meaningless from one processor to another in a distributed memory architecture). On the other hand, having too many federates can lead to an overwhelming overhead. If hundreds of LPs are expected in a parallel simulation, having hundreds of federates is not the best solution.

However, these considerations are not strong enough to completely reject the multi-federate solution. In a large communication network model, one may still want to split the model into a relatively small set of independent federates, each one simulating part of the network. For a conservative parallel simulation, the reason to definitely reject the multi-federate model is that having several federates does reduce the amount of parallelism. Let assume that the conservative simulation consists of n LPs partitioned into m federates that have joined the federation execution, each of them being time-regulated and time-constrained as in figure 2.

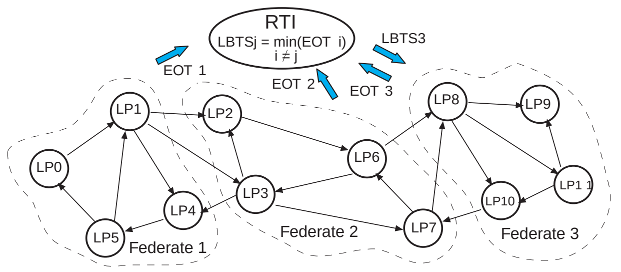

Figure 2: Dependencies in a multi-federate model.

Each federate i has a different $LBTS_i$ perceived by the RTI and has to take into account the $LBTS_{j,j\neq i}$ of the other $(m-1)$ federates. This is equivalent to adding extra ICs in the conservative simulation that did not exist in the original parallel simulation. Having several federates for a single parallel simulation introduces extra dependencies in a model that has originally been parallelized with a space-time scheme to exploit the maximum number of parallel events. In the actual implementation of the RTI, it is not possible to explicitly specify the topology, i.e. the dependencies or the lack of dependencies between several time-regulated federates. Therefore, as can be seen in figure 2, the LPs in federate 3 are dependent of the LPs in federate 1 even if there are no such dependencies in the original model.

For an optimistic parallel simulation, having several federates increases the complexity of the GVT_{fed} computation, as defined in the previous section. In a one-federate model, the native GVT computation mechanism could be used to compute both the GVT for fossil collection and the GVT_{fed} that takes into account the federate's LBTS. With a multi-federate model, these computations must be done on a federate basis, distinguishing between LPs belonging to different federates. The fossil collection process must also be done on a federate basis. All these modifications add additional complexity to an approach that is already highly complex.

3.4 Translation of RTI Notifications into Events

The large majority of PADS simulators are designed to work as event-driven simulators. Every change in the system must be made as the result of the processing of a time stamped event. The callback mechanism of the RTI must then be converted into events for the parallel federate. These events must have an associated timestamp that indicates the logical time at which it should be processed. The HLA framework defines two types of events, receive order events (RO) and time stamped order events (TSO). The RO type comes from the DIS real-time needs. Most of PADS simulators only deal with TSO messages. The translation of RTI notifications into events for the parallel federate automatically adds a timestamp to every message, even those that were originally of the RO type.

When the RTI delivers the notifications to a federate, it will call user-provided functions of the `Federate-Ambassador` base class. These functions could simply encapsulate the arguments provided by the RTI into a special event and send it to the appropriated LPs. Since each LP in the parallel federate is at a different simulation time, the notification events must be time stamped in a way to prevent any time causality violations. This is mandatory for conservative federates and advisable for optimistic federates. RO events should be time stamped with a value no lesser than the federate's simulation time. TSO notifications from the RTI carry a timestamp that should be reported in the translated event.

3.5 When and How to Advance the Federate Time?

Given the time services provided by the RTI, the parallel federate must now use them in a coordinated manner, i.e. when to advance the time and when to get messages from the RTI. We will now assume that the parallel federate is both time-regulated and time-constrained. For a conservative federate, working within an HLA federation is equivalent for each LP in having an additional IC with an EIT_{rti} from the RTI. This EIT_{rti} is simply the federate's LBTS (but only for TSO messages from the RTI!) The advancement of each LP will therefore depend on the advancement of EIT_{rti}. The EIT for an LP will then take into account the EIT_{rti} in addition to the EIT of its real ICs. Each LP can theoretically execute all eligible events from its ICs, i.e. their timestamps are smaller than the LP's EIT, and advance accordingly in simulation time. Clearly, each LP in the parallel federate can not report its local simulation time to the RTI as the federate's time. We said previously that a kind of GVT computation was needed. We will assume that one LP in the parallel simulation (a dedicated LP for the GVT computation or one among all the existing LPs) finally knows the federate's simulation time. This LP is the only one that can safely request a time advance grant and we call it LP_{TM} (Time Manager).

Requesting a time advance grant is performed mainly by three services offered by the RTI. The `timeAdvanceRequest` service is intended to be used by time-stepped simulators whereas the `nextEventRequest` is available for event-driven simulators. This latter service either issues the time advance grant to the time requested or only "*to the timestamp of the next TSO messages that will be delivered to the federate*" (DMSO Interface, 1998). There is one more service, called `flushQueueRequest`, that allows a federate to receive all the queued messages (RO and TSO), regardless of their timestamps. `flushQueueRequest` is certainly useful for optimistic federates, but appears also to be much more convenient for conservative federates as explained below.

As said previously, time advance grants are needed to increase the federate's time perceived by the RTI. The federate can then receive TSO messages from the RTI. There are two possibilities for when to request a time advance grant to time t: before the execution of the event with timestamp t or after. A sequential federate usually requests the time advance grant beforehand: assume that the next local event has timestamp t, a sequential federate would request a time advance grant for time t before executing the event. In this way, the federate can get all TSO messages up to time t from the RTI. It is the simplest way for the sequential simulator to advance the time since it does not have to worry about the LBTS. For a parallel federate, the LPs can report their advancement to time t to LP_{TM} either before or after the processing of the event of timestamp t. In a conservative federate, the knowledge of the federate's LBTS (the additional EIT_{rti} for each LP) makes each LP i regulated by the timestamp of the next TSO message from the RTI. Let's take as an example the configuration depicted in figure 3a. The LP has 3 ICs from the other LPs in the parallel simulation and has one IC from the RTI that indicates the federate's LBTS.

Assume we use `nextEventRequest` to advance the federate's time. It is still safe for the LP to report the time after since only safe events are eligible (events at timestamp 3 and 4, figure 3a). Suppose that (i) $LBTS = 5$, and (ii) the timestamp reported is the minimum timestamp in the federate: when LP_{TM} requests the time advance grant it gets it immediately. To process event with timestamp 7, reporting the value 7 before allows the increase of the LBTS and the delivery of the event with timestamp 6 from

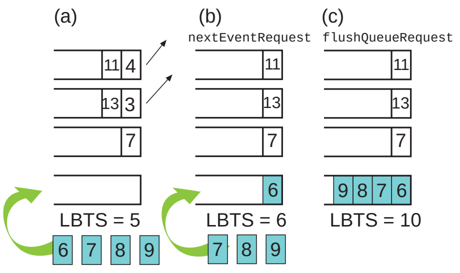

Figure 3: LP with 3 real ICs.

the RTI (figure 3b). Reporting the value 4 after blocks the LP since the event with timestamp 7 is still not eligible. An additional step is required to increase the LBTS. If the LP reports before, it does not need to wait for the time advance grant unless the time it reported is greater than the federate's LBTS. However, the main drawback of `nextEventRequest` is that it only increases the LBTS from one incoming TSO message's timestamp to another. This can decrease dramatically the number of simultaneous events.

If we consider the use of `flushQueueRequest`, the advantage is two-fold: (i) it is possible to report after without any blocking and, (ii) the LP can get as many messages as possible. There is no blocking because if `flushQueueRequest` is called, the LBTS will increase as all queued TSO messages are delivered (see figure 3c). The only blocking situation is when the increase of the LBTS is delayed by some reason (messages from the RTI are delayed on the network for example). Reporting after has also the following advantage: in an existing simulator, reporting after requires no changes in the simulator kernel because the additional code for HLA compatibility can be added as user-code. This is important for us in the particular case of PARSEC as it will be explained later. Of course, reporting before is always possible. Whenever the report is done, generally speaking, `flushQueueRequest` appears to be more convenient because the TSO messages from the RTI are directly handled by the conservative synchronization algorithm of the federate.

For an optimistic federate, the time advance mechanism is a little bit simpler: the GVT_{fed} for the federate must be computed first and a time advance grant to the GVT_{fed} requested. No rollback should occur before GVT_{fed}. The time advance grant to time t can be requested before or after the processing of the event of time t, but as opposed to conservative federates, no care needs to be taken to get incoming TSO in time because the federate can always roll back on time errors. However, unless the optimistic federate monitors the federate's LBTS (and behaves much like a conservative federate), requesting the time advance grant after would certainly increase the probability of receiving a TSO messages in the federate's past.

3.6 What about the Speedup Finally?

The purpose of creating parallel federates is to plug into an HLA federation as-fast-as possible simulators that can handle a large and complex system in a shorter amount of time than a sequential federate. Parallel simulation in the PADS community has already demonstrated that such speedups can be obtained. The question now is whether speedup can still be obtained within an HLA federation. First, speedup in this context refers to the comparison between the execution time of a federation consisting of only sequential federates and the execution time of a federation, performing the same task, where some federates are parallel. With non time-constrained and non time-regulating federates, a speedup can certainly be easily achieved, provided that the level of parallelism is sufficient. For time-constrained and time-regulating federates, even if their parallel implementations perform well outside the HLA framework, it is not necessary that they still perform well within an HLA federation. As can be seen from the previous sections, the services offered by HLA are numerous but they have to be used in an optimal way in order to give the parallel federate as much freedom as possible.

At the moment, there is no common definition of what a simulation time represents for a simulator so the simulation time can have a completely different meaning from one federate to another. If we can define a common semantic for time, still one of the problems introduced by the cooperation of several different simulators is the time scale difference between them. The time scale difference can have a dramatic impact on the execution of a parallel federate. For a conservative federate, the existence of at least one federate working at a much smaller time scale has disastrous consequences on the performance of the simulator. For an optimistic one, the probability of time errors increases dramatically. Special care must be taken when constructing a federation execution to verify that there is no time scale incompatibility, in which case the idea of interoperability is impossible.

4 CASE STUDY WITH THE PARSEC LANGUAGE

PARSEC is a C-based language that was designed to neatly separate the simulation model from the underlying algorithm (sequential or parallel) that may be used to execute the model. It has been used for the parallel simulation of a number of applications in diverse areas including queueing networks, VLSI designs, parallel programs, mobile wireless multimedia networks, ATM networks, and high-speed communication switches and protocols.

A PARSEC program is a collection of entity definitions and C functions. An entity definition describes a class of objects. An entity instance, called simply an entity, represents a specific object in the physical system. PARSEC defines a type called `ename` which is used to store entity-identifiers. Each entity knows its own ename with the `self` variable. Entities communicate with each other using buffered message passing. PARSEC uses typed messages; an entity definition must define the types of messages that may be received by its instances. An entity sends a message by executing a `send` statement. Each message is transparently time stamped with the current simulation time and is deposited in the destination buffer at the same (simulation) time at which it is sent. An entity accepts messages from its message-buffer by executing a PARSEC `receive` statement on a given message type. The simulation time of an entity can be advanced only when it receives a message or when it executes a `hold` statement. For the conservative synchronization, `add_source` and `add_dest` build the topology.

4.1 Architecture of a Conservative Federate

The parallel simulator built with PARSEC is a single-federate model. Callback notifications from the RTI are translated into special PARSEC events and sent to the relevant entities, time stamped by the simulation clock. Each entity can declare its interest in receiving a given set of notifications. At the moment, the target architecture is a SparcServer 1000 with shared-memory. Therefore each entity can access the RTI objects. Implementation on a distributed memory machine will be considered in the future. A time manager is defined to perform the GVT computation and the time advance grant request.

In PARSEC, the time manager is simply a special entity added to the source set of the other entities. In figure 4, each LP has one additional input channel from the time manager (solid-line arrows), in addition to those needed for the communication with the other LPs (dashed-line arrows). The task of the time manager is three-fold. First, to compute the federate's simulation time and send it to the RTI (by requesting a time advance grant with `flushQueueRequest`), second, to request the federate's LBTS from the RTI in order to enable the other entities to advance with sufficient parallelism and, third, to periodically tick the RTI runtime in order to allow the delivery of incoming messages. For the time manager to compute the federate's time, it is required that the other entities send to it the timestamp of the last event they executed (with a specially defined `TimeReport` PARSEC message). According to section 3.6, the LPs report their simulation time after the processing of events. To have a sufficiently accurate federate's time, this should probably be done every time an entity process an event, but it is still safe to report only the timestamp of the last processed event that must be smaller than the federate's LBTS.

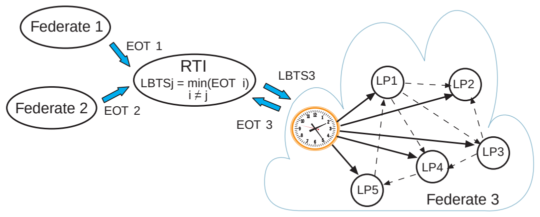

Figure 4: The time manager.

Actually, only entities that are likely to send/receive TSO messages to/from the outside, i.e. to/from the RTI, need to report their clock value to the time manager. The time manager and the other LPs work concurrently and therefore the LPs can process their eligible local events while the time manager gathers all the LPs' local simulation time and requests for the time advance grant. The tasks realized by the time manager and a user entity are shown below in a pseudo-PARSEC code (commands starting with an underscore represent calls to the RTI services):

```
message TimeReport { ename sender; double clockTime; }

entity TM() {
   double LPtime, Stime;
   ename  sender;

   receive (TimeReport theTimeReport) {
        LPtime = theTimeReport.clockTime;
        sender = theTimeReport.sender;
   }
   STime = ComputeFederateTime(LPTime, sender);
   _flushQueueRequest( (FederateTime)STime );
   _tick;
  receive (TimeAdvanceGrant theTimeAdvanceGrant) { }
   RequestFederateLBTS();
   SendLBTS();
}

entity myLP() {
   add_source(TMename);
   while (1) {
        receive (ModelMessage theModelMessage) {
             . . . process the message
        }
      send TimeReport { self, simclock() } to TMename;
   }
}
```

`ComputeFederateTime()` simply computes the federate's time after taking into account the newly received time report. `SendLBTS()` informs the user-defined LPs of the federate's LBTS. This can be done by sending a message on the channel connected to each user-entity. The implementation of the time manager is not completed yet. We are still optimizing how often the time report should be done and how often the LBTS should be sent to user-defined LPs. For the user-defined entity, as the entity's simulation time advances only on a `receive` statement, reporting after the execution of the event significantly simplifies the scheduling policy.

4.2 Example: The Wireless Network Model

HLA features have been added to the PARSEC language and an HLA-compatible "helloWorld" example has been successfully developed. We are now evaluating the performance of a parallel federate within an HLA federation. As a first attempt, we choose an application that has good lookahead, good time scale for the parallel federate and simple publication/subscription requirements. The test federation consists of 2 federates: a sequential federate and a parallel federate. The parallel federate models a wireless network of radio transmitters and the sequential federate models a weather condition. Speedu is measured by comparing the execution time of the federation execution when (i) the wireless network is simulated with a distributed simulation algorithm and (ii) the wireless network is executed sequentially.

Each radio transmitter is defined as a PARSEC entity. A time manager entity is defined as previously described. Messages exchanged between transmitter entities represent the arrival of a call. Calls are forwarded randomly from one transmitter to another before finally being accepted by a transmitter. Since radio transmission is used, the packet-loss rate depends on the weather condition, modeled by the sequential federate. This federate periodically publishes an indicator, called `weather_condition`, subscribed to by the wireless model federate. The weather condition changes every 10 minutes. This provides both good lookahead and time scale for the parallel federate. Both the weather model and wireless network model are intentionally simple and are not intended to accurately represent the real system they model.

5 CONCLUSION

In this paper, we addressed the problem of parallel and distributed time-constrained federate developments within the HLA framework. This development is not an easy task and several different design choices can be made: one or several federates, when and how to advance the federate's time, how to handle RTI notifications, etc. From our experiments, we think that it would be more convenient to give the possibility for a time-regulated and time-constrained federate to automatically receive events from the RTI without any queuing from the RTI. In the case of a conservative or optimistic federate, the underlying synchronization algorithm of the parallel federate would take care of the time order of the messages in a traditional PADS way (an additional IC for conservative federates and a rollback facility for optimistic federates). This study has helped clarify which choice may provide the best parallel execution. Future work will address the performance issue using the wireless model described previously.

ACKNOWLEDGMENTS

The work of C. D. Pham as a post-doctoral fellow at UCLA is supported by DGA/SREA/SC/EXP under contract 9381100095 and by the U.S. Department of Defense/ARPA DOMAINS project under Contract DAAB07-97-C-D321.

REFERENCES

Bagrodia, R., et al. PARSEC: A Parallel Simulation Environment for Complex Systems. *To appear in Computer Magazine*.

Chandy, K. M., and J. Misra. 1979. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *Trans. on Soft. Eng., Vol. 5(5)*, pp440-452.

Defense Modeling and Simulation Office. RTI Programmer's Guide. Version 1.0, version 1.3.

Defense Modeling and Simulation Office. HLA: Interface Specification. Version 1.3, version 1.2, version 1.1.

Fujimoto, R. M. 1995. Parallel And Distributed Simulation. In *Proceedings of the WSC'95*, pp118-125.

Fujimoto, R. M., and R. M. Weatherly. 1996. Time Management in the DoD High Level Architecture. In *Proceedings of the PADS'96*, pp60-67.

Jefferson, D. R. 1985. Virtual Time. *ACM Trans. on Prog. Lang. and Sys., Vol. 7(3)*, pp405-425.

Jha, V., and R. L. Bagrodia. 1994. A Unified Framework for Conservative and Optimistic Distributed Simulation. In *Proceedings of PADS'94*, pp12-19.

AUTHOR BIOGRAPHIES

C. D. PHAM is a post-doctoral fellow at UCLA. He received a Ph.D. in computer science from the University of Paris 6, France. His research interests focus on parallel discrete event simulation algorithms.

R. L. BAGRODIA is a Professor of Computer Science at UCLA. He obtained a Bachelor of Technology in Electrical Engineering from the Indian Institute of Technology, Bombay, in 1981. He obtained his M.A. and Ph.D. degrees in Computer Science from the University of Texas at Austin, in 1983 and 1987 respectively. His research interests include distributed algorithms, parallel languages, programming methodology and performance evaluation.