A new API to the HLA declaration and object management services for runtime modifications

Valéry Raulet
Alexis Nédélec
Vincent Rodin
Software Engineering Laboratory (LI^2)
École National d'Ingénieurs de Brest (ENIB)
Parvis Blaise Pascal
Brest, 29200, France
email: {raulet,nedelec,rodin}@enib.fr

Keywords:
High Level Architecture, Multi Agent System, Collaborative Prototyping,
Distributed Virtual Environment, Virtual Reality

ABSTRACT : This paper presents how the CERTI HLA API has been modified to include improvements needed for runtime dynamic applications. This means that some applications need to change data structuration while running to adapt to new behaviors as it was stated in a previous article[7].

Current HLA standard is turned away to military simulation which assumes – requires ? – exchanged information is already known before starting. We want to propose a new API subset in order to provide more flexible interactions with internal data for the declaration and object management services.

We details how this new API was added to a HLA implementation called CERTI.

1. Introduction

The HLA architecture is aimed at providing distributed simulations but is also particularly interesting for distributed virtual reality applications. Its opening to other areas, the way interactions are done between federates,its available services and its standardized API are promising. These are the main reasons why it was choosen for our collaborative prototyping platform.

But despite all these features, this architecture is not flexible enough for all usage – and will certainly never. Our problem affects the unability to change the object model while running the application. Our prototyping platform offers the ability to redesign a prototype while executing, avoiding to stop the application to make changes. Other platforms such as Bamboo[8] are expecting this runtine modification availability.

In this paper, we present the new HLA API features added to the CERTI RTI implementation. Only eight methods, four *RTIambassador* and four *Federate Ambassador*, were added to provide these on-the-fly modification capabilities. We first introduce our platform – oRisDis, a dynamic prototyping platform – and the CERTI RTI implementation.

2. The oRisDis platform

oRisDis, our distributed platform, consists of three main components. First, oRis is the core architecture which provides a multi agent language and a simulation engine for agents described in this language. The second component, ARéVi, is a 3D rendering API plugged around the oRis core. It provides many capabilities for rendering and external user interactions. The last one, oRisDis is our work in progress to provide distributed facilities to the oRis/ARéVi environment. Based on HLA, it provides data sharing. This data sharing is done the most transparently as possible to avoid bothering the designer with distribution problems.

2.1 oRis environment

oRis[1][4], the core component, is a Multi Agent System (MAS) developed in our laboratory. Built for many platforms (IA32 Linux, PPC Linux, SGI Irix and Windows), oRis provides an agent language and a simulation platform.

oRis provides an original prototyping tool. Indeed, instead of designing a prototype in phases between off-line and on-line design, oRis provides enough flexibility to fully modify prototype during simulation.

2.1.1 Prototyping

Prototyping consists in designing a model as expected. Classical prototyping is a round trip between off-line design and on-line tests. This way of acting tends to disappear in favor of complete numerical design. We want to go

[1] *http://www.enib.fr/~harrouet*

a step further by allowing its designer to modify its model while testing. Figure 1 shows interactive prototyping where adjustment are now done on-line.

2.1.2 An agent language

Based on a syntax similar to C++ and Java, oRis is a language very close to object oriented languages with few more features.

Features provided by this language are agent behavior, the `main()` module, agent communication capabilities, and dynamic behavior evolving.

In the `main()` module, we can describe the agent's behavior. At the simulation startup, this module is called and represent the agent's entry point. Then, this module is periodically called, boundlessly, by the simulation engine.

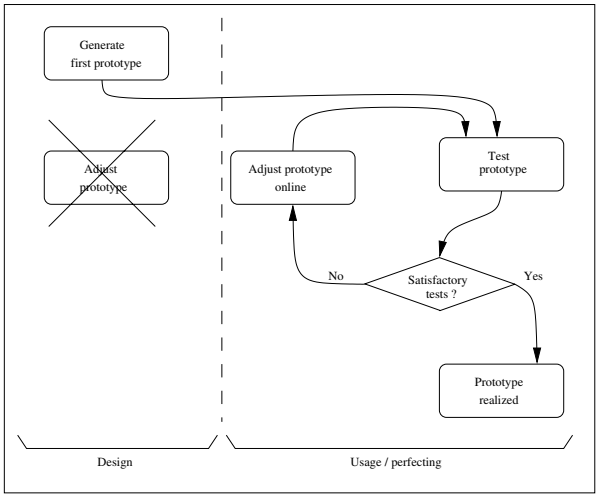

Figure 1. Interactive prototyping approach (from [4])

Agent communication capabilities allows agents to share data. This can be simple data such as entity state or more complex information described in ACL[2] language. ACL is a formalized way for communicating between agents. FIPA[3] or KQML[4] are some of these ACL[6].

Dynamic evolving is a specificity of our platform. Indeed, during execution, agents can evolve in a way that was unpredicted by its designer. The whole language is available at runtime and is interpreted. We call this a *dynamic language*.

[2] ACL : Agent Communication Language
[3] FIPA : Foundation for Intelligent Physical Agents
[4] KQML : Knowledge Query and Manipulation Language
[5] ARéVi, ***A**telier de **R**éalité **Vi**rtuelle*
[6] Office National d'Études et de Recherches Aérospatiales
[7] *http://www.cert.fr/CERTI/*

2.1.3 A simulation engine

Designed for simulating systems behaviors, the platform provides a simulation engine for making live agents. A fine tuned scheduler has been designed to avoid bias and thus each agent lives with fairness to its neighbor.

The aim of the engine is to simulate each agent behavior without privileging one of them and to provide complete access to the agent code and the oRis language. This allows an user to modify the application while running.

2.1.4 ARéVi toolkit

Based on oRis core component, ARéVi[5][5], our Virtual Reality Toolkit provides a 3D graphical environment. It allows entity representation in three dimensions and user interaction with adapted devices.

It can be used as a stand-alone tool for rapid prototyping or can be embedded into an existing system. Based on notions of entities, scene (group of entities) and viewers, it handles various features such as animations, level of details, lightning or collision detection.

Interactions are provided through usual or VR peripherals like 3D mice, force feedback joystick, flock of birds, Phantom and so on.

2.2 oRisDis Multiuser Platform

oRisDis is made of two parts: a plug-in C++ module for oRis and some oRis code. The module provides bindings between oRis language and the C++ HLA API.

The oRis code, called oRisRTI, is another RTI, designed specifically for our platform. Agents living onto the platform can interact with this oRisRTI. This latter is in charge of translating oRis agent calls into HLA calls. Conversely, it translates HLA calls into agent module calls. This is shown on figure 2.

3. CERTI RunTime-Infrastructure

The CERTI is a RTI prototype[2] originally developed at the ONERA[6] – a french governmental laboratory involved in aeronautic and spatial studies and researches – which provides security extensions[1]. This project started in 1996 and has been released as free software recently[7].

Currently, only a subset of the HLA Interface Specification IEEE 1516 is implemented (federation management,

synchronization, object management, some other services). But new capabilities are added each month.

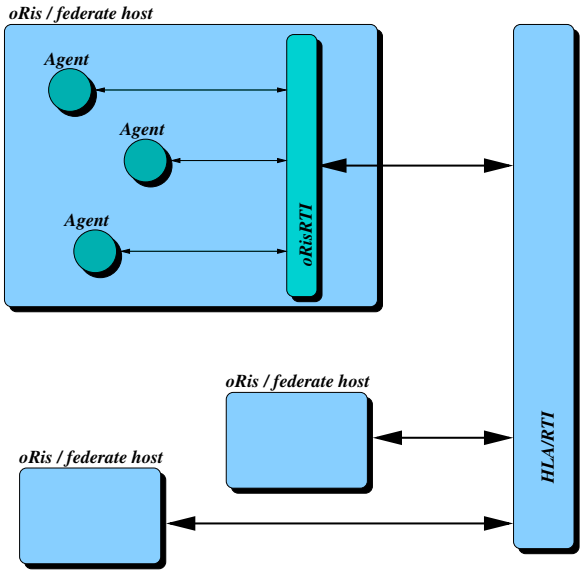

Figure 2. Double RTI infrastructure

3.1 Architecture

The CERTI architecture is build around three components communicating each others by socket :

- a RTI library linked with each federate,
- a local process for the RTI Ambassador (RTIA),
- a global process for data exchanges and RTI services with the RTI Gateway (RTIG).

The relations between each component is summarized on figure 3. They are detailed in the next three sections.

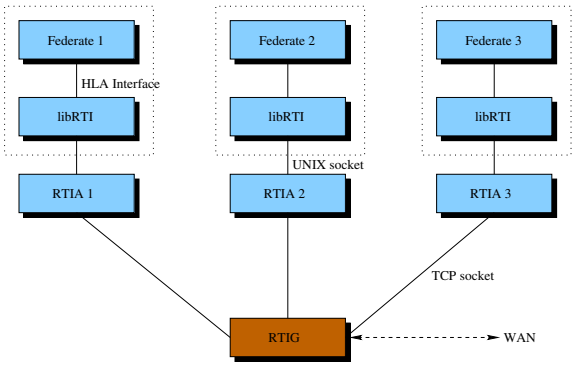

Figure 3. CERTI architecture (from [2])

[8] LRC : Local RTI Component
[9] *http://www.dmso.mil/public*

3.1.1 The libRTI library

This library is linked with the federate application. It is a small library which provides the HLA API. Each call to a *RTI ambassador* method is transformed into an UNIX socket call containing attributes information. It is sent to the RTIA and it then waits for an answer coming from the RTIA.

Calls to *federate ambassador* are received when the `tick` method is called. They are extracted from the UNIX socket call and transmitted to the application federate ambassador.

3.1.2 The RTIA

The RTIA – RTI Ambassador – receives data information coming from the libRTI through UNIX socket or from the RTIG through TCP (and/or UDP) socket. It acts as an intermediary between the federate and the RTI. It is comparable to the LRC[8] from the DMSO[9] (Defense Modeling Simulation Office) RTI.

Some requests from support services such as `Get Attribute Handle` or `Get Ordering Name` can be retrieved from the RTIA without being transmitted up to the RTIG. This allows the RTIA to answer quickly to request from its federate. This can be done since the RTIA contains an identical copy of its RTIG object model.

Other requests are sent to the RTIG and if accepted, some actions can be done locally.

3.1.3 The RTIG

The RTIG – RTI Gateway – is a central server which serves two purposes. Firstly, it permits data exchange between the differents federate through dialog between RTIG and RTIAs. Secondly, its centralized model allows a simple implementation of the RTI services.

All data exchange between federates are received by the RTIG. For example, a call to the service `Update Attribute Values` is received by the RTIG. Then the RTIG determines implied federates and send a call to the federate ambassador service `Reflect Attribute Values`.

3.1.4 Data-Transfer scenario

A typical data exchange is shown on figure 4. UAV is the acronym for `Update Attribute Values` and RAV

refers to `Reflect Attribute Values`. A message from federate is transmitted to its RTIA up to the RTIG. This message is processed and then forwarded to other RTIA up to the concerned federates.

4. Dynamic Object Model API

Simulations using the HLA API are developed by enterprises that must respond to strict specifications. Established before the development and certainly certified by the american US army, the object model is fixed.

We are developing a collaborative prototyping tool and as its name implies, the application needs refinements and improvements while designing. So does the object model. A fixed object model prevents us from doing online modifications.

A first version of our prototyping tool circumvent this limitation by encapsulating the instances into another API providing extensions. This is quicly explained in the first section. Our second implementation is using the CERTI implementation. This implementation has been modified accordingly to include new functionnalities allowing us to modify the object model while executing and is detailled in the remainder of this paper.

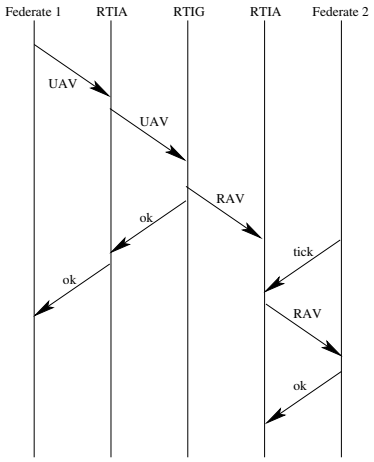

Figure 4. Data transfer scenario (from [2])

4.1 Encapsulating to provide a dynamic object model

A first solution to provide a dynamic object model to our application was to encapsulate RTI calls into another API which hides the HLA static object model[7]. The implementation was introducing a class into the object model to provide capability to extend the application object model. This class has only one attribute such as presented on figure 5.

If an `AirPlane` has only two parameters `Position`, `Orientation` and we want to extend this object model to be able to see light state. We can add a new attribute `Lights` to our application object model. Since HLA can't do so, it is hidden behing an instance from the `Dynamic` class.

```
(class objectRoot
  ...
  (class Dynamic
    (attribute Attribute reliable timestamp)
  )
)
```

Figure 5. Class for an external dynamic object model

A specialized protocol is used – by exchanging HLA interactions – which informs which instance from the `Dynamic` class is related to the application instance attribute. On figure 6, two instances from `AirPlane` are shown. `AirPlane` object model has been enriched by a new `Light` attribute and this is reflected on instances by a new `DynamicClass` which carries information on the HLA side (ellipses show added information).

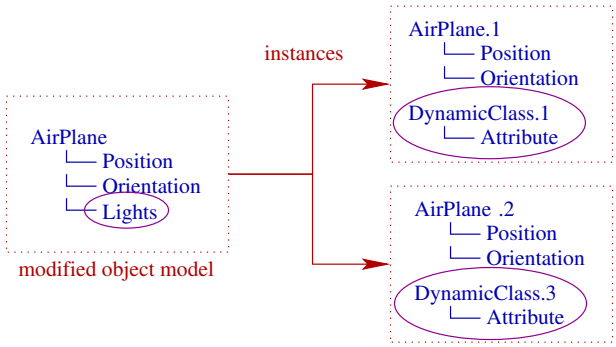

Figure 6. Dynamic Object Model

This solution was a simple way to avoid this difficulty. But not all services where usable with this solution. The adding of new methods to the HLA API was the only way to take full advantage from HLA.

4.2 Enrish the API for online modification

The object model is built when lauching the application. The RTI receives a file to read which informs it what kind of data (objects and interactions) are going to be exchanged during the execution. In the CERTI implementation, these information are used to build class instances (`ObjectClass`, `ObjectClassAttributes`, ...) which are containing associations (parent and son classes, attributes), subscribers, publishers and instances. These information are maintained by the RTIG and each RTIA.

Developping new RTIambassador and federate ambassador methods needs to be able to twist the RTI instances to insert new object classes and/or new attributes. We only developped a new API for object classes since we are not yet interested in adding new interactions. However, the implementation done for object classes doesn't differ for interactions since interactions are simpler than objects.

4.2.1 The Object Model

The object model is initially described in the FED[10] file. A FED file example is given below. Information described into this file are classes inheritance and class attributes.

```
;; (class <name>
;;  (attribute <name> <transportation>
               <ordering> [<space>])
;;  . . .
;; )
  (class entity
    (attribute location best_effort receive)
  )
```

Class attributes information are its name, transportation type, ordering and optionally associated space. Once set, inheritance and class attribute list (names) can't be changed.

Transportation is modified by the method `Change Attribute Transportation Type`. Ordering can be changed by the method `Change Attribute Order Type`. Last parameter associates an attribute to a routing space defined previously in the FED file. This is an optional feature that can be omitted if spaces are not used. This space can't be changed during execution but any regions can be created for needs. Currently, no services are provided for enabling (or disabling) data distribution management.

For interactions, the same process is developed. The `Change Interaction Transportation Type` is used for changing transportation, `Change Interaction Order Type` allows changes to the order. In the same manner, space is optional and cannot be changed during execution. Initially, all these information is set in the FED file :

```
;; (class <name> <transportation> <ordering>
          [<space>]
;;  (parameter <name>)
;;  . . .
;; )
  (class splat reliable timestamp
    (parameter target)
  )
```

Interactions differs from object classes because transportation, ordering and space are applied to the whole interaction class while object class applies these specifications to attributes (attribute granularity).

[10] FED : Federation Execution Data

4.2.2 The API methods

A first step to provide a dynamic object model is to allow adding of any element into the object model. Thus, new object or interactions classes can be added to the object model by setting its inheritance in the tree hierarchy. Also, new attributes or parameters have to be added to the object model. The appendix describes the RTI and federate ambassador methods added for providing these new services.

Object Class To provide a dynamic object model, we create only two methods for the RTIambassador and two other methods for the federate ambassador. All the implied operations are shown on figure 7.

The first method `Add Object Class` is used to add a new object class. Only information are its inheritance described by its first parameter and its name, the second parameter. The last parameter is used to transmit other data to other federates.

This call is received by the RTI and if the parent doesn't exists or if the object class name has already been used, an exception is thrown. Otherwise, the RTI adds this new object class to its object model and informs the other federates to do so. This result in a call to `Discover Object Class` which will contain the same information.

The federate receiving this call can process data locally to being able to treat this new added class. This is done by calling the usual method `Get Object Class Handle`.

Likewise, an attribute can be added to an object class by the call to the method `Add Attribute`. If the owner class handle is correct and the attribute has not yet been used, this attribute is added to the RTI. A call `Discover Attribute` to the federates reflects the new added attribute. Then, the federate can retrieve the attribute handle by calling `Get Attribute Handle`.

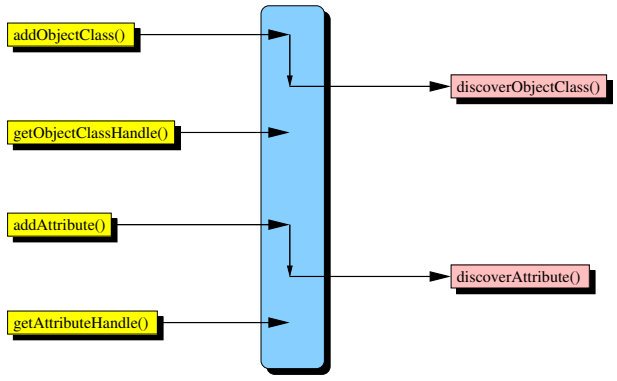

Figure 7. The dynamic object model API sequence

Interaction Class For interactions, the process is similar and is shown on figure 8.

`Add Interaction Class` is used to add a new interaction class in the object model tree. Inheritance is described by its parent handle parameter. This service is processed by the RTI and results in a `Discover Interaction Class` to the other federates. In the same manner, a new parameter can be added by calling `Add Parameter`. This results in a `Discover Parameter` method call.

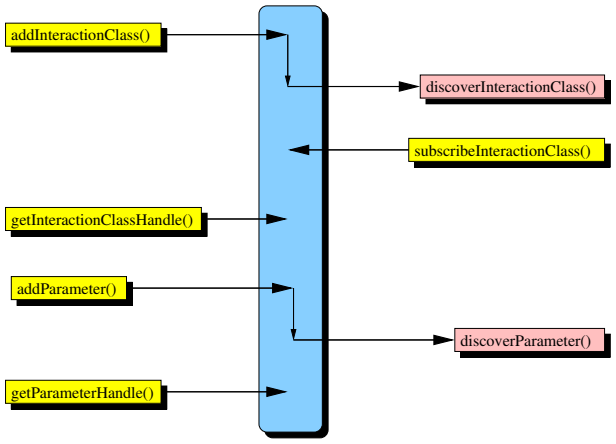

Figure 8. The dynamic interaction object model API sequence

Discovering Discovering of a new interaction or object class is done based on federate registration. `Discover Object Class` or `Discover Interaction Class` is sent to the federate only if it has subscribed to the parent class or if parent is `objectRoot` or `interactionRoot`.

For parameters, federates are informed of any new created parameters when it subscribes to the interaction by calling `Subscribe Interaction Class`. This mean that every object model modifications are marked to be sent to a federate if it subscribes to such a class.

For attributes, we cannot wait that federate subscribes to the object class. This way of acting is impossible since IEEE standard[3] says that any subscription to an object class (`Subscribe Object Class Attributes`) with an empty set of class attributes is equivalent to an `Unsubscribe Object Class`.

Two solutions are thinkable. First one is to modify the `Subscribe Object Class Attributes` to accept subscription if object class designator is a dynamic one but this is a special behavior that we do not retained.

The second one we used is to send a `Discover Attribute` to every federate that subscribed to the associated parent class.

4.2.3 Initial settings

When a new attribute or a new interaction is registered, its values are defaulted. Transportation is set to *reliable* and ordering to *receive*. This can be changed by calling corresponding methods. For associating a routing space to an attribute or an interaction, we added new fonctionnalities to the HLA API. This is described in the next section.

5. Routing space API extension

Previous methods doesn't allow using routing spaces since no such parameter is available. Indeed, we chose not to provide two methods : one without and one with data distribution management. Instead of this, we added further methods for modifying routing spaces associated with attributes or interactions. These methods are shown on figure 9.

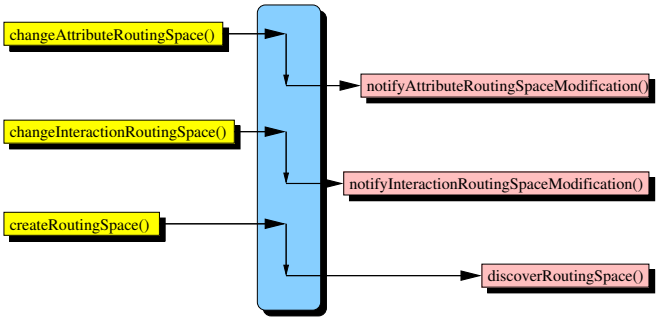

Figure 9. Routing space modifications during execution

`Change Attribute Routing Space` and `Change Interaction Routing Space` allows changing or setting the associated routing with a set of attributes and an interaction. We are conscious that modifying an existing used set of attributes cannot be done easily. Currently, these methods are intended only for setting the routing space with newly created attributes and interactions.

These two methods results in calls to `Notify Attribute Routing Space Modification` and `Notify Interaction Routing Space Modification`. These federate ambassador calls are needed for correctly setting `Register Object Class With Region` and `Send Interaction With Region`.

6. Conclusion

In this paper, we proposed a new API subset for managing evolving object model. These features are lacking in the HLA standard and may be needed by some networked applications – such as prototyping tools or long lasting adaptive environments.

We shown that implementing a dynamic object model is really a simple operation to do. Method behavior doesn't differ from usual ones, one call to `Add Object Class` results in n calls to `Discover Object Class`. The only trouble we encountered was keeping the object class handle counter since it was destroyed after the FED file was read.

Further control over the object model by modifying the routing space set becomes more tricky to handle. More in depth should be done in this part of the API.

This feature is essential for our application and we proved that some of these features could be added to the HLA API easily.

Acknowledgements

We would like to thank the CERTI team who released their API to the GPL license. This allows small teams as us to use complex API in research projects.

References

[1] P. BIEBER, J. CAZIN, P. SIRON, AND G. ZANON, *Security Extensions to ONERA HLA RTI Prototype*, in Fall Simulation Interoperability Workshop (98F-SIW-086), Orlando, USA, September 13–18 1998.

[2] B. BRÉHOLÉE AND P. SIRON, *CERTI: Evolutions of the ONERA RTI Prototype*, in Fall Simulation Interoperability Workshop (02F-SIW-018), Orlando, USA, September 8–13 2002.

[3] DEPARTMENT OF DEFENSE, *High Level Architecture Interface Specification – Version 1.3*, IEEE P1516.1, Standard for Modeling and Simulation (M&S), (1998).

[4] F. HARROUET, *oRis : in immersion through the language for virtual prototyping based on multi agents (in French)*, PhD thesis, Université de Bretagne Occidentale, Equipe d'accueil 2215, Laboratoire d'Informatique Industrielle, Décembre 2000.

[5] F. HARROUET, P. REIGNIER, AND J. TISSEAU, *Multiagent systems and virtual reality for interactive protot yping*, vol. 3, ISAS'99, Orlando (USA), July 31 - August 4 1999, pp. 50–57.

[6] A. NÉDÉLEC, P. REIGNIER, AND V. RODIN, *Collaborative Prototyping in Distributed Virtual Reality Using Agent Communication Language*, in IEEE SMC'2000, Nashville, USA, Octobre 2000.

[7] V. RAULET, V. RODIN, AND A. NÉDÉLEC, *oRisDis: using HLA and dynamic features of oRis multi-agent platform for cooperative prototyping in virtual environments*, in European Simulation Interoperability Workshop 2002 (02E-SIW-022), London, UK, June 2002, SISO.

[8] K. WATSEN AND M. ZYDA, *Bamboo - A Portable System for Dynamically Extensible, Real-time, Networked, Virtual Environments*, in Proceedings for the IEEE Virtual Reality Annual International Symposium (VRAIS'98), Atlanta, Georgia, USA, 14–18 Mars 1998, IEEE Computer Society Press, pp. 252–259.

Author Biographies

VALÉRY RAULET is a PhD student in Computer Science at the École Nationale d'ingénieurs de Brest (France). His work aims at providing a toolbox for building distributed and collaborative applications.

VINCENT RODIN is born on February 28 1966. Lecturer at the École Nationale d'Ingénieurs de Brest (France), he's currently working on image processing, virtual reality and computer simulation of biologic processes.

ALEXIS NÉDÉLEC is born on June 03 1961. Lecturer at the École Nationale d'Ingénieurs de Brest (France), he's currently working on Agent Communication Language (ACL) in Multi Agent System for collaborative applications development in virtual reality.

Appendix

RTIambassador API extension

```
class RTIambassador
{
...

// Object class modifications
void
addObjectClass(
  ObjectClassHandle theParent,
  const char *      theName,
  const char *      theTag)
throw (
  ObjectClassNotDefined,
  ObjectClassAlreadyDefined,
  FederateNotExecutionMember,
  ConcurrentAccessAttempted,
  RTIinternalError);

void
addAttribute(
  ObjectClassHandle theClass,
  const char *      theAttributeName,
  const char *      theTag)
throw (
  ObjectClassNotDefined,
  AttributeAlreadyDefined,
  FederateNotExecutionMember,
  ConcurrentAccessAttempted,
  RTIinternalError)

// Interaction class modifications
void
addInteractionClass(
  InteractionClassHandle theParent,
  const char *           theName,
  const char *           theTag)
throw (
  InteractionClassNotDefined,
  InteractionClassAlreadyDefined,
  FederateNotExecutionMember,
  ConcurrentAccessAttempted,
  RTIinternalError);

void
addParameter(
  InteractionClassHandle theClass,
  const char *           theParameterName,
  const char *           theTag)
throw (
  InteractionClassNotDefined,
  ParameterAlreadyDefined,
  FederateNotExecutionMember,
  ConcurrentAccessAttempted,
  RTIinternalError);

// Routing space modifications
```

```
void
changeAttributeRoutingSpace(
  ObjectHandle              theObject,
  const AttributeHandleSet& theAttributes,
  SpaceHandle               theSpace)
throw (
  ObjectNotKnown,
  AttributeNotDefined,
  InvalidSpaceHandle,
  RTIinternalError);
};

void
changeInteractionRoutingSpace(
  InteractionHandle theObject,
  SpaceHandle       theSpace)
throw (
  InteractionClassNotDefined,
  InteractionClassNotPublished,
  InvalidSpaceHandle,
  RTIinternalError);

void
createRoutingSpace(
  const char * theName,
  const char * dimensionsName[])
throw (
  SpaceAlreadyDefined,
  FederateNotExecutionMember,
  RTIinternalError);
```

Federate ambassador API extension

```
class FederateAmbassador
{
...

virtual void
discoverObjectClass(
  ObjectClassHandle theParent,
  const char *      theName,
  const char *      theTag) = 0;

virtual void
discoverAttribute(
  ObjectClassHandle theClass,
  const char *      theName,
  const char *      theTag) = 0;

virtual void
discoverInteractionClass(
  InteractionClassHandle theParent,
  const char *           theName,
  const char *           theTag) = 0;

virtual void
discoverParameter(
  InteractionClassHandle theClass,
  const char *           theName,
  const char *           theTag) = 0;
};

// Routing space modifications
virtual void
notifyAttributeRoutingSpaceModification(
  ObjectHandle              theObject,
  const AttributeHandleSet& theAttributes,
  SpaceHandle               theSpace) = 0;

virtual void
notifyInteractionRoutingSpaceModification(
  InteractionHandle theObject,
  SpaceHandle       theSpace) = 0;

virtual void
discoverRoutingSpace(
  const char * theName,
  const char * dimensionsName[]) = 0;
```