CERTI Message specification language

definition, tools, application and perspectives

Eric NOULARD - eric.noulard@onera.fr
with the help of the CERTI developer community

June, 6th 2011

ONERA - Centre de Toulouse
http://www.onera.fr/dtim
2 avenue E. Belin, B.P. 74025, F-31055 Toulouse Cedex 4, FRANCE

ONERA
THE FRENCH AEROSPACE LAB

Summary

Outline

Outline

1. **The needs for a message specification language**
   - Message language specification why?
   - CERTI practical needs
   - Embedded/Real-time CERTI
2. **The CERTI message specification language**
   - Basic features
   - Advanced Features
   - Perspective
3. **Demo**
4. **References**

Distributed Architecture

Communication is needed

As soon as a system has a distributed architecture, each part needs to communicate with each other.

- avionic system of an airplane (ARINC 659, ARINC 654/AFDX, . . .)
- embedded automotive system (CAN, FlexRay, . . .)
- people in a project (Phone, WebEx, E-mail, . . .)
- sailor on a boat (Morse Code, . . .)
- networked computer systems (distributed filesystem [NFS], time synchronization protocol [NTP, IEEE-1588], monitoring [SNMP], . . .)

Message based communication

Many communication systems are **message based**.

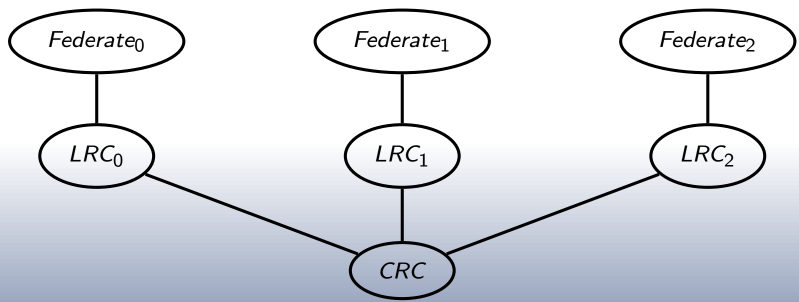
High Level Architecture (HLA) components

An HLA federation is

- a set of federates, which are user defined component,
- a centralized and/or [set of] decentralized RTI (Run Time Infrastructure) components

A set of communicating processes

One or more user federate processes, one or more LRC (Local RTI Component) processes, possibly CRC (Central RTI Component).



HLA specification

The HLA specification beginning with 1.3 [6] then with IEEE-1516-v2000 [8] and now with IEEE-1516-v2010 [9] are describing HLA services as:

- informal textual description, which includes relationship between services,
- some state charts,
- some message sequence chart,

Reminder: HLA is just an example

HLA is taken here as an example but almost **any middleware** has the message exchange need.

HLA specification: informal textual description

A set of services described as messages

The message are exchanged between Federate, LRC and possibly CRC

Create Federation Execution

- Supplied Arguments
  - Federation execution name
  - FED designator
- Returned Arguments
  - None
- Exceptions
  - The federation execution already exists.
  - Could not locate FED information from supplied designator
  - Invalid FED
  - RTI internal error

Easy message structure

We should be able to easily (and may be formally) specify the content of message corresponding to HLA services (including exceptions).

HLA specification: HLA state diagrams and/or MSC

- The message are transition event of HLA state chart [8]
- Some message sequence chart (MSC) [4] of correct HLA federation execution

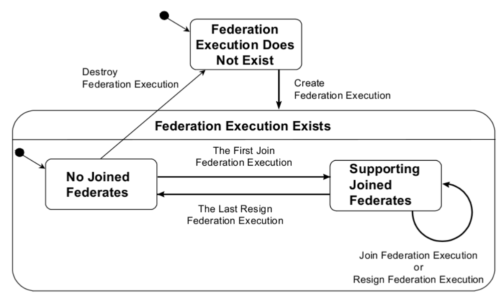


Figure 1—Basic states of the federation execution

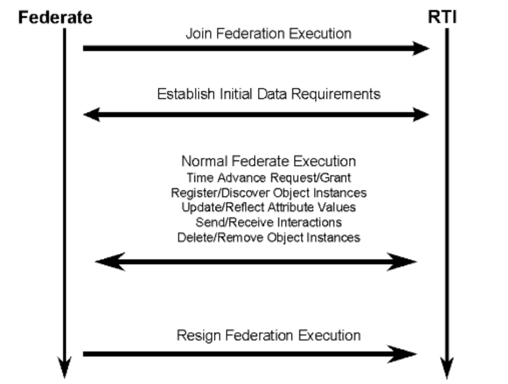


Figure 2—Overall view of federate-to-RTI relationship

More formal message

Message specification and code generation should enhance the formal specification, test and validation of CERTI.

Other middlewares

Middleware message use

Almost all middleware which support distributed execution and communication needs more or less formalized message specification (and code generation for message handling).

- ONC RPC [5] (a.k.a. SUN RPC used in NFS) ⇝ ONC RPC IDL and `rpcgen`
- OMG Data Distribution Service [7] ⇝ OMG IDL and `IDL compiler`
- Any Message-oriented middleware `http://fr.wikipedia.org/wiki/Message-Oriented_Middleware` like JMS [1] (but this one has no IDL, just Java).
- Sometimes there is no middleware at all, "**just message**". This is the case for the Google Protocol buffer [3] and the `protoc` compiler.

Many more IDLs

`http://en.wikipedia.org/wiki/Interface_description_language`

Predictable and/or observable middleware

Generate message [handling] code

Generating verified code is usually far simpler that verifying hand-written code.

If we target predictable and/or observable message-oriented middleware we must have message specification in order to:

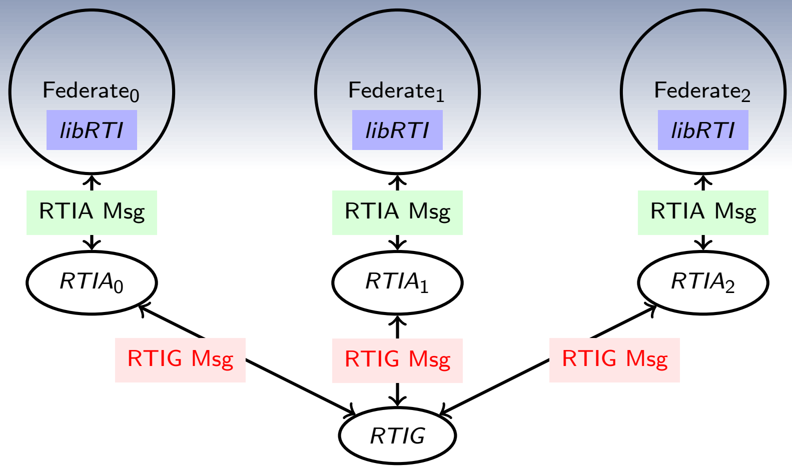
- ensure that we know the **exhaustive** list of message,
- generate serialize/de-serialize (or marshall/un-marshall) code with appropriate properties (bounded memory footprint, bounded execution time, fault tolerance . . .)
- be able to **generate observation** code, specification runtime checking code, [formal] **trace analysis** code (passive testing) . . .

Predictable Middleware

The more formally we can specify message [exchange] in the middleware the more predictable middleware we can produce.

Outline

Current CERTI messaging system



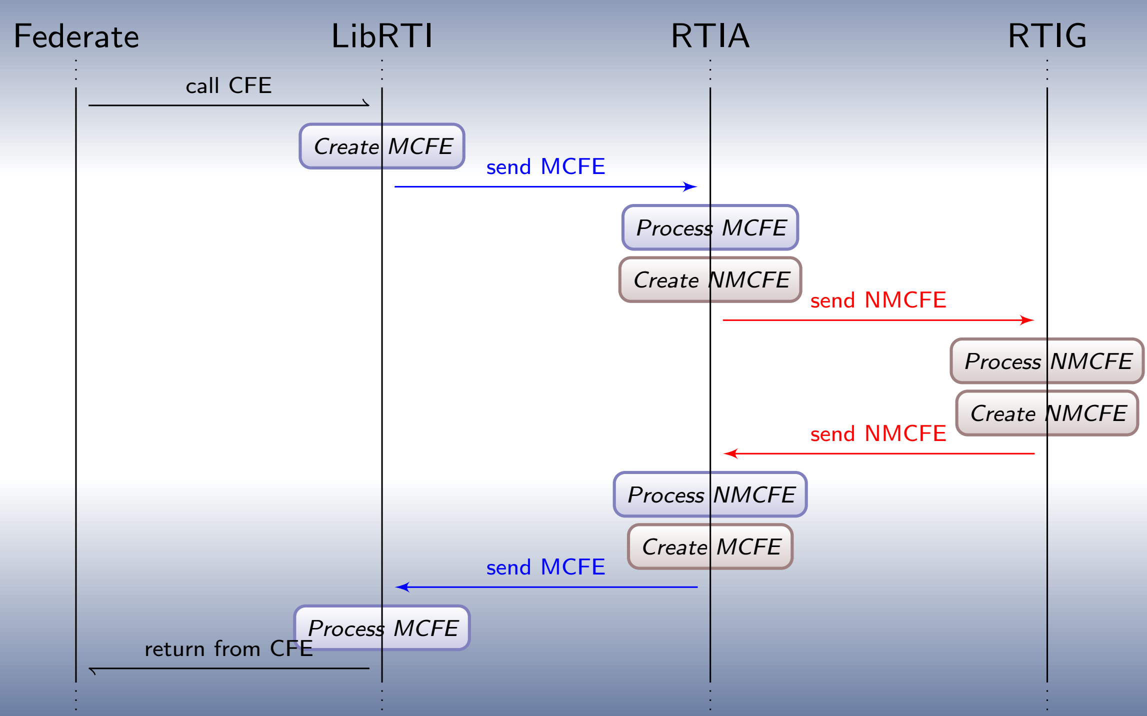
Message a.k.a. RTIA Message

The messages exchanged between libRTI and RTIA (= CERTI LRC).

NetworkMessage a.k.a. RTIG Message

The messages exchanged between RTIA's and RTIG (= CERTI CRC).

Create Federation Execution (CFE) Sequence



Typical messages path (detailed) - I

For a **create federation execution** [distributed] service call here is the sequence:

1. Federate invoke libRTI (RTIambassador service)
2. libRTI **builds an RTIA Message** M_Create_Federation_Execution
3. libRTI **serialize the message** and sends it to RTIA, then usually wait for an answer,
4. RTIA **deserialize the message**
5. RTIA invoke appropriate local service which may. . .
6. RTIA **builds an RTIG Message** NM_Create_Federation_Execution
7. RTIA **serialize the message** and sends it to RTIG, then usually wait for an answer,
8. RTIG **deserialize the message**, invoke the concerned central service and. . .
9. RTIG **builds a new RTIG Message** NM_Create_Federation_Execution which contains the answer (including may be an exception)
10. RTIG **serialize the message** and sends it to RTIA,

Typical messages path (detailed) - II

11. RTIA **deserialize the RTIG message** (he was waiting for this one) and . . .
12. RTIA **builds a new RTIA Message** M_Create_Federation_Execution from the received RTIG Message,
13. RTIA **serialize the message** and sends it to libRTI,
14. libRTI **deserialize the RTIA message** (he was waiting for this answer), and give back the control to the Federate or raise an exception if the Message was conveying one.

A lot of message handling

CERTI is basically a set of **message** handling processes. Messages are built and exchanged (unicasted or broadcasted) between Federates, RTIAs and RTIG.

Typical of MOM (Message-Oriented Middleware)

This is not CERTI-specific probably all MOM do that kind of work.

CERTI Messages C++ source code usage examples I

Listing 1: CERTI libRTI: Join Federation

```
1  RTI::FederateHandle
2  RTI::RTIambassador::joinFederationExecution(
3                          const char *yourName,
4                          const char *executionName,
5                          FederateAmbassadorPtr fedamb)
6  throw (...)
7  {
8    M_Join_Federation_Execution request, answer;
9    request.setFederateName(yourName);
10   request.setFederationName(executionName);
11   privateRefs->executeService(&request, &answer);
12   return answer.getFederate();
13 }
```

- Line 8 declares 2 message objects of type M_Join_Federation_Execution,
- Lines 9–10 setup message content,
- Line 11 call the message send/receive generic service,
- Finally line 12 we return the expected value from the answer

CERTI Messages C++ source code usage examples II

Listing 2: CERTI libRTI: generic execute service

```
void
RTIambPrivateRefs::executeService(Message *request, Message *answer) {
  // send request to RTIA
  try { request->send(socket,msgBufSend); }
  catch (NetworkError) {
    throw RTI::RTIinternalError("libRTI: Network Write Error");
  }
  // waiting RTIA reply.
  try { answer->receive(socket,msgBufReceive); }
  catch (NetworkError) {
    throw RTI::RTIinternalError("libRTI: Error waiting RTI reply");
  }
  // Services may only throw exceptions defined in the HLA standard
  // the RTIA is responsible for sending 'allowed' exceptions only
  processException(answer);
}
```

CERTI Messages C++ source code usage examples III

As it can be seen in this second listing, message handling is generic and all-over-the-place in the CERTI code.

Manual usage for generated code

The usage of message object is hand-written but the source code of message itself may ?must? be generated.

CERTI messages numbers: code generation needs

There is currently a lot of messages:

- 153 Message types
- 106 Network Message types

Multi-language binding

We want to generate the code for several languages: C++, Java, Python, may be more . . .

We must generate - boring to write code

- serialize/deserialize code
- virtual constructor (the factory method pattern [2])

We should [be able to] generate

- self verifying code (e.g. required field should be there)
- may be observability code

Outline

Embedded and/or Real-time CERTI

Embbeding CERTI

We are targeting to produce an embeddable and or realtime version of CERTI. Those specialized CERTI instance much fullfill several ressource constraints. Since CERTI is essentially a message processing library being able to produce message code is necessary (but not sufficient) for reaching this goal.

A MUST-DO

We have to generate the message handling code if we want to generate an embeddable and/or real-time CERTI.

Outline

The CERTI message specification language

Outline

CERTI Message example

CERTI Message language

CERTI Message is home-brewed **message specification** language, used specify the content of a message. Then a code generator (message compiler) may be used to generate helper code for using messages.

Listing 3: CERTI Message

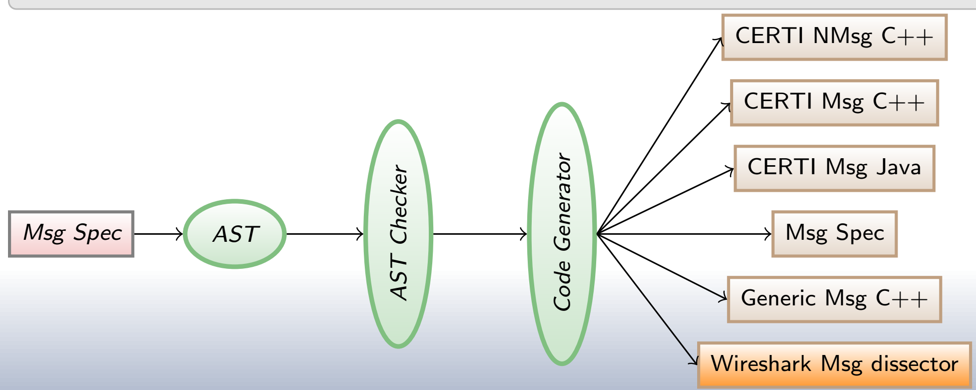
```
1 message M_Create_Federation_Execution : merge Message {
2   required string federationName // the federation name
3   required string FEDid          // the Federation ID (filename)
4 }
```

- CERTI Message (Federate/libRTI ↔ RTIA) M_Create_Federation_Execution is defined at line 1,
- It contains two **string** fields which are **required**,
- The defined message is a **merge** from another **Message** which has been previously defined. **Merging** is a kind of message content inheritance, meaning that the content of the currently defined message will be the content of the **merge** target **plus** the defined fields.

CERTI Message generator architecture

A classical compiler design

The CERTI message generator has a classical compiler architecture with a parser front-end which turns the specification file in a intermediate AST-like structure, this AST is then checked (and sometimes augmented), afterward several backends may generate source code (msg, C++, Java, etc...)



CERTI Message Features

- A message specification file may have a **version**
- In a each specification file one can define a **package** name where the message defined in the file will be put,
- The CERTI message language defines the following **basic** types byte, **bool**, onoff, **uint8**, **uint16**, **uint32**, **uint64**, **int8**, **int16**, **int32**, **int64**, **double**, **float**, **string**.
- There is 3 type constructors:
  - **enum** which may be used to defined enumerated types,
  - **message** which is used to specify a message content,
  - **native** [message] which may be used to reference **natively** implemented message.
- A message contains 0 or more typed fields. The field type may be basic type or any already defined **enum**, **message** or **native**.
- A field may have a **qualifier**:
  - **optional** meaning that the field may be present in the message or not,
  - **required** meaning that the field is mandatory and will be in each message of this type,
  - **repeated** meaning that the field is a sequence of 0 or more items of this type.

CERTI Message BNF I

Listing 4: Courtesy of Lucas ALBA

```
<identifier> ::= [a-zA-Z][a-zA-Z0-9]*
<number> ::= [0-9]+

<messageSpecification> ::= <package> <version> <message>* <factory>
<package> ::= package <identifier>
<version> ::= version <version_identifier>
<version_identifier> ::= <number> ''.'' <number>

<message> ::= <native> | <integralMessage>

<native> ::= native <identifier> ''{''
              [<representation>]   <langage>*
             ''}''
<representation> ::=  representation ( <basic_type> | combine )
<langage> ::= langage <langage_name> ''['' <texte>  '']''

<integralMessage> ::= message <message_name> '':'' merge <message_name>
            ''{'' <field_list> ''}''

<field_list> ::= <field>*
<field> ::= <simple_field> | <combine_field>
<simple_field> ::= <qualifier> <type> <identifier>
```

CERTI Message BNF II

```
<combine_field> ::= combine <identifier> ''{'' <field_list> ''}''
<qualifier> ::= required | repeated | optional

<type> ::=  <basic_type> | <Message>

<basic_type> ::= onoff | bool | string | byte |
                 int8 | uint8 | int16 |
                 uint16 | int32 | uint32 |
                 int64 | uint64 |
                 float | double

<message_name> ::= <identifier>
<langage_name> ::= <identifier>

<factory> ::=  factory  <identifier> ''{''
                 <factory_creator> | <factory_receiver>
               ''}''

<factory_creator>  ::= factoryCreator
                        <identifier> <identifier>(<identifier>)
<factory_receiver> ::= factoryReceiver
                        <identifier> <identifier>(<identifier>)
```


Outline

Native Message

Living with existing code

Introducing message specification should not generate complete rewrite of the code. Sometimes its easier to live with existing code.

Listing 5: Native Message

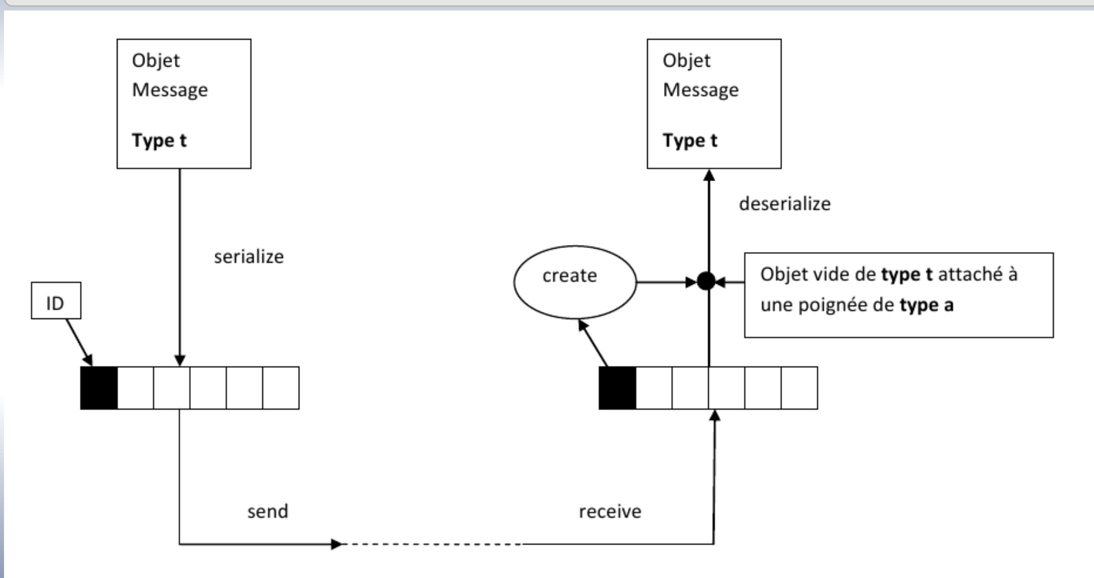
```
1 // Message is the base class for message exchanged between
2 // RTIA and Federate (libRTI) AKA CERTI Message.
3 // Every message which is a merge from Message will first
4 // include the content of a Message
5 native Message {
6     language CXX    [#include "Message.hh"]
7     language Java   [import certi.communication.CertiMessage]
8 }
```

- A "**native**" message is a message whose content is defined in a language specific manner. The source code defining the "**native**" is not generated by the CERTI Message compiler.
- **Message** is defined in C++ by line 6
- **Message** is defined in Java by line 7

Factory Method

Polymorphic reconstruction of message

We want to polymorphically reconstruct the message received.



[possibly] Bounded/Fixed size encoding

Receiver decoding order

CERTI encoding works like CDR (CORBA encoding): endianity of the message is the endianity of the sender.

User Information (Msg Payload)	ID	Msg Length	Endianity	bytes stream begin
msg dependent size	4 bytes	4 bytes	1 byte	

- Fixed size Header : 5 bytes, Endianity and Message Length.
- ID : used for polymorphic reconstruction (factory method)
- Message Payload: could be enforced to fixed sized by the message compiler.

Outline

Conclusion

1. Improve message code generator quality (error handling) [partially done by Lucas ALBA]
2. Implement C backend, [on-going work by Daniel JARTOUX]
3. Implement Wireshark dissector backend, [on-going work by Daniel JARTOUX]
4. Generate self verifying debug code for C++,
5. Implement Python backend,
6. Make the generator less-CERTI specific.
7. Work on an eventual complementary language in order to help formal trace verification.

Outline

Small Demo

Should work

Address book example.

Outline

References I

Java Message Service specifications, v1.1 edition, April 2002.
http://jcp.org/en/jsr/detail?id=914.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.
Design Patterns: Elements of Reusable Object-Oriented Software.
Addison-Wesley Professional, 1994.

Google.
Google Protocol Buffers developer guide, April 2010.
http://code.google.com/intl/fr/apis/protocolbuffers/docs/overview.html.

I. ITU.
Recommendation Z. 120.
Message Sequence Charts (MSC'96), 1996.

Sun Microsystems.
XDR: External Data Representation standard.
RFC 5531, May 2009.
See http://www.ietf.org/.

U.S. Department of Defense.
High Level Architecture Interface Specification, version 1.3, 4 1998.

OMG.
Data Distribution Service for Real-time Systems, Version 1.2.
Object Management Group, formal/07-01-01, January 2007.

References II

IEEE Computer Society.
IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)–Federate Interface Specification, IEEE std 1516.1-2000 edition, 9 2000.

IEEE Computer Society.
IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)–Federate Interface Specification, IEEE std 1516.1-2010 edition, 2010.