HLA Support in a Discrete Event Simulation Language

C. D. Pham
RHDAC
Université Claude Bernard Lyon I
FRANCE
cpham@lhpca.univ-lyon1.fr

R. Bagrodia
Computer Science Department
University of California, Los Angeles
USA
rajive@cs.ucla.edu

Abstract

There has been a growing interest in the HLA proposal as a common framework for real-time, distributed and interactive simulations. Although most of HLA compliant federates are for the moment developed with general purpose languages, the development of HLA federates with an existing simulation language would enable the rapid porting of a large number of existing simulations into the HLA framework. The difficulties of supporting the HLA features in a discrete event simulation language come from the different way the two approaches manage the simulation time and the interactions between objects in the simulation. This paper addresses these problems and uses the PARSEC *discrete event simulation language as a test case.*

1. Introduction

HLA (High Level Architecture) [1, 2] has been proposed as the common framework for all DoD simulation applications. The federated approach used in HLA calls for the interoperability of independent simulators within a federation. Each simulator is referred to as a federate and uses the services of a software component called the RTI (RunTime Infrastructure) [3]. HLA builds upon the results of the DIS experiments with the desire to incorporate advanced time management policies that were lacking in the DIS approach. As HLA is becoming more popular, many DIS simulations are now being converted into the HLA framework ([4, 5, 6] to name a few).

HLA compliant federates are for the moment mostly developed with general purpose languages. Coming from the discrete event simulation (DES) community, we found interesting to permit the development of HLA federates with an existing simulation language in order to benefit from its high-level features. Moreover, this would enable the rapid porting of a large number of existing simulations into the HLA framework. The difficulties of supporting the HLA features in a DES language come from the different way the two approaches manage the simulation time and the interactions between objects in the simulation. HLA is still mainly oriented towards real-time simulations with no or few logical time concerns whereas a discrete event simulator is mainly based on logical time and requires that every change in the system results from the processing of a time stamped event. However, the advanced time management features in HLA [7], that did not exist in the previous DIS, has been obviously proposed with the desire to merge the previously different two communities into the HLA framework, making HLA as general as possible.

This paper addresses these problems and uses the PARSEC [8] DES language as a test case. PARSEC is based on the C language and is designed to neatly separate the simulation model from the underlying simulation algorithm (sequential or parallel) that may be used to execute the model. It has been used for 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 `receive` statement on a given message type.

In order to introduce HLA features into the PARSEC environment, compatibility between PARSEC and HLA will be addressed following 3 main concerns : the API, the object management and the time management. We will say that a DES language is HLA-compatible if it can (i) use the RTI services and (ii) receive and process all types of notifications from the RTI while ensuring that such notifications are compatible with its internal rules. This paper is organized as follows: Section 2 describes the C wrapper to the C++ RTI interface (RunTime Infrastructure : the software kernel of the HLA [3]). Section 3 discusses the time management issues. Section 4 addresses the handling of the RTI notifications within the discrete-event simulation language. Section 5 explains how to incorporate calls to RTI functionalities within a PARSEC program and describes basic operations such as subscription and publication. For didactic purposes, a simple "helloWorld" PARSEC program will serve as an example throughout this section. Section 6 concludes.

2. Interfacing PARSEC with the RTI

A PARSEC program is translated into a C program with the PARSEC compiler. This C program will then be compiled with a C compiler and linked to the PARSEC runtime to produce the executable. In order to interface PARSEC with the RTI, a C wrapper to the C++ RTI interface is needed. The C wrapper for the current version of the RTI (v1.3r6) is based on a previous C wrapper for RTI 1.0 provided by Virtual Technology Corporation. The information given here describes the main characteristics of the C wrapper that would allow the user to make further translation from one interface to another for his own needs. C++-based discrete-event simulation languages can use the RTI interface directly.

2.1. The C wrapper and its main characteristics

The C wrapper is intended to provide an exact C interface to all methods defined in the C++ interface. There is no notion of classes or methods anymore in the C wrapper. All functions are provided as global functions and the class hierarchy is flattened. The naming convention consists in replacing the scope syntax *RTI::* by *RTI_*. For example, the *RTI::ObjectClassHandle* type becomes *RTI_ObjectClassHandle*. In general, each time a method *theMethod* of a class *theClass* is called, this is translated to a call to a C function *RTI_theClass_theMethod* with a pointer to a *theClass* object in the argument list. For example, the following statements:

```
RTI::Handle h ;
rtiAmb.createFederationExecution("helloWorld",
      "helloWorld.fed") ;
```

becomes

```
RTI_Exception ex ;
RTI_Handle h ;
RTI_RTIambassador_createFederationExecution(
   &ex, rtiAmb, "helloWord",
   "helloWorld.fed");
```

The `RTI_Exception ex;` statements and the `&ex` argument are used for exception handling. When several methods only differ from the argument list, a number is concatenated to the C function name to differentiate the functions in C. This number is incremented following the declaration order of the methods in the original C++ RTI declaration header file. For example, *RTI_RTIambassador_tick2(...)* replaces *RTIambassador::tick(minimum, maximum)* and *RTI_RTIambassador_tick()* replaces *RTIambassador::tick()*.

2.2. Callback mechanisms

In the C++ RTI interface, the RTI kernel communicates with the user federate by calling methods of a user defined class *myFederateAmbassador* derived from the *FederateAmbassador* class. In the C wrapper this callback mechanism is implemented differently : a callback structure *RTI_FederateAmbassadorCallbacks* is defined as an array of pointers to functions. An additional mechanism based on PARSEC messages is provided to translate callback calls into messages and will be explained later on. In this case, the user does not have to provide user-defined callback functions but needs only to take into account callback messages.

3. Time management

The time management is certainly one of the main concerns when interfacing HLA with a DES language. We will review below the basic characteristics of both approaches.

3.1. In DES

In DES, the changes in the system occur at discrete points in the simulation time with the model jumping from one state to another according to the progress of event processing. Events are taken from a Future Event List (FEL) and processed in timestamp order. New generated events that may be produced are then inserted in the FEL to be processed in the future.

3.2. In HLA

In HLA, the time-regulation and the time-constrained booleans define how a federate manages the time. A time-regulated federate is allowed to send time stamp order (TSO) messages. A time-constrained federate processes

TSO messages in timestamp order, as opposed to receive order. The RTI will deliver TSO messages for a time-constrained federate only when the federate's logical time has sufficiently advanced. If a federate is time-regulated, or time-constrained, or both, it has 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 indicate to the RTI that the federate is ready to receive TSO messages up to the requested time.

Within the traditional definition of DES, a PARSEC federate must be time-regulating and time-constrained. TSO messages for the federates in a federation are normally delivered conservatively by the RTI. Each time-regulated federate participates in a Lower Bound Time Stamp (LBTS) computation for the federation execution. This LBTS value is computed by taking into account the lookahead of all the time-regulating federates and represents "the greatest time-stamp such that it can be guaranteed that no time-stamp-ordered events will be subsequently generated in the federation with a lesser time-stamp". That does not imply that events with smaller timestamp are not currently queued for the federate. Federates can request "the minimum timestamp of all time-stamp-ordered events that may be subsequently delivered to the federate". For a time-constrained federate, it is not advisable to process local events with timestamp greater than the previous defined minimum timestamp.

3.3. HLA time management with PARSEC

Seven new PARSEC functions are introduced to manage the time according to the HLA standard. These functions are intended to simplify the time management to the end user. Six functions are related to the time-constrained and the time-regulation invocations. They are listed below:

```
void HLA_enableTimeRegulation(double theTime, double theLookahead);
void HLA_disableTimeRegulation();
void HLA_enableTimeConstrained();
void HLA_disableTimeConstrained();
RTI_Boolean HLA_isTimeConstrained();
RTI_Boolean HLA_isTimeRegulation();
```

They are provided to remove the need for the user to receive explicit notifications from the RTI. For instance, *RTI_RTIambassador_enableTimeRegulation(...)* is replaced by *HLA_enableTimeRegulation(...)* and *HLA_isTimeRegulation()* can be used to determine when the time-regulation is effective. Using these functions will automatically set the simulation time of the PARSEC federate to the correct time. Function for handling time-constrained facilities are similar.

The seventh function, i.e. *void HLA_timeAdvanceGrant_request(double t)*, is for time advance requests. Calling this function hangs the calling entity until the grant is accorded and allows the PARSEC runtime to correctly manage the federation time. There is no need for the user to explicitly receive a time advance grant because the call will simply return if the time grant succeeds.

4. Translation of RTI notifications

In the original RTI concept, notifications are implemented as callback functions. Each federate implements a set of predefined functions, one for each notification type, and passes them to the RTI via the *FederateAmbassador*. In the discrete event paradigm, every change in the system is the result of processing a time stamped event. Therefore, the callback calls from the RTI must be converted into events for the PARSEC federate. These events must have an associated timestamp that indicates the logical time at which they should be processed. HLA 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. Discrete event simulation usually deals with TSO messages only. The translation of RTI notifications into events for the PARSEC federate automatically adds a timestamp to every message, even those that were originally of the RO type. Since each entity in the PARSEC federate can be at a different simulation time, the notification events must be time stamped in a way to prevent any time causality violation. TSO notifications from the RTI carry a timestamp that is copied in the translated event.

Obviously, each entity can not provide its own set of callback functions to the RTI because (i) one federate allows only one callback structure and (ii) because it is not a good idea to let the end-user implement and provide callback functions in PARSEC. The solution we proposed is the following: predefined generic callback functions are implemented as a separate module. These functions simply encapsulate the arguments provided by the RTI into a special PARSEC event type and send it to the appropriate entities.

For example, the *turnUpdatesOnForObjectInstance* notification provides an object handle *theObject* and an attribute handle set *theAttributes*. In PARSEC, an entity would receive a message *m* of the *HLACallback* type with three relevant fields: *m.type* which would be set to *turnUpdatesOnForObjectInstance* and *m.theObjectHandle* and *m.theAttributeHandleSet*. A *HLACallback* message contains one field for each variable type that can be provided by all the RTI notifications. The naming convention for the message fields is to add the prefix *the* to the original variable type.

4.1. Registration of entities and callback function mask

In order to send the notifications to the entities, it is necessary to keep track of the entities which should receive the notifications. It would have been possible to automatically keep track of the entities based on the publication and subscription declarations, but this would require additional functions to encapsulate the basic functions provided by the RTI (e.g. *RTIambassador::subscribeObjectClassAttribute*). We chose the simple solution that consists in letting the entity specifying its own needs.

For example, some entities may be interested by the *RequestRetraction* notification but some others just do not need it. To filter the sending of messages, we have defined the notification type in such a way that binary masks can be created. A mask indicates the set of callback functions in which the entity is interested in. The registration of an entity is done with the *HLA_registerEntity* function that takes 2 arguments : an ename and a callback function mask. The callback mask may be changed dynamically at run-time. Generally, an entity would need to process several callback functions depending on what it has to do in the federate. If an entity does not need callback notifications at all, it can be registered with the *None* flag or not registered at all. If it needs to receive all callback notifications, it can be registered with the *All* flag. Between these two end-points, most entities would need to receive only a sub-set of all the callback notifications. For example, the most common sub-set would be the callback notifications for publication and subscription purpose.

In order to facilitate the registration procedure, notifications are grouped in small sub-sets, one for each identified category. For instance, notifications for declaration management are grouped in the following sub-set:

```
#define DECLARATION_MANAGEMENT_SET \
  startRegistrationForObjectClass|\
  stopRegistrationForObjectClass|\
  turnInteractionsOn|\
  turnInteractionsOff
```

There are 5 sub-sets for the federation management, the declaration management, the object management, the ownership management and the time management. Each sub-set is disjoint from the others. Practically, the only entities that should be registered are the ones that simulate some classes and attributes declared in the FED file. All other entities can be considered as internal to the PARSEC program.

It is not easy to keep track of what attributes or interactions an entity has subscribed to. Such features would introduce a high overhead. Therefore, every entity that has registered to receive a notification of a given type will receive all notifications of that type. Figure 1 shows a case where entities e_1 and e_2 have subscribed to attribute `a1` of class `c1` and attribute `a1` of class `c2` respectively. Both entities have registered them-self to receive `reflectAttributeValue` notifications. The appropriate callback function will send a PARSEC message with the relevant data to both entities. In the example, only e_1 should process the message by detecting that the provided arguments match the class and attribute handles it has subscribed to.

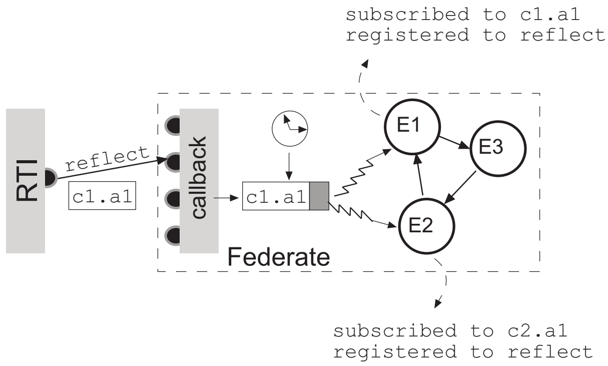

Figure 1. Translation of RTI notifications.

In a PARSEC federate, it is most likely that the entities have identical behaviors, e.g. each entity represents a node in a communication network. Since subscriptions in HLA are performed on a federate basis, the PARSEC federate will only receive one *reflectAttributeValue* per attribute value update. The multicast mechanism in PARSEC allows each entity to receive the *reflectAttributeValue* and to take independent actions. With the original callback mechanism, if a federate contains several identical objects which have subscribed to the same set of attributes, the same kind of multicast feature must be implemented otherwise the federate will only receive one *reflectAttributeValue* notification. Also, in the original callback mechanism, the complexity is put in the callback functions that usually have to perform several test statements to determine which action to take. For instance, for a *reflectAttributeValue* notification, the number of test statements is directly related to the number of different object classes the federate has subscribed to. In PARSEC the complexity is put in the entity. Since each entity knows exactly which classes it has subscribed to, the test is much simpler.

4.2. Special case for the time advance grant callback

We said previously that notifications from the RTI are translated into a PARSEC *HLACallback* message. One exception is the *TimeAdvanceGrant* that uses a special message type called *HLATimeAdvanceGrant* to handle this notification. The reason for this is because it must be possible to wait for a time advance grant event without pulling the other events from the event list. The *HLATimeAdvanceGrant* message contains the time (a double value) at which the time grant is accorded. Practically, the entities will not have to register to receive the time advance grant notification because the time advancement should be performed with the *HLA_timeAdvanceGrant_request* function described previously. Neither will the *HLATimeAdvanceGrant* message be broadcasted to all entities because the ename of the calling entity is used by the callback function to send the *HLA-TimeAdvanceGrant* message directly back to the entity.

4.3. The major difference introduced by callback messages

Even if the result of a callback message is the same as that of an original callback call, using PARSEC messages implicitly introduces time-constrained features. In RTI, a federate can be non time-constrained in which case all the notifications can be performed as soon as possible. In PARSEC, even if we declare the federate as a non time-constrained federate, it will be able to process a callback message only when the simulation time advances to the time of the callback message. This is because receive statements are used instead of direct callbacks. This difference is not very important since most of applications developed with PARSEC should be time-constrained.

Another difference with PARSEC is that the handling of callback functions can be done within the user PARSEC code of the entity. The advantage of this method is twofolds: (i) the callback mechanism is now consistent with the event-driven paradigm of PARSEC, and (ii) since the entity receives the callback notification directly, it can directly gain access to its local variables. In the original scheme, it was difficult to provide such an access to callback functions. For instance, if a federate contains more than one object type, then the original callback mechanism would require each callback function to determine to which object the notification should be applied. This is overwhelming even for a small number of object types.

5. Example

We provide in this section an example similar to the RTI "helloWorld" program but transcribed into the PARSEC language (we will called it *pcHello*). It differs from the original "helloWorld" by its event driven approach instead of a time-stepped one. Instead of periodically update the country's population by a time step of 1 year, *pcHello* uses a *Survey* event to trigger the update of the country's population every year. The 4 main phases of the program is (i) creation of the federation and of the entities, (ii) joining to the federation execution, (iii) publish and subscribe and (iv), simulation of the model and response to RTI notifications. Each of these main phases will be described below. We assume that the reader is familiar with the RTI interface and programming requirements.

5.1. Creation of the federation and of the entities

In PARSEC, the *driver* entity has a special purpose : it is the entry point of the program. Usually, this entity would create the other entities needed in the simulation. An additional entity called Country handles the simulation of a simplified country model. Like the original "helloWorld" federate, each country is defined by a name and an initial population. The creation of the federation execution is performed at line 7 in the *driver* entity. Note that the *RTIambassador* object is created using the *HLA_createAmbassador* function of the HLA PARSEC runtime so that all entities can have access to it (line 5). `TEST_EXCEPTION` (line 11) is a macro that simply tests for the exception value, prints the corresponding reason and can either exit or continue. Line 16 creates the Country entity with the PARSEC `new` statement.

```
1   entity driver(int argc, char **argv) {
2   RTI_RTIambassador* rtiAmb;
3   RTI_Exception ex;
4
5   rtiAmb = HLA_createAmbassador();
6
7   RTI_RTIambassador_createFederationExecution(
8      &ex, rtiAmb, "HelloWorld",
9      "helloWorld.fed");
10
11  TEST_EXCEPTION(ex.Type, "create federation",
12                 NO_EXIT_ON_EXCEPTION);
13
14  /* Creates the country with a name and
15     an initial population */
16  countryId = new Country(self,
17                          argv[1], argv[2]);
18  }
```

5.2. Joining a federation execution

We choose to join to the federation execution in the Country entity instead of in the *driver* entity because we wanted to keep the *driver* entity as simple as possible. Note that the first thing the Country entity has to do is to get the pointer to the *RTIambassador* object with the PARSEC *HLA_getAmbassador()* function (line 21).

```
19  entity Country() {
20
21  ms_rtiAmb = HLA_getAmbassador();
22  while (!Joined && (numTries++ < 20)) {
23    RTI_RTIambassador_joinFederationExecution(
24       &ex, ms_rtiAmb, "USA", "HelloWorld",
25       HLACallbackStruct);
26
27    Joined = RTI_Boolean_RTI_TRUE;
28
29    if (ex.Type ==
30        FederationExecutionDoesNotExist) {
```

```
31      printf("Federation does not exit\n");
32      Joined = RTI_Boolean_RTI_FALSE;
33      sleep(2);
34    }
35  }
```

RTI_RTIambassador_joinFederationExecution() needs a callback structure instead of a pointer to a *FederateAmbassador* object (line 23). PARSEC provides the predefined set of callback functions in *HLACallbackStruct*.

5.3. Time management declaration

Here, *pcHello* declares itself to be time-constrained and time-regulated. *HLA_isTimeConstrained()* and *HLA_isTimeRegulation()* are used to loop until the invocations take effect. *HLA_isTimeRegulation()* will also set the logical time of the entity to the time for which the regulation has been turn on. Parameters of *HLA_enableTimeRegulation()* are the same as those of *RTIambassador::enableTimeRegulation()* (here the lookahead value is 1.0, line 45).

```
36  ...
37  HLA_enableTimeConstrained();
38  /* Tick the RTI until we get the
39     timeConstrainedEnabled */
40  while (!HLA_isTimeConstrained()) {
41     RTI_RTIambassador_tick2(&ex, ms_rtiAmb,
42                             0.01, 1.0);
43  }
44
45  HLA_enableTimeRegulation(0.0, 1.0);
46  while (!HLA_isTimeRegulation()) {
47     RTI_RTIambassador_tick2(&ex, ms_rtiAmb,
48                             0.01, 1.0);
49  }
```

5.4. Subscribe, publish and registrations

The RTI uses run-time type identification to manage classes and attributes declared in the FED file. Since the user typically deals with string characters such as "Country" or "Name", the first thing to do is to get from the RTI the handles (typically integer or some enumeration types) assigned to these objects. Here we can almost use the direct translation from C++ functions to the C wrapper functions.

```
50  ...
51  m_countryTypeId =
52   RTI_RTIambassador_getObjectClassHandle(
54                   &ex, m_rtiAmb, "Country");
55  m_nameTypeId =
56   RTI_RTIambassador_getAttributeHandle(
57                   &ex, m_rtiAmb, "Name",
58                   m_countryTypeId);
59  m_popTypeId =
60   RTI_RTIambassador_getAttributeHandle(
61                  &ex, m_rtiAmb, "Population",
62                  m_countryTypeId);
```

To actually subscribe and publish we need to build an *AttributeHandleSet* that contains a list of attribute handles. The corresponding portion of code is shown below:

```
63  ...
64  {
65    RTI_AttributeHandleSet* countryAttributes;
66    countryAttributes=(RTI_AttributeHandleSet*)
67      RTI_AttributeHandleSetFactory_create(
68         &ex, 2);
69
70    RTI_AttributeHandleSet_add(&ex,
71       countryAttributes, m_nameTypeId);
72    RTI_AttributeHandleSet_add(&ex,
73       countryAttributes, m_popTypeId);
```

RTI_RTIambassador_subscribeObjectClassAttribute must be used for subscription. For publication, we use the *RTI_RTIambassador_publishObjectClass* function. The code for the subscription and publication is provided below:

```
74    ...
75    RTI_RTIambassador_subscribeObjectClassAttribute(
76       &ex, m_rtiAmb, m_countryTypeId,
77       countryAttributes);
78
79    RTI_RTIambassador_publishObjectClass(
80       &ex, m_rtiAmb, m_countryTypeId,
81       countryAttributes);
82
83    RTI_AttributeHandleSet_empty(&ex,
84                                countryAttributes);
85    RTI_AttributeHandleSet_delete(&ex,
86                                 countryAttributes);
87  }
88  /* same for interaction here */
89  ...
90  m_id = RTI_RTIambassador_registerObjectInstance2(
91           &ex, ms_rtiAmb, ms_countryTypeId);
92  HLA_registerEntity(self, PUBLISH_SUBSCRIBE_SET);
```

Note that it is necessary to free the memory used by the *RTI_AttributeHandleSet* when it is not needed anymore. This is performed by additional C functions *..._delete* that replace the C++ *delete* keyword. For subscription/publication to/of interactions, the processus is similar: (i) get the handles, (ii) call *RTI_RTIambassador_subscribeInteractionClass* and *RTI_RTIambassador_publishInteractionClass*.

RTI_RTIambassador_registerObjectInstance2 (line 90) is used for registering the object instance to the RTI since *RTI_RTIambassador_registerObjectInstance* requires an additional string argument that we do not want to use. Registration for receiving callback messages is performed at line 92.

5.5. Simulation of the model and response to callback messages

After publication/subscription and registration, the entity begins its simulation loop. Within the `while` statement, a `receive` statement on HLA callback messages is executed. The `receive` statement contains a `switch` statement that processes the message according to its type. The entity receives only the callback messages it has subscribed to (via the mask definition at entity registration). The code for message catching is shown below and we have intentionally given more details to illustrate how entities should handle RTI notifications. However, we assume that the reader understands and knows what should be usually performed upon reception of the notifications.

```
 93   while ( 1 ) {
 94   receive (HLACallback m) {
 95    switch (m.type) {
 96
 97     case discoverObjectInstance:
 98       if (m.theObjectClassHandle ==
 99            ms_countryTypeId) {
100          /* creates new structures,
101             saves the object handle */
102       }
103       break;
104
105     case reflectAttributeValues:
106       p = Country_Find(m.theObjectHandle);
107       if (p) Country_UpdateRemote(pCountry,
108          m.theAttributeHandleValuePairSet);
109       break;
110
111     case receiveInteraction:
112       if (m.theInteractionClassHandle ==
113            ms_commTypeId)
114          Country_UpdateInteraction(
115           m.theParameterHandleValuePairSet);
116       break;
117
118     case turnUpdatesOnForObjectInstance:
119       if (m.theObjectClassHandle ==
120            ms_countryTypeId)
121          Country_SetUpdateControl(
122           RTI_Boolean_RTI_TRUE,
123           m.theAttributeHandleSet);
124       break;
125
126     case turnUpdatesOffForObjectInstance:
127       /* same as above with RTI_FALSE */
128
129     case turnInteractionsOn:
130       if (m.theInteractionClassHandle ==
131            ms_commTypeId)
132          Country_SetInteractionControl(
133           RTI_Boolean_RTI_TRUE,
134           m.theInteractionClassHandle);
135       break;
136
137     case turnInteractionsOff:
138       /* same as above with RTI_FALSE */
139       }
140     } or timeout after(0) {}
141
142   /* Here we update the country state */
143   receive (Survey theSurvey) {
144     CountryUpdateTime();
145   }
146 }
```

As we said previously, having the entity receiving directly the notifications as PARSEC messages simplifies the management of several object types in the federate because the complexity is put in the entity rather than in the callback functions. Operations that are usually performed upon reception of a *reflectAttributeValues* notification (line 105) are to extract the updated attributes from the *theAttributeHandleValuePairSet* variable. Since direct translation from the C++ functions can be done, we will not explain further how to do it.

Callback messages should be check first, and then simulation model messages can be checked after. Note that a non-blocking receive statement is used for callback messages as indicated by the `timeout after(0)` construct. If the model has more message types then the PARSEC `receive (...) or receive (...) ...` construct can be used.

5.6. Federate termination

PARSEC provides a special statement for handling the termination of an entity: statements enclosed in the `finalize{ ... }` block are executed at the end of the entity life. There should not be `send` or `receive` statements. In our case, this is the best place to put the code for resigning from the federation execution, and to finally delete the *RTIambassador* object.

```
147 finalize {
148  RTI_RTIambassador_resignFederationExecution(
149     &ex, ms_rtiAmb,
150     DELETE_OBJECTS_AND_RELEASE_ATTRIBUTES);
151
152  RTI_RTIambassador_destroyFederationExecution(
153     &ex, ms_rtiAmb, "HelloWorld");
154
155  RTI_RTIambassador_delete(&ex, ms_rtiAmb);
156 }
157 } /* end of entity country*/
```

5.7. Time advancement request

Entities are required to ask for a time advance grant from the RTI in order to progress accordingly to the federation rules. This is performed by the PARSEC *HLA_timeAdvanceGrant_request()* function described previously.

```
158 void Country_UpdateTime() {
159  double federateTime;
160
161  federateTime = (double) simclock();
162  /* This would hang the entity until
163     the grant is accorded */
164  HLA_timeAdvanceGrant_request(federateTime);
165  deltaTime=federateTime-(double)m_lastTime;
166
167  /* Update the population, send the
```

```
168     attribute's updates,
169     send the interactions */
170   ...
171   send Survey {} to self after 1;
172 }
```

Once again, sending attribute updates and sending interactions can directly be derived from the original "helloWorld" program, we will not describe these steps further. Line 171 sends the next *Survey* message that would trigger a new state update according to the event-driven rules.

6. Conclusion

We addressed the problem of developing HLA federates with a discrete event simulation language, and in particular within the PARSEC language. Several mechanisms have been described to make this task achievable. In particular, the time management and the RTI notifications must be gracefully handled by the simulation language. We have proposed some solutions that can be used to incorporate HLA features in other similar simulation languages. As the demand for HLA-compliant simulators increases, enabling the development of HLA federates with an existing simulation language can encourage the porting of a large number of simulations into the HLA framework.

References

[1] Defense Modeling and Simulation Office, "HLA: Interface Specification", version 1.3, version 1.2, version 1.1.

[2] J. Dahmann, "High Level Architecture: Overview, Rules and Interface Specification", Slides.

[3] Defense Modeling and Simulation Office, "RTI Programmer's Guide", version 1.0, version 1.3.

[4] D. Coffin et al., "Developing a SOM for ModSAF", SIW'98 (Fall), *http://hla.dmso.mil/hla/papers/siw/siw4/98f-siw024.pdf*.

[5] P. T. Barham et al., "Implementing a Simple HLA FederateL Lessons Learned from a DIS Perscpective", SIW'98 (Spring), *http://hla.dmso.mil/hla/papers/siw/siw3/98s-siw120.pdf*.

[6] W. K. Braudaway, S. M. Harkrider,"Implementation of the High-Level Architecture into DIS-Legacy Simulations", *http://hla.dmso.mil/hla/implement/dis-hla_trans/hlalega.pdf*.

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

[8] R. Bagrodia et al., "PARSEC: A Parallel Simulation Environment for Complex Systems", *Computer Magazine*.