MASARYK UNIVERSITY
FACULTY OF INFORMATICS


# Integration of Java and C++ Federations in M&S HLA Simulations

BACHELOR THESIS

**Andrej Pančík**

Brno, spring 2010

# Declaration

Hereby I declare, that this thesis is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Andrej Pančík

**Advisor:** Ing. Petr Gotthard

# Acknowledgement

I would like to thank my advisor Petr Gotthard for patient guidance, wise advice and helpful criticism. Without him this thesis could not have been completed.

I would also like to thank CERTI developers team for great job. In particular I would like to thank Eric Noulard who kindly supported me all the time.

# Abstract

High Level Architecture constitutes a modern approach to distributed simulation of complex systems. In this bachelor thesis, we discuss extending CERTI, an open-source Run-Time Infrastructure, with binding to previously unsupported Java language. In addition, we investigate ways of simplifying the process of adding support for new languages by using automated code generation. We test the extension by modificating the OpenRADAR to use the Flight Gear simulator data while utilizing the Virtual Air middleware.

# Keywords

M&S, Modeling & Simulation, Distributed simulation, HLA, High Level Architecture, RTI, Run-Time Infrastructure, CERTI, Java.

# Contents

# List of Figures

# 1 Introduction

Computer simulation has become a natural part of the system design process. It is also commonly used to verificate and validate theories, to study the behavior of complex systems and to analyze possible outcomes of strategies. [31] defines *simulation* as "the process of designing a model of a real or imagined system and conducting experiments with that model."

This generally entails the representation of key characteristics and behaviors with their subsequent analysis and evaluation. Increasing the interest in simulation brings up questions related to interoperability and portability. Several standards are used to maintain these features in simulation applications.

In this thesis we focus on a specific part of computer simulation called distributed simulation and practical aspects of adopting it in production environment. We address the issue of extending the existing software platform CERTI, implementing High Level Architecture standard, with binding to previously unsupported Java language. In addition, we explore ways of simplifying the process of adding new language support by using automated code generation.

Next we describe foundations of modern simulation techniques. A brief introduction to High Level Architecture, its terminology and its history then follows in chapter 3. Chapter 4 addresses Run-Time Infrastructure and provides information about CERTI architecture. Chapter 5 discusses one of the practical outcomes of this thesis – Java LibRTI, including the reasoning behind Java binding and its creation and a brief description of architecture design. Chapter 6 in turn examines the message generator used for automated code generation. Results of the testing are presented in chapter 7 and our final thoughts and remarks are contained in chapter 8.

# 2 Distributed Simulation and its Technologies

Demands of the modern science and business are shifting from simulation of individual and isolated systems towards simulation of highly complex and/or parallel systems often not running in real-time. This shift is reflected also by architectures of simulation frameworks which focus on loosely coupled systems executing units of simulation.

One of the reasons for such trend is that "every time we wish to build a simulation to represent a complex activity, it makes sense to first build smaller simulations to represent individual entities and then to make these smaller simulations interact with each other to create the desired larger simulation while spreading the computational load. It also makes sense that if we build simulations at a later date, then these simulations can interact with other existing simulations as required." [24]

Assuring this kind of compatibility between simulations is another important aspect. Standards related to *interoperation*[1] between them emerged. To name a few: Aggregate Level Simulation Protocol [14], Distributed Interactive Simulation [26], High Level Architecture (covered in greater detail in chapter 3) and Test and Training Enabling Architecture [17].

To sum up, "*distributed simulation* is concerned with the execution of simulations on loosely coupled systems where interactions take much more time [...] and occur less often." [15] They are usually used to simulate system of systems which is highly elaborated and the disadvantages of this complexity are compensated with issuing standards allowing better interoperation.

---

1. M&S Interoperability as defined by US Department of Defense – "The ability of a model or simulation to provide services to and accept services from other models and simulations, and to use the services so exchanged to enable them to operate effectively together." [32]

# 3 High Level Architecture

Our work is focused on *High Level Architecture* (HLA) – one of the architectures commonly used nowadays. In this chapter we will discuss what HLA is, its structure and advantages. We will also define terminology used throughout the thesis and look at a brief history.

"The HLA is a software architecture for creating computer models or simulations out of component models or simulations. The HLA has been adopted by the United States Department of Defense (DoD) for use by all its modeling and simulation activities. The HLA is also increasingly finding civilian application" [19].

In other words, it is a general purpose architecture for distributed computer simulation systems. In addition, it provides a flexible framework for creating simulation and interfaces to live systems. It is also used to facilitate the interoperability of different models and units of simulations. Likewise, HLA has important role in reusability of the code implementing it.

While being mainly a software architecture in these days it has no reference implementation. [24] There are three main components that comprise HLA:

- Framework and Rules [1]
- Object Model Template (OMT) Specification [3]
- Federate Interface Specification [2]

The *Framework and Rules* is the collection of rules that must be obeyed by a HLA compliant simulation. Rules must be unchanged across all the simulation units as they address the abstract behavior and define the overall architecture. They include manners of the interaction, the design principles and responsibilities of components.

The *Object Model Template (OMT) Specification* describes the structure of the objects transferred between the units of simulation,

all interactions managed by the unit and visible outside the unit. [25]

The *Federate Interface Specification* addresses the interface by which the federate is connected to Run-Time Infrastructure (RTI). RTI is the data distribution mechanism in HLA simulation described in greater detail in chapter 4.

## 3.1 Terminology

To properly define the structure of HLA powered simulation it is important to mention the terminology commonly used in connection with HLA. This nomenclature is also being used throughout this thesis.

In HLA a *federate* is a single simulation – a basic unit, a component, of the result system. "A federate may take the form of an aircraft wing or missile or it may take the form of a complete squadron. The level of aggregation of federates is determined by the developer to meet the required need. A federate is also the unit of software reuse." [24]

There is no limitation on a purpose of the federates. They may be simulation models, data collectors, simulators, autonomous agents or just passive viewers. [30]

If we connect multiple federates via one RTI and use a common OMT the resulting compound is called the *federation*. A session in which a group of federates participate is then called *federation execution*.

During such execution *objects* and *interactions* are transferred between federates. Every object represents a collection of data fields called *attributes* which are used for communication. Analogically every interaction representing the events sent between simulation has data fields called *parameters*.

## 3.2 Evolution of High Level Architecture

HLA evolved from its predecessor Distributed Interactive Simulation (DIS – described in standards of IEEE 1278 family – "IEEE Standards for Modeling and Simulation: Distributed Interactive Simulation"). DIS had its roots in SIMNET (SIMulation NETworking – Defense Advanced Research Projects Agency funded project from year 1983 [21]) and it was designed to support loosely coupled training exercises. It is oriented to local area networks and has difficulties with scaling to WANs. [24]

HLA shares some of its characteristics while it advances several aspects of simulations. It supports enhanced data distribution and time management. Additionally it does allow faster-than-real-time simulations and event-stepped war-games. Systems interacting can include "simulations which simulate objects which are hierarchical aggregates of individual entities (platoons, companies, or battalions) all the way to high-fidelity engineering models which run much slower than real time and simulate individual subsystems with very high accuracy." [16] HLA is also meant to be more flexible and scalable than DIS by allowing developers to have their own interpretation of its components. However, unless all federates agree on a single interpretation and single FOM they are not able to interoperate even though they are HLA compliant.

First attempt to create a standard presenting this new approach was HLA 1.0 in 1996 issued by Defense Modeling and Simulation Office that started with initial definition drafts the year earlier. It was mainly targeted on defense applications but with a considerable potential in other fields. Two years later HLA 1.3 came, under supervision of US Department of Defense, with several improvements and clarification of some ambiguous parts. HLA 1.3 standard went under examination of Simulation Interoperability Standards Organization

and in 2004 Dynamic Link Compatible (DLC) HLA API [1] Standard for the HLA Interface Specification Version 1.3 was born [28]. Additional description of DLC API can be found in section 4.1.

IEEE started to recognize HLA and founded a commission for preparation of international open HLA standard. The standard was developed using IEEE processes with strong focus on verifiability of compliance. It was targeted on both defense and non-defense application. The effort culminated in September of 2000 when IEEE issued 1516 standard Standard for Modelling and Simulation High Level Architecture with several components namely Framework and Rules, Federate Interface Specification, Object Model Template [1] Specification and Guidelines for Federation Architecture and Design. This standard was later revised by SISO as well [27].

"The [IEEE 1516.1] standard did not define the levels of concurrency and reentrancy that an RTI shall support. This led to differences in RTI implementations and limits federate portability." [29] That was one of the reasons why evolution continued and IEEE has on March 25, 2010 approved IEEE 1516-2010 series of standards with many bug fixes and new features. HLA 1516-2010 revision is referred to as "HLA Evolved" and it provides for example, proper fault tolerance support and fault handling and signaling. It contains encoding helpers unifying data manipulation which caused a lot of problems in previous HLA incarnations. [20]

HLA, in both IEEE 1516 and 1.3 version, is the subject of the NATO standardization agreement (STANAG 4603) for modeling and simulation: Modeling And Simulation Architecture Standards For Technical Interoperability: High Level Architecture. [12]

1. API – Application Programming Interface

# 4 Run-Time Infrastructure

HLA itself is a high level standard. It is focused on describing the system from the point of an architect which is different than that of a software engineer. It does not describe the actual implementation details but more the abstract model of the simulation. There is no network protocol specification, no message format and no encoding details.

Software part of HLA is known as the *Run-Time Infrastructure*. It is a middleware that supports the HLA simulation with necessary services and provides essential building ground for the software developers. Currently there are several RTIs available both on commercial and non-commercial basis. The brief overview of actively developed RTIs can be found in table 4.1.

Modern RTIs tend to conform to IEEE 1516 [1] and/or HLA 1.3 [28] interface specifications. However, it is always up to the developer to specify implementation details. This loose definition is the reason why the interoperability between different RTIs from different vendors is not guaranteed.

Most of the RTIs use a centralized structure which helps to facilitate time management and data distribution services. While *Central RTI Component* overseers the simulation and manages requests regarding communication of components, *Local RTI Component* exists in one instance for each federate and provides federate-specific functions and services. Deviations from this model are plausible in simulations where services bounded to centralized architecture are not necessary.

Table 4.1: The list of RTIs

| Name | Standard/Target | Bindings | License |
|---|---|---|---|
| CAE RTI | 1.3, IEEE 1516 | C++ | Commercial |
| Chronos RTI | IEEE 1516 | C++, .NET | Commercial |
| MÄK HP RTI | 1.3, IEEE 1516 | C++, Java | Commercial |
| S2Focus (HLA Direct) | 1.3 | C++ | Commercial |
| Openskies RTI | 1.3, IEEE 1516 | C++ | Commercial |
| Pitch pRTI | 1.3, IEEE 1516 | C++, Java | Commercial |
| Mitsubishi ERTI | 1.3 | C++ | Commercial |
| RTI NG Pro | 1.3, IEEE 1516 | C++, Java | Commercial |
| CERTI | 1.3, IEEE 1516 | C++, Matlab, Fortran 90, Python, Java | GPL, LGPL |
| EODiSP HLA | WIP IEEE 1516 | Java | GPL |
| GERTICO | 1.3 | C++ | |
| Portico | 1.3, IEEE 1516 | Java, C++ | CDDL |
| Open HLA | 1.3, IEEE 1516 | Java | Apache License |
| RTI-S | 1.3 | C++, Java | US Government |
| Rendezvous RTI | 1.3 | C++, Java | NUST |

## 4.1 Dynamic Link Compatible API

The lack of common ground in the RTI implementations caused any potential migration of an application to another RTI to be difficult. APIs were not fully specified and switching RTI vendors also meant recompiling and relinking the code. To address the problem SISO has developed in year 2004 a complementary HLA API specification called Dynamic Link Compatible (DLC) API (1.3 version is available at [28] whereas IEEE 1516 at [27]).

DLC defines several constraints on both RTI and the federate in order to guarantee the possibility of switching RTI without recompiling the whole application. Today, both major interface specifications exist in DLC flavor and thanks to them the developers can build application independent on specific RTI.

SISO DLC partially addressed the issue with encoding. "The encoding/decoding of data for consistency with the IEEE Std 1516.1 specification requires significant programming by developers." [29] That was the reason of including encoding helpers [1] right into the API. Unfortunately, they were originally included only in Java versions and none were provided for C++ API.

## 4.2 CERTI

Our thesis targets CERTI which is an open-source RTI distributed under GNU General Public License 2 with libraries licensed with GNU Lesser General Public License to allow use in proprietary applications [23]. Development started at ONERA Laboratories in 1996 and the first version was available by the end of year 1997. It is under active development since and it has attracted an active community with the

1. Encoding helpers – functions dedicated to lessen programming burden on federate developer

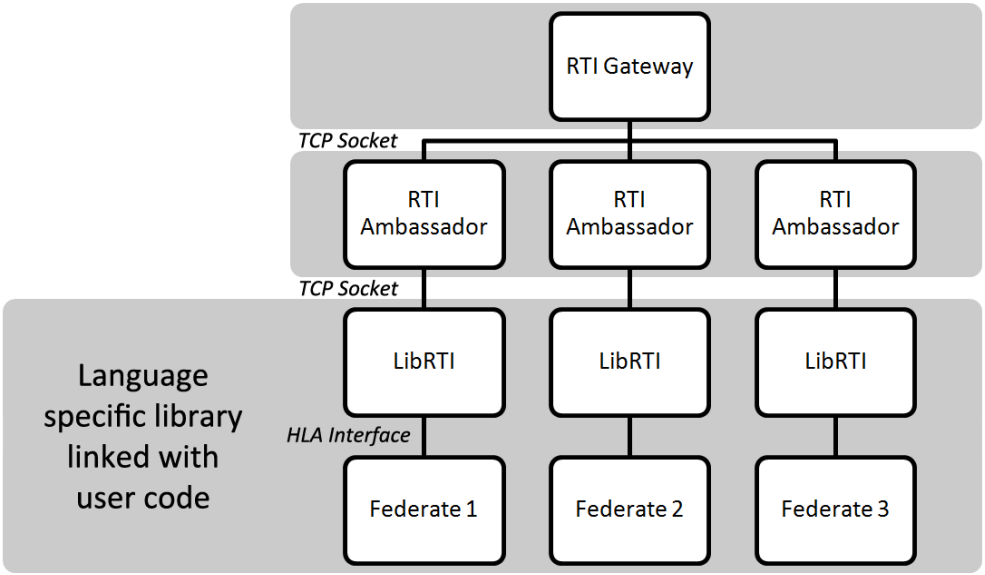


Figure 4.1: CERTI architecture

transformation to open-source development model in year 2002 [5].

CERTI is partly compliant with both HLA 1.3 and HLA 1516 and currently supports five language bindings specifically C++, Matlab, Fortran 90, Python and Java [5]. The support of the last one is the part of this thesis content. Supported platforms contains various flavors of Linux, Microsoft Windows, Solaris, FreeBSD and IRIX.

CERTI is designed in a very modular way. It has centralized structure and follows the client-server architecture as seen on figure 4.1. As a result, simulation using CERTI is scalable and can be easily distributed. All the main components communicate by sending messages through sockets on TCP network.

*LibRTI* is a library linked with each federate using DLC interface (for description of DLC see section 4.1). Purpose of LibRTI is to transform HLA service calls into messages sent to *RTI Ambassador* (RTIA) and receive responses in form of callbacks. Obviously the LibRTI

needs to contain all the necessary implementation of data structures and logic for sending and parsing messages.

Each federate connects to an RTIA. It is a process which exists in one instance for each federate. Its role is to satisfy some requests immediately, while forwarding some requests to *RTI Gateway* (RTIG) [13]. RTIA as Local RTI Component has important role in facilitating federate specific services.

Last main component of CERTI architecture is RTIG. RTIG is responsible for administering the simulation and routing messages between federates. There is only one instance of RTIG in simulation and it is also the central point. This simplifies the implementation of some HLA services such as the creation and destruction of federation execution or maintaining distribution of object classes. Single RTIG may handle several federations but the federation itself must be linked to single RTI.

# 5 Java LibRTI

In this chapter we address the necessity of extending CERTI with new language bindings and analyze the process of their creation. Moreover, we propose an architecture of such binding and demonstrate it on concrete implementation of Java LibRTI nicknamed jCERTI.

To utilize the CERTI framework in the federate one needs to use LibRTI. Implementing it in non-supported language is a non-trivial task. Considering large amount of HLA services or messages sent between LibRTI and RTIA one must take a fair amount of effort.

However, it is very important to support multiple languages as it gives freedom to developers of federates. Every language has some advantages and disadvantages and the final choice must consider them with the purpose of the federate in the scope. What is more, the code on which the federate is based is very often already existent. Developer has to use existing libraries, portions of code or sometimes just architectures and concept that are not present in all languages.

All in all, there is a strong motivation to implement LibRTI in multiple languages. CERTI is programmed mostly in C++ and it has existing bindings to Matlab, Fortran 90 and Python. We decided to explore the possible ways of simplifying the process of adding new language support and to add a new binding to Java.

## 5.1 Analysis of the Problem

An important part of the new language support development is an analysis of possible routes. In this case there are two possible paradigms that had to be analyzed in order to fully evaluate their outcomes.

One obvious option is to implement a simple wrapper around existing C++ version of LibRTI. This can be done with some automa-

tion using specialized tools i.e. SWIG [10]. This approach has several advantages. There is variety of supported target languages out of the box after some initial work on specification. On the other hand, wrapping has reported some performance issues in several languages. Java, for example, uses Java Native Interface for calling native code which has significant slowdown factor when used in all major virtual machines [22]. Other than that, generated code does not comply to any specific DLC without some additional work.

Another way to approach the support of a new language is to recreate a part of the CERTI from the scratch. There are obviously no setbacks related to performance as it runs as fast as possible when properly programmed. In addition, the resulting code is clean and it respects the coding style associated with the language. However, other problems arise from decentralization of the code. There is a unique set of bugs for each new implementation, difficult distribution of changes in network protocols, etc.

During the development process the pros and cons of each of the paradigms were carefully reviewed and we decided to follow the latter one. The challenge was to compensate drawbacks of this approach. We explored the possible ways to cope with this issue and we describe our solution, the message generator, in chapter 6.

Additionally, it was important to decide what needs to be reprogrammed in order to achieve our goals. Clearly implementing LibRTI was essential. In contrast, new RTIG programmed in Java would be a redundant effort with no significant positive outcomes. The question was: should there be a new RTIA programmed in Java or should we use the existing one?

Communication between LibRTI and RTIA process was originally done with UNIX socket. From the Java point of view, there is no native way to support this type of sockets so it would be better to include RTIA in Java LibRTI. However, UNIX sockets are not supported on

all platforms and they had to be changed to TCP sockets in order to support Microsoft Windows anyway. After transforming the communication to TCP sockets the answer was simple: use the existing C++ version of RTIA process as reprogramming it from the scratch would not bring any advantages.

## 5.2 Java SISO DLC 1.3

Another very important objective that was essential to keep in mind while implementing jCERTI was compliance with standard. One of the request for Java LibRTI was compliance with SISO DLC version 1.3 [28].

It consists of interfaces describing the data structures and RTI ambassador specification. As a result, Java application using DLC does not have to be recompiled when switching RTI. The only thing developer has to do is to change the Java classpath to correspond preferred RTI.

During the implementation this behavior was highly valued and breaking it was not an option since it was the main reason of choosing the use of DLC. Therefore, our effort was to keep the binary compatibility with other RTIs as much as possible.

First step when implementing the DLC 1.3 was to program the data structures. Data structures are used in communication between ambassadors. They are almost always present when executing callbacks or calling RTIA functions. Our implementation is stored in package *certi.rti.impl* as suggested in DLC guidelines [28].

Second step was implementation of RTI ambassador interface which standardizes the function names corresponding to HLA services. Our implementation of RTI Ambassador in LibRTI builds the message based on requested HLA service and sends it over the socket to linked RTIA process.

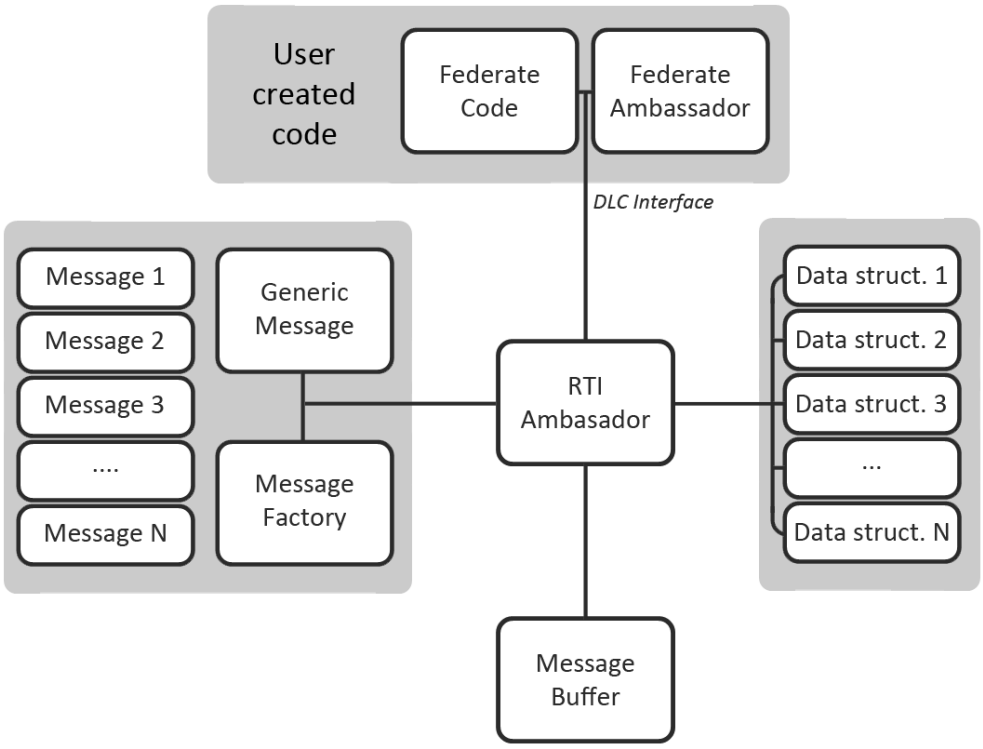


Figure 5.1: jCERTI architecture

Implementing DLC 1.3 interface specification brought up several issues to the development of jCERTI. For example, in Java and C++ DLC APIs handles are represented by types with different ranges. This fact itself theoretically hinders compatibility between C++ and Java code strictly implementing DLC 1.3 interfaces. However, the problem would become visible only with extremely high numbers used as handles and rational handle allocation on the CERTI side effectively prevents this type of complications.

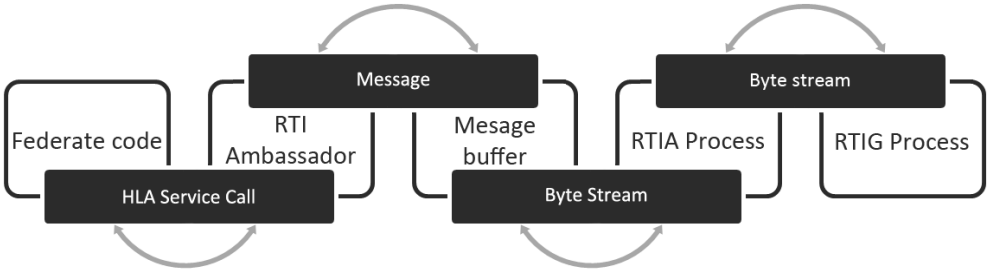


Figure 5.2: HLA Service Call lifecycle in jCERTI

## 5.3 Architecture

Our Java LibRTI was designed in a modular way to support maintainability and allow generated code to be easily deployed. It was based on some preliminary work done in 2007 by Yannick Bisiaux, Ronan Bossard and Samuel Reese. Overall architecture is shown as diagram in figure 5.1.

Federate code communicates with *RTI Ambassador* which exports available functions via HLA DLC 1.3 interface. RTI Ambassador processes the requests and communicates with other parts of CERTI architecture. The process of distribution of these requests and can be observed on figure 5.2. To put it in the nutshell parameters on DLC methods are converted into messages and then serialized on buffer into the byte stream. After that stream is sent to RTIA process through the TCP socket. At that point when the response to the service call is received, it is then parsed and distributed back to federate code.

Over time, RTI Ambassador receives callbacks from RTIA process and forwards them to *federate ambassador*. Federate ambassador is a part of federate code and it is mainly responsible for processing callbacks with data.

### 5.3.1 RTI Ambassador

*RTI Ambassador* is the main part of the jCERTI. Most of the methods in it are specified by interface whose implementation is required by DLC 1.3. As previously mentioned RTI ambassador is responsible for receiving HLA Service Calls and for subsequent transformation of the service calls parameters into messages. Additionally it translates callbacks during time advancements and communicates directly with federate ambassador.

Linking between standard-defined service calls and CERTI specified messages is based on semantics. Therefore, RTI ambassador code can not be automatically generated without explicit linking data.

### 5.3.2 Messages

*Message classes* are responsible for proper serialization and deserialization of data to/from buffer. Every message represents logical record of information transferred between RTIA and LibRTI. In each instance of message class there is holds data fields relevant to the specific message.

In jCERTI there is about 150 messages. All message classes code is generated from specification file. More detailed description of generator can be found in chapter 6.

### 5.3.3 Message Buffer

*Message buffer* is the class that is used to build the message from the ground up on the low level and send it over the socket. In addition, it facilitates parsing messages with different endianess [1] from the socket. Basically it is the lowest level of abstraction over the actual sockets and it transforms the complex structures to the series of bytes and vice versa. Message buffer is mainly used by the serialization and

1. Endianess – ordering of bytes in word

deserialization code inside message classes, but it can also be used *ad hoc* by federate code to serialize RTI related structures into series of bytes, if necessary.

### 5.3.4 Data Structures

In core DLC data structures resemble the standard Java collections with very few specific functions. Unfortunately, by the time the DLC was proposed Java did not contain the generics or any other more sophisticated ways to treat the data. That is the reason why DLC uses its own way which can be nowadays considered somewhat clumsy.

Internal data structures transferred over sockets in CERTI were mapped to the DLC structures. This was done by a very transparent way by overriding the *write* method in Message buffer. This way the DLC structure is distributed from the federate through the ambassador to the message serialization code and finally to message buffer and vice versa after receiving the message as seen on figure 5.2. Message buffer is always responsible for proper serialization and deserialization of the structure.

### 5.3.5 Logging

During development process we decided to use at least some level of logging to allow users debug their federates and check the RTI side of the simulation. We decided to use Java Logging API through the jCERTI code. Debugging aspect of logging is very important considering multi-layered complex system responsible for simulation. Without logging incorporated into LibRTI there would be little or no control over RTI part of simulation.

It is also beneficial to be able to set level of logging. Importance of such customization becomes apparent in production environment. To allow such configuration we introduced property file containing,

among others, verbosity setting. On the most verbose mode logging can serve as message log.

### 5.3.6 Technical Notes

To maintain compatibility with C++ code jCERTI had to close every connection with closing message. The closing message was supposed to end communication between LibRTI and RTIA process and it had no corresponding HLA Service Call. In spite of clear and rational semantics this requested behavior turned to be a challenge. Main reason was the fact that Java language does not have destructor concept as some other languages e.g. C++.

The solution we have chosen was the use of *Java Virtual Machine hook*. VM hook allows attaching sleeping thread to the virtual machine and start the execution on the time VM termination. This way we were able to build the thread acting as destructor, sending messages and closing connections.

# 6 Messages Generator

The number of messages transferred between the LibRTI and RTIA process is as high as 144. However, the messages are very similar to each other and they share the common base. They can be described by much simpler rules and a very little amount of data such as the name and the type of a field transferred.

That is the main idea behind the generator: one writes the message specification and let the script do the work and generate the actual code. This approach has one major advantage. To support a new language it is sufficient to just write a simple generator and use the common message specification. Moreover, the maintenance of the code is centralized and there is a proper distribution of changes in message specifications so one does not have to change each language separately.

We use the generator to generate the code of all messages in jCERTI. Flexibility of this paradigm was put to test during the development process when structure of the messages had to be changed to reflect changes in C++ code. After the release of CERTI 3.3.3 overhaul of RTIA and LibRTI communication was commenced. The target was all the communication between CERTI components to be generated from specification files. jCERTI was at first developed for version 3.3.3 and only later evolved into support of RTIA process with generated code.

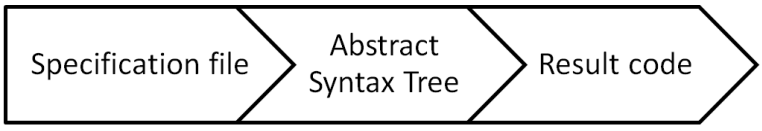


Figure 6.1: Generator principle

## 6.1 Generator Architecture

Overall idea of generator is inspired by the Google Protocol Buffers which are "Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data." [6] On the other hand, our generator is lightweight and suits our requirements well. Google Protocol Buffers takes specification file as input and compiles it into requested language. Java, C++ or Python developers can use this mechanism for serializing data in similar fashion as XML but without additional overhead caused by markup.

The base and C++ message file generator were written by Eric Noulard in Python [5] to maintain C++ code generation. They use the Python PLY module [8] to generate abstract syntax tree [1](AST) from the specification file, the form of which can be seen in a listing 6.1.

Specification file uses simple format to describe messages as small logical record of information. Every message contains a series of type-name pairs with additional modifiers. Type of data determines the way how the data are transferred over the socket in particular message. Order of the fields transferred is the same as in the specification file..

Listing 6.1: Sample specification file snippet

```
message M_Person {
  required string objectName
  required bool status
  optional string email
}
```

*Required* keyword precedes the field that is always present in the message. It notes basic atomic component of the message.

In contrast, *optional* keyword notes that the field is streamed first

1. "The structure of an AST is basically a simplification of the underlying grammar of the programming language, e.g., by generalization or by suppressing chain rules." [18]

as a boolean value, representing whether the field is present. If the the boolean value is *true* then the actual field follows.

Third interesting modifier, which is not present in snippet, is *repeated*. It represents an array of values preceded with integer – the size of the array.

Last keyword *combined*, also not included in sample, expresses the structure of the message and groups contained fields into one semantic whole. For example by utilizing combined keyword we are able to transfer handle/value pairs together. In combination with previous modifier whole collections of data can be transferred.

# 7 Testing

Testing provides basic level of quality assurance and protects codebase from major regressions. Importance of testing rises with complexity of system where manual checks usually fail. From the beginning of the jCERTI development process testing was considered to be a very important part. Thanks to our testing some bugs were found in both jCERTI and CERTI itself. Besides, certain level of testing was important to check functionality of generator.

## 7.1 Sample Federates

During the first phases of development we started to test jCERTI by using simple test federates. They were specifically designed to send and receive data to test and demonstrate data distribution services. After initial testing we decided to build sample suite to be bundled with jCERTI on release. The suite was also meant to provide example federates for interested developers.

Sample suite, namely two federates *uav-send* and *uav-receive*, is inspired with similar federates by Petr Gotthard included in PyHLA repository [9]. Complexity of the federates is reduced on purpose and code is simple and well commented. That makes them suitable for introduction to federate programming in Java.

## 7.2 Testing with OpenRADAR

To test the functionality of jCERTI in more advanced way we decided to modify the OpenRADAR, an open-source application using standard Air Traffic Control (ATC) [1] symbolics, to use the Flight Gear

1. ATC – service provided by ground-based controllers who direct aircraft on the ground and in the air

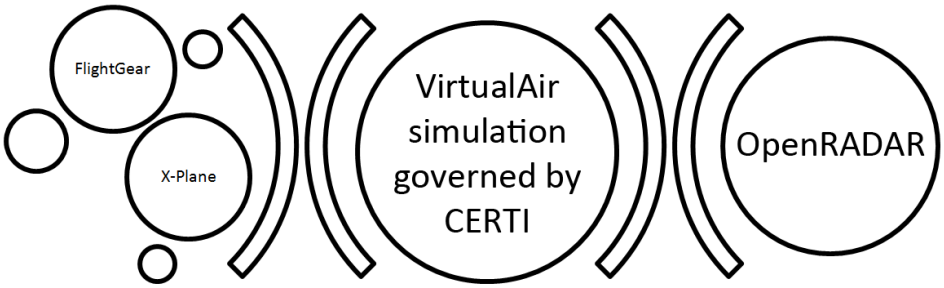


Figure 7.1: Virtual Air simulation

data while utilizing the VirtualAir middleware. Simply put the OpenRADAR is virtual radar application used to visualize the situation on Flight Gear Multiplayer Server (FGMS). Diagram of this simulation can be seen in figure 7.2

Default behavior of OpenRADAR was to connect to FGMS. For our purposes we needed to replace the FGMS data fetcher with our code that fetched the data from VirtualAir backbone and pushed them to visualization pipeline. Architecture of OpenRADAR allowed us to do it in straightforward manner and we implemented the necessary code

Data structure used for aircraft's world position is defined via AviationSimNet specification [4]. Current state of the project allows multiple instances of FlightGear to exist and interact while being visualized by OpenRADAR. Demo simulation is captured in figure 7.2. White arrow point towards the location of a virtual plane.

Based on these observations we believe that other flight simulator such as X-Plane and Microsoft Flight Simulator with appropriate interfaces would be suitable as data providers [11].


Figure 7.2: FlighGear and OpenRADAR demo simulation

# 8 Conclusion and Future Prospects

In this thesis, we present a new approach to extending CERTI architecture with previously unsupported language bindings. The method used supports fast and manageable bug fixing and patch distribution across the whole architecture. Using python based automated code generator makes it possible to introduce new languages more efficiently than before.

While this thesis aims primarily at High Level Architecture, we believe the outcomes may be applied across the modern architectures facilitating distributed simulation.

One of the practical outcomes of this thesis is the publicly available Java LibRTI – jCERTI. Its functionality was demonstrated by the modifications to the virtual radar screen software that made interoperation with flight simulator possible.

Java LibRTI was an anticipated extension of the existing CERTI platform and it is now available from source code repository [7], compatible with the soon-to-be-released CERTI version 3.4.0.

We see many possibilities for future work. One of the options is to extend current jCERTI codebase with unit tests that can be used to facilitate continuous integration. jCERTI would use expanded support of IEEE 1516 interface specification at least to the same extent as the C++ version. Another obvious direction for future work is implementing LibRTI in another language (i. e. .NET platform). This would give developers the freedom to choose an appropriate language for their HLA compliant applications using CERTI.

# Bibliography


[1] *IEEE Std 1516-2000: "IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Framework and Rules"*. 2000.

[2] *IEEE Std 1516.1-2000: "IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Federate Interface Specification"*. 2000.

[3] *IEEE Std 1516.2-2000: "IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Object Model Template (OMT)"*. 2000.

[4] *AviationSimNet*. 2010. Available online at `http://aviationsimnet.org/`.

[5] *CERTI Homepage*. 2010. Available online at `https://savannah.nongnu.org/projects/certi/`.

[6] *Google Protocol Buffers*. 2010. Available online at `http://code.google.com/apis/protocolbuffers/`.

[7] *jCERTI Repository*. 2010. Available online at `http://cvs.savannah.gnu.org/viewvc/jcerti/?root=certi`.

[8] *PLY (Python Lex-Yacc)*. 2010. Available online at `http://www.dabeaz.com/ply`.

[9] *PyHLA*. 2010. Available online at `http://www.nongnu.org/certi/PyHLA/`.

[10] *SWIG*. 2010. Available online at `http://www.swig.org/`.

[11] *Virtual Air*. 2010. Available online at `http://virtualair.sourceforge.net/`.

[12] NATO Standard Agency. *NATO Modelling and Simulation Standard Profile*. 2000.

[13] Benoit Breholee and Pierre Siron. *CERTI: Evolutions of the ONERA RTI Prototype*. 2002. Available online at `http://breholee.org/files/02F-SIW-018.pdf`.

[14] Mary C. Fischer, Anita Adams, and Gordon Miller. *Aggregate Level Simulation Protocol (ALSP) – Training for the Future*. 1994. Available online at `http://ms.ie.org/alsp/biblio/mors_94_fischer/mors_94_fischer.html`.

[15] Richard M. Fujimoto. *Parallel and distributed simulation systems*. Simulation Interoperability Standards Organization, 2001. Proceedings of the 2001 Winter Simulation Conference.

[16] Deb Fullford and Ben Lubetsky. *Transitioning your DIS simulator to HLA*. MÄK Technologies, 1995.

[17] Gene Hudgins. *The Test and Training Enabling Architecture, TENA, Enabling Technology For The Joint Mission Environment Test Capability (JMETC) and Other Emerging Range Systems*. 2008. Available online at `https://www.tena-sda.org/download/attachments/6750/ITC-Paper-2008.pdf`.

[18] Rainer Koschke and Jean-Francois Girard. *An Intermediate Representation for Reverse Engineering Analyses, Proc. WCRE, pp. 241-250, IEEE Computer Society*. 1998.

[19] F. Kuhl, R. Weatherly, and I. Dahmann. *Creating Computer Simulation Systems; An Introduction to the High Level Architecture*. Prentice Hall, 1999.

[20] Björn Löfstrand. *HLA Standard and Certification*. Pitch Technologies, 2006.

[21] D. C. Miller and J. A. Thorpe. *SIMNET: The Advent of Simulator Networking*. 1995. pp 1114-1123 of Proceedings of the IEEE: Special Issue on Distributed Interactive Simulation.

[22] Paul Murray, Suresh Srinivas, and Matthias Jacob. *Performance Issues for Multi-language Java Applications*. Princeton University, Princeton, NJ, 2000.

[23] Eric Noulard, Jean-Yves Rousselot, and Pierre Siron. *CERTI, an Open Source RTI, why and how*. 2009. Available online at `http://download.savannah.nongnu.org/releases/certi/papers/09S-SIW-015-final.pdf`.

[24] Department of Defence Canberra. *Distributed Simulation Guide*. Australian Defence Simulation Office, 2004.

[25] Department of Defense Defense Modeling and Simulation Office. *RTI 1.3 – Next Generation Programmer's Guide Version 3.2*. 2000.

[26] United States. Dept. of the Army. *Army Science and Technology Master Plan, Chapter VI. Infrastructure*. 1997. Available online at `http://www.fas.org/man/dod-101/army/docs/astmp/c6/P6C3.htm`.

[27] Simulation Interoperability Standards Organization Dynamic Link Compatible HLA API Product Development Group (PDG). *Dynamic Link Compatible HLA API Standard for the HLA Interface Specification (IEEE 1516.1 Version)*. Simulation Interoperability Standards Organization, 2004. Available online at `http://www.sisostds.org/index.php?tg=fileman&idx=get&id=5&gr=Y&path=SISO+Products%2FSISO+Standards&file=SIS-STD-004.1-2004.zip`.

[28] Simulation Interoperability Standards Organization Dynamic Link Compatible HLA API Product Development Group (PDG).

*Dynamic Link Compatible HLA API Standard for the HLA Interface Specification Version 1.3*. Simulation Interoperability Standards Organization, 2004. Available online at `http://www.sisostds.org/index.php?tg=fileman&idx=get&id=5&gr=Y&path=SISO+Products%2FSISO+Standards&file=SISO-STD-004-2004-Final.pdf`.

[29] Roy Scrudder, Björn Möller, and Katherine L. Morse. *High Level Architecture (HLA) Evolved – Product Development Group Status Update*. 2006.

[30] Xiaojun Shen, Ramsey Hage, and Nicolas Georganas. *Agent-aided Collaborative Virtual Environments over HLA/RTI*. Multimedia Communication Research Lab (MCRLab) School of Information Technology and Engineering University of Ottawa, 1999.

[31] Roger D. Smith. *Encyclopedia of Computer Science*. Grove's Dictionaries New York, New York, 2000.

[32] David Wilton. *The Interoperability of Military Simulation Systems in an AUSCANNZUKUS Context*. Australian Defence Science and Technology Organisation, 2001.