Projet d'initiation à la recherche

Spécification et implantation d'un générateur de messages pour CERTI

Lucas ALBA
2ème année
Supaéro
ISAE

PIR effectué à l'Onera
DTIM
Tuteur : Eric NOULARD

Table des matières

Introduction

Le cadre du PIR

J'ai effectué mon PIR pendant le mois de juin 2010 au Département de Traitement de l'Information et Modélisation (DTIM) de l'ONERA à Toulouse.

Le thème du PIR

Ce PIR s'inscrit dans un contexte de recherche particulier qui est celui de l'unité de recherche Systèmes Embarqués et Répartition (SER) de l'ONERA. En effet cette unité de recherche a développé depuis plus de 10 ans un RTI [1] HLA [2][1] open source nommé CERTI afin de concevoir, d'étudier et d'expérimenter les concepts et techniques de simulation distribuées (les notions de HLA et de RTI seront expliquées plus en détails dans le corps du rapport).

L'objectif du projet est de poursuivre le développement d'un générateur de code prenant en entrée la spécification complète d'un jeu de messages et produisant en sortie le code nécessaire à la création/envoi et la réception/reconstitution de ces messages dans divers langages. L'approche envisagée est similaire aux Protocol Buffer de Google [2] mais qui ne peuvent être utilisés dans notre cas pour satisfaire nos contraintes (maîtrise de la taille des messages et de l'encodage, générations de classes de messages polymorphes, totale autonomie du code généré etc. . .)

Un prototype de ce générateur écrit en langage Python existe. L'objectif du projet sera dans un premier temps, après un rapide état de l'art d'écrire la spécification du générateur puis d'amener le prototype dans un état utilisable pour CERTI. Le générateur sera écrit en Python et devra générer du code C++ (ainsi que Java et Python si le temps le permet).

Organisation du rapport

Le rapport s'organise autour de deux grandes parties. D'abord on trouvera une partie sur le contexte de recherche dans lequel s'inscrit le PIR, cette partie étant essentielle pour comprendre le sujet du PIR. Une deuxième partie sera consacrée au travail effectué pendant le PIR, c'est-à-dire une mise en pratique directe des notions vues dans la première partie.

1. Run-Time Infrastructure : Le RTI permet d'assurer la communication entre différents participants à une simulation
2. High Level Architecture : HLA est une architecture pour des simulations distribuées. En utilisant HLA, des simulations sur ordinateur peuvent communiquer avec d'autres simulations sur ordinateur, indépendamment de la plateforme de l'ordinateur.

1 Le contexte de recherche

Remarque : cette partie fait à plusieurs reprises référence au travail d'Eric NOULARD et de la communauté de développeurs de CERTI et notamment à la présentation du langage de spécification [3].

1.1 Le besoin d'un générateur de message pour CERTI

1.1.1 High Level Architecture (HLA)

Comme nous l'avons dit en introduction, CERTI est un RTI HLA. Commençons par définir le terme HLA. HLA est une architecture pour des simulations distribuées. En utilisant HLA, des simulations sur ordinateur peuvent communiquer avec d'autres simulations sur ordinateur, indépendamment de la plateforme de l'ordinateur. Les participants à une simulation sont appelés les *fédérés* (ils sont potentiellement sur des machines différentes). Pour pouvoir communiquer entre eux, ces fédérés utilisent un composant RTI centralisé (appelé CRC pour Central RTI Component) et/ou des composants RTI décentralisés (appelés LRC pour Local RTI Component). L'ensemble des fédérés et des composants RTI forment une *fédération*. On peut voir une représentation d'une fédération HLA sur la figure 1.1 extraite de [3].

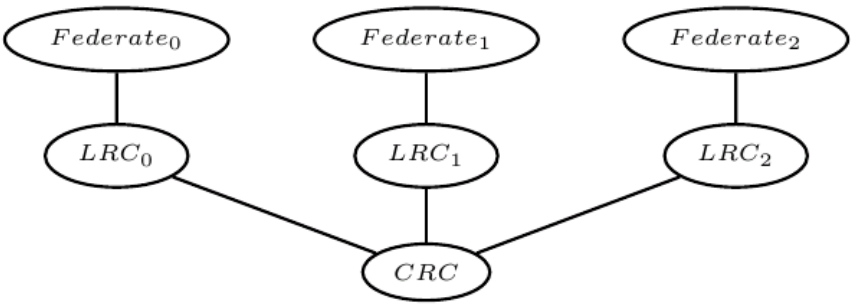

FIGURE *1.1 – Représentation d'une fédération HLA*

Encore une fois les fédérés ainsi que les composants RTI (CRC ou LRC) peuvent se trouver sur des machines différentes. Au final HLA est donc un ensemble de processus qui communiquent entre eux (processus des fédérés et processus des composants RTI).

A partir d'une fédération, la norme HLA (la version actuelle est IEEE-1516-v2010 [4]) spécifie

des *services* qui doivent être rendus par l'architecture. Ces services peuvent être décrits par :

- une description textuelle informelle qui inclut les relations entre services,
- des diagrammes d'état,
- des diagrammes de séquences.

Nous n'entrerons pas plus en détails dans la description de HLA car les notions exposées ici sont suffisantes pour comprendre le sujet du PIR. On peut maintenant expliquer brièvement ce qu'est CERTI.

1.1.2 CERTI

Comme son nom l'indique, CERTI est un *RTI (Run-Time Infrastructure)*, autrement dit un ensemble de composants permettant de faire communiquer entre eux les fédérés d'une fédération HLA. CERTI possède un CRC appelé RTIG et des LRC appelés RTIA. CERTI est cependant un RTI particulier car il utilise des *messages* pour rendre les services spécifiés par la norme HLA. Ainsi à chaque service rendu correspond un ensemble de messages générés et échangés par les composants de CERTI et les fédérés. On peut voir une représentation d'une fédération utilisant CERTI avec les messages échangés entre les différents éléments sur la figure 1.2 extraite de [3].

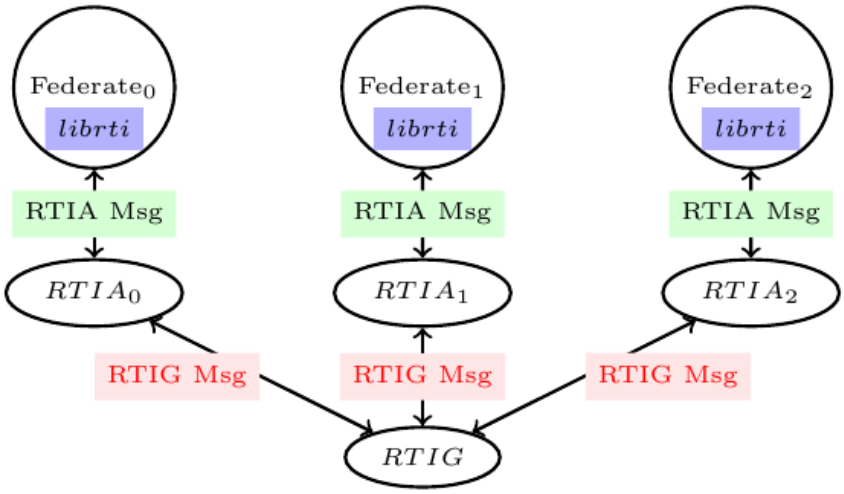

FIGURE *1.2 – Représentation d'une fédération utilisant CERTI*

Là encore le PIR ne justifie pas d'aller plus loin dans la description de CERTI. Il faut surtout comprendre que pour fonctionner et rendre les services attendus CERTI a besoin de gérer un grand nombre de messages (environ 250).

1.1.3 Génération de code et langage de spécification des messages

L'utilisation des objets `Message` dans le code source de CERTI est écrite "à la main". En effet si on considère un objet `Message` qui doit être envoyé par un élément et reçu par un autre, ce message devra être créé, renseigné (ces attributs devront être remplis), envoyé sur le réseau sous forme d'octets (c'est ce qu'on appelle *sérialiser*), reçu puis recréé à partir du flux d'octets (c'est

ce qu'on appelle *désérialiser*). Il faudra donc écrire à la main l'appel des méthodes permettant d'effectuer ces différentes étapes. Ceci étant dit, faut-il écrire ces méthodes proprement dites à la main ? Surtout faut-il les écrire à la main pour environ 250 objets `Message` possibles sachant qu'elles ne vont différer que légèrement d'un objet à l'autre. A priori la réponse est non, d'où l'idée de créer un *générateur de code* capable de générer les classes de messages contenant toutes les méthodes nécessaires à la création et l'échange des messages. Bien entendu le générateur de code ne peut générer à partir de rien. Il faudra donc lui spécifier en entrée les caractéristiques de l'objet `Message` dont on veut générer la classe. Ceci sera fait grâce à un *langage de spécification de messages* que nous détaillerons plus loin.

Finalement pour pouvoir gérer son grand nombre de message de manière efficace, CERTI a donc principalement besoin de deux choses :

- un générateur de code permettant la création et l'échange des messages,
- un langage de spécification de messages qui sera traité par le générateur pour générer le code de gestion de ces messages.

1.2 Le fonctionnement du générateur de message (ou générateur de code)

Remarque : Dans la suite du rapport on pourra utiliser indifféremment les termes de "générateur de code" ou "générateur de message".

Essayons de détailler le fonctionnement du générateur de code. Il prend en entrée un fichier écrit en langage de spécification et il produit en sortie un code dans un langage de programmation choisi (C++, Java, Python ...). En résumé, il traduit un langage en un autre langage. On peut voir ce fonctionnement général du générateur de code sur la figure 1.3. L'intérêt n'est pas simplement la traduction, mais bien le passage d'un langage simple d'utilisation (le langage de spécification) à un langage de programmation avec des parties entières de code générées. Ce travail réalisé par le générateur de code est en fait celui de n'importe quel *compilateur*. En effet, un compilateur va prendre en entrée un langage donné et générer une sortie dans un autre langage, l'intérêt étant que le langage d'entrée soit de plus haut niveau que le langage de sortie. Ainsi le compilateur réalise un travail qui serait très pénible s'il fallait le faire à la main.

Prenons l'exemple du compilateur `gcc`. Il prend en entrée (ce qu'on appelle aussi *Frontend*) un langage de programmation (C++, Fortran ...) et il produit en sortie (ce qu'on appelle aussi *Backend*) un langage machine spécifique du microprocesseur utilisé (intel x86, ppc ...). Mais comment réalise-t-il cette traduction ? En fait la traduction se fait en plusieurs étapes. D'abord le compilateur doit vérifier que le fichier fourni en entrée est correct grammaticalement. En effet, sans entrer dans des détails de linguistique, les langages informatiques sont, commme les langues parlées et écrites, régis par des règles de grammaires. Le compilateur va donc d'abord faire ce que ferait un lecteur face à un texte qu'il découvre, c'est-à-dire vérifier que les phrases de ce texte respectent bien les règles de grammaire de la langue dans lequel le texte est écrit. Un lecteur fera ce travail de vérification de manière presque inconsciente surtout si c'est sa langue maternelle qu'il lit. Le compilateur, lui, le fait de la façon suivante :

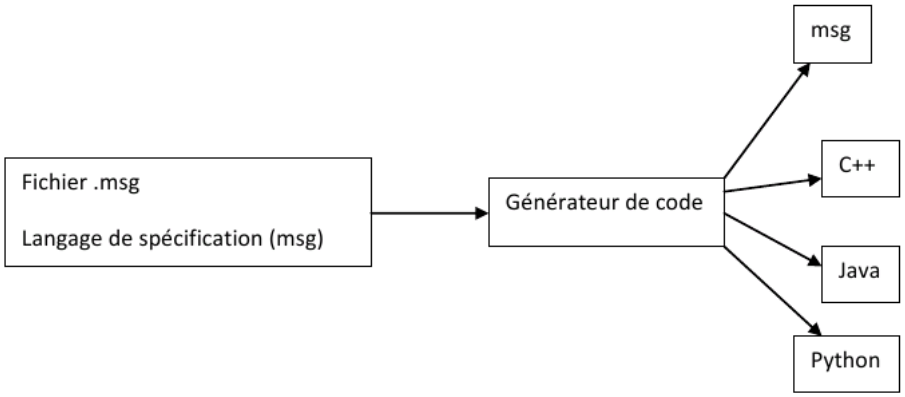

FIGURE *1.3 – Fonctionnement global du générateur de code*

1. il découpe le fichier d'entrée en unités de base du langage appelés *léxèmes*. Ce travail est effectué par un outil appelé *lexer*.
2. il vérifie que les groupes de léxèmes permettent de vérifier les règles du langage d'entrée (grammaire) dont il a la connaissance. Ce travail est effectué par un outil appelé *parser* (en français *analyseur syntaxique*).

Le couple (lexer+parser) permet donc d'assurer que le fichier d'entrée est correct du point de vue syntaxique et doit normalement arrêter le processus de traduction si ce n'est pas le cas. Un fois cette étape passée avec succès, le compilateur génère à partir du fichier d'entrée un objet intermédiaire appelé *AST (Abstract Syntax Tree)* (En fait la création de l'AST se fait en même temps que l'analyse syntaxique par le parser). Cet objet est une abstraction du fichier d'entrée et contient donc les mêmes caractéristiques que le fichier d'entrée. Pour reprendre l'analogie avec un langage humain, un lecteur va lire un texte et l'analyser syntaxiquement (i.e. le parser) pour former des entités abstraites telles que sujet, verbe ou compléments qui seraient des éléments d'un AST. Le compilateur fait la même chose avec le fichier d'entrée via le parser. Cet objet AST sera lui aussi examiné pour vérifier qu'il ne contient pas d'erreur. Les erreurs détectées à ce stade ne sont pas des erreurs de grammaire détectables par le couple (lexer+parser). Là encore la traduction (ou génération) doit s'interrompre si l'AST contient des erreurs. L'AST peut également subir des modifications/annotations ou des contrôles. Dans le cas de gcc ce peut être une passe d'optimisation et dans le cas de notre générateur de message c'est l'étape dite "check" (vérification que tous les types sont définis, comptage des héritiers et vérification des factory...). Une fois cette étape passée avec succès, le compilateur utilise l'AST pour générer la sortie en fonction du Backend choisi. Le processus de génération pour un compilateur tel que gcc est résumé sur la figure 1.4. Il existe par ailleurs de nombreuses références sur la compilation et notamment le "Dragon book" [5] mais les notions exposées ici suffisent pour la compréhension du rapport.

Comme nous l'avons dit le générateur de code de CERTI effectue le même travail qu'un compilateur. Il est donc logique que son mode de fonctionnement soit le même que celui d'un générateur tel que gcc. Il suffit alors d'adapter le cas de gcc au cas du générateur de code de

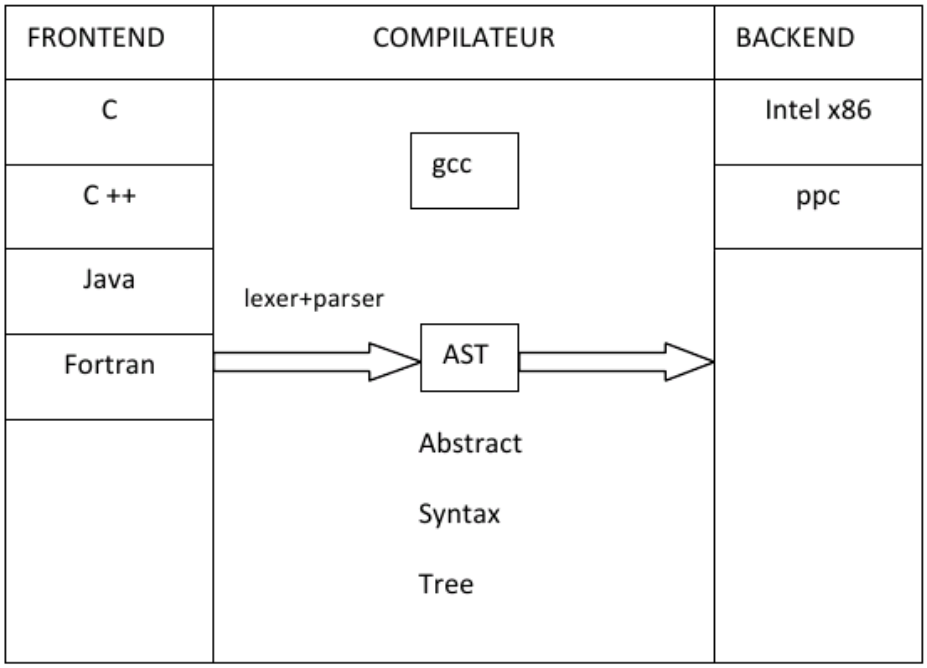

FIGURE *1.4 – Fonctionnement du compilateur gcc*

CERTI où l'entrée est un fichier écrit en **langage de spécification** (on pourra l'appeler **msg** pour simplifier) et la sortie un langage de programmation (C++, Java, Python,...) ou même msg (on considère que le générateur doit être capable de regénérer l'entrée). Le reste est tout à fait similaire comme on peut le voir sur la figure 1.5.

La particularité du générateur de code de CERTI est qu'il est écrit en langage **Python**. Or Python propose un [générateur de] couple (lexer+parser) (mais ce n'est pas le seul) appelé *PLY (Python Lex Yacc)* et c'est ce couple (lexer+parser) que l'on utilise dans notre générateur de code. Cet outil permet de définir facilement en langage Python les règles de grammaire du langage msg en entrée et de créer le couple (lexer+parser) permettant de les vérifier. Ces règles seront donc codées dans le générateur et utilisées lors de l'analyse syntaxique. Pour comprendre comment fonctionne PLY et de manière générale un analyseur syntaxique on peut se reporter à [6, 7] mais ce n'est pas dans l'objet du PIR de détailler plus cet aspect.

1.3 Création, envoi, encodage et réception des messages

Reprenons l'exemple d'un objet `Message` qui doit être créé puis envoyé par un élément de la fédération HLA et reçu par un autre via un envoi sur le réseau. Comme nous l'avons dit le générateur de code pourra se charger de générer le code permettant de réaliser ces différentes étapes. Il faut tout de même savoir ce qui se passe précisément lors d'un envoi de message pour comprendre l'utilité du code généré. Sans entrer pour l'instant dans les détails du code généré, il faut malgré tout donner quelques précisions. En fait deux questions se posent principalement :

– l'objet `Message` envoyé possède un certain type. Mais une fois sérialisé et envoyé sur le

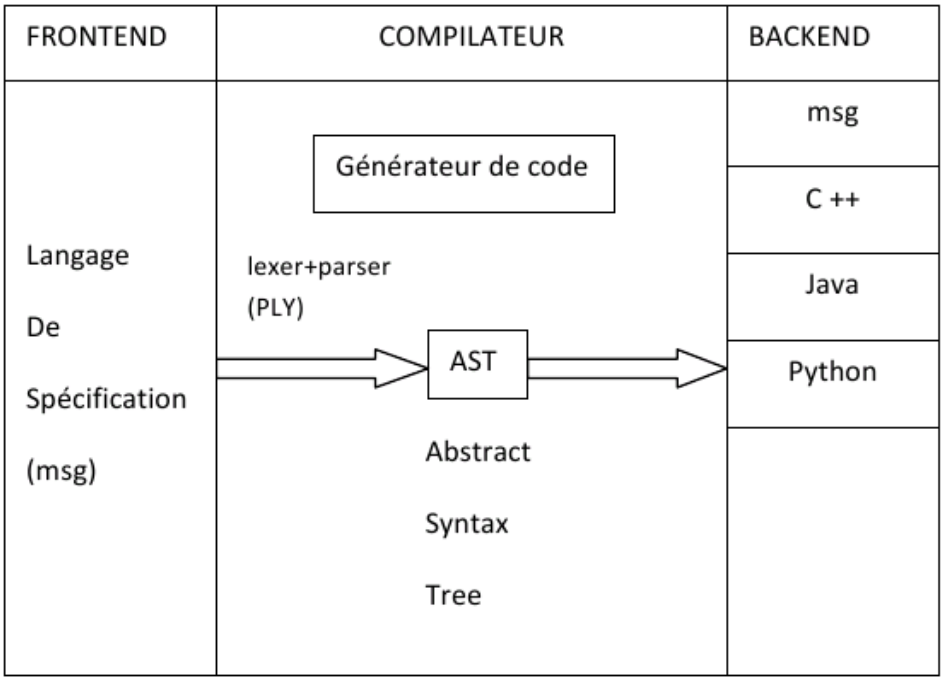

FIGURE *1.5 – Fonctionnement du générateur de code de CERTI*

réseau, comment faire lors de la réception pour reconstruire un objet du bon type ?
– comme l'objet envoyé va être sérialisé (i.e transformé en flux d'octets), quel va être l'**encodage** utilisé ?

1.3.1 Reconstruction d'un message envoyé sur le réseau

Concernant la première question, le seul moyen est que le message sérialisé (par la méthode `serialize`) conserve l'information de son type même une fois envoyé sur le réseau. Autrement dit un certain nombre d'octets sera réservé à l'information du type de l'objet envoyé (ce qu'on appelle **ID**). Cet ID n'est pas litéralement le type de l'objet mais il permettra de choisir le bon type lors de la reconstruction de l'objet. Pour comprendre comment sera recréé (reconstruit) l'objet après réception (par la méthode `receive`) avec le bon type, il faut comprendre comment a été pensé le code généré. La méthode générée permettant de recréer un objet après réception est la méthode `create`. Cette méthode utilise un patron de conception appelé *factory*. Elle fait appel dans son corps aux constructeurs des objets qu'elle est capable de reconstruire. Son type de retour est un pointeur (si on est en C++) sur un ancêtre commun à tous les types d'objet qu'elle est capable de reconstruire. Ce que fait cette méthode c'est que, à partir d'un identifiant de type passé en argument, elle renvoie un pointeur sur l'ancêtre de ce type (en Java on dirait une poignée du type de l'ancêtre) et elle attache à ce pointeur ou cette poignée un objet du type passé en argument. Cela impose notamment que tous les messages que l'on souhaite pouvoir reconstruire soient spécifiés en entrée comme héritant d'un type de base qui est le type de retour de cette méthode `create`. La signature de cette méthode est alors : `type_base* create(sous_type)`

Remarque importante : En fait la notion d'héritage n'est pas tout à fait exacte ici.

La véritable relation qui existe entre l'ancêtre et ses descendants est une relation de *fusion* (*merge* en anglais) dans la mesure où un message qui est le merge d'un autre contiendra i.e. sera la fusion du message dont il merge. Il se trouve qu'en C++/Python cette relation est implémentée avec un héritage. Dans la suite du rapport, les termes "héritage" ou "héritier" pourront être employé abusivement pour désigner cette relation particulière du merge. Il faudra que le lecteur soit conscient de cet abus.

Une fois cet objet du bon type attaché à la poignée, la méthode `deserialize` permet de reconstruire les attributs de l'objet à partir du flux d'octets.

L'astuce utilisée pour la reconstruction est que l'identifiant de type passé en argument (qui sera donc aussi l'ID porté par le message sérialisé) ne sera qu'un simple entier. En effet lors de la génération du code, le générateur dénombre les sous-types susceptibles d'être reconstruits (encore une fois cela est fixé au niveau de la spécification) et génère en sortie un **enum** associant à chacun de ces sous-types un entier (0, 1, 2...). La méthode générée `create` utilise ensuite cet enum pour reconstruire le bon type en fonction de l'entier passé en argument. Il faut préciser que la méthode `create` aura la même implémentation quel que soit le type de message reconstruit alors que les méthodes `serialize` et `deserialize` auront elles une implémentation différentes en fonction du type de message. On peut résumer les différentes étapes de l'envoi d'un objet `Message` de type `t` héritant d'un type de base `a` (i.e. son ancêtre) sur le schéma 1.6.

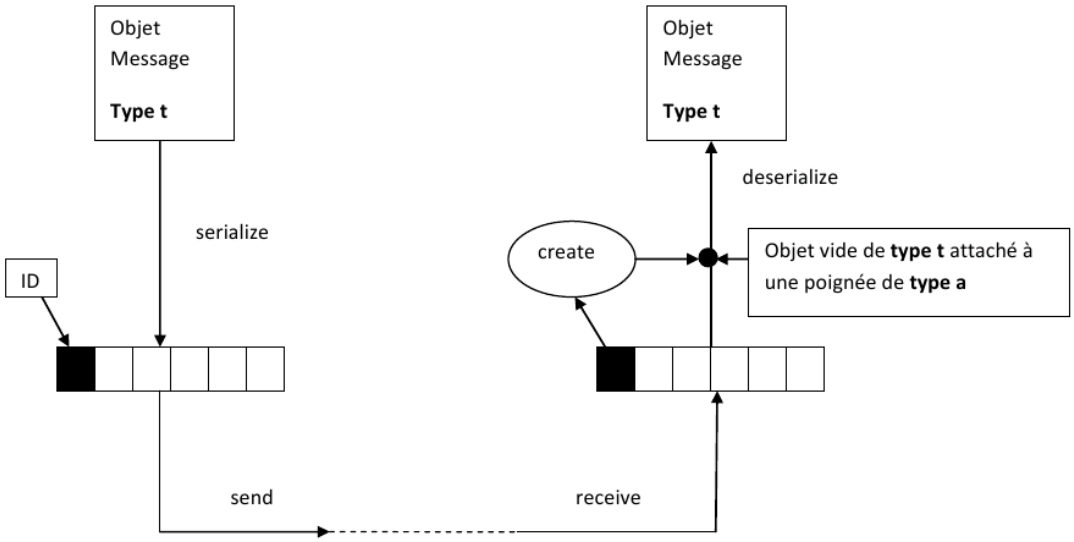

FIGURE *1.6 – Schéma explicatif de la reconstruction d'un message envoyé sur le réseau*

1.3.2 Encodage d'un message envoyé sur le réseau

L'encodage doit notamment spécifier deux choses :

- la taille en octets du message encodé,
- le sens de lecture des octets, autrement dit dans quel ordre sont envoyés les octets, ce qu'on appelle aussi *endianité*.

Concernant la taille du message encodé, l'encodage choisi pour CERTI est un *encodage à taille fixe*. Cela signifie que si dans sa spécification un message contient un entier `int32`, cet entier sera bien encodé sur 32 bits (4 octets) sur le réseau. Un message donné contenant un certain nombre de champs aura donc toujours la même taille en octets sur le réseau d'où la notion d'encodage à taille fixe. Il s'agit d'une différence avec les Google Protocol Buffers où un *encodage à taille variable* est utilisé. Pour un entier par exemple, les Google Protocol Buffers utilisent une méthode de sérialisation appelée `Varints` qui permet d'adapter la taille de l'encodage à la valeur de l'entier sérialisé (un entier plus petit occupe un plus petit nombre d'octets). Cet encodage à taille variable est décrit de manière complète dans [8].

Concernant l'endianité, il n'y a que deux possibilités :

- soit on envoie en premier le bit de poids le plus faible (*LSB* pour Least Significant Bit) et les autres bits suivent par poids croissants jusqu'au bit de poids le plus fort (*MSB* pour Most Significant Bit). On appelle cette endianité *Little Endian*.
- soit on envoie en premier le bit MSB et les autres bits suivent par poids décroissants jusqu'au bit LSB. On appelle cette indianité *Big Endian*.

Il n'y a pas que l'endianité du message sur le réseau car les machines émettrices et réceptrices (potentiellement différentes avec HLA) ont aussi leur propre endianité pour stocker les valeurs en mémoire. Dans le cas de CERTI on considère que le message envoyé a la même endianité que la machine émettrice. Reste alors à savoir si la machine réceptrice a aussi la même endianité que la machine émettrice ou l'endianité inverse. Pour résoudre ce problème, lorsqu'on envoie un message sur le réseau, on indique dans le premier octet l'endianité de la machine émettrice. A partir de là la machine réceptrice sait si elle doit inverser ou non l'endianité du message reçu pour l'adapter à sa propre endianité.

Finalement pour un message envoyé, il faut aussi préciser dans l'encodage son ID (comme on l'a déjà vu), son endianité et aussi la taille du message qui va être lu par la machine réceptrice (car elle ne sait pas a priori où commence et où finit le message). Tout cela donne une certaine structure pour un message encodé comme on peut le voir sur la figure 1.7.

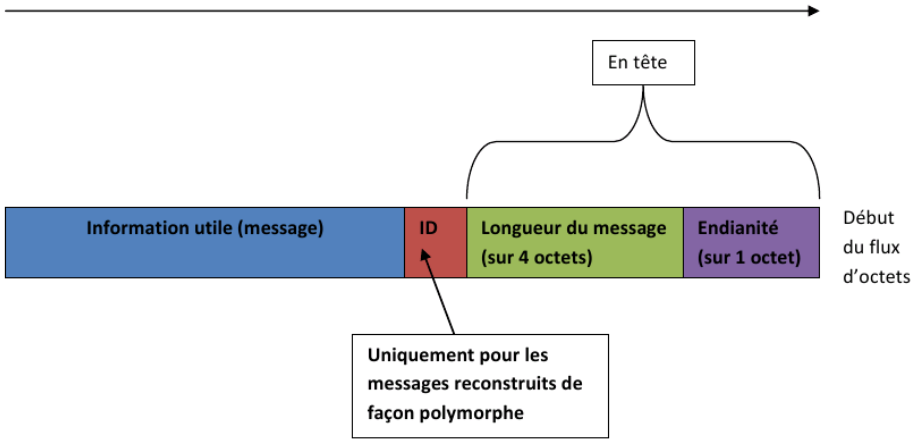

FIGURE *1.7 – Structure d'un message encodé*

2 La mise en pratique au cours du PIR

Remarque : dans la mesure où le générateur de code est écrit en langage Python, une connaissance de base de Python était nécessaire pour ce PIR. Pour cela des références comme [9, 10] sont utiles.

2.1 Description du langage de spécification des messages avec la forme de Backus-Naur (BNF)

Nous avons expliqué la nécessité de générer du code et d'utiliser un langage de spécification de message en entrée du générateur. Pour écrire le générateur, les développeurs de CERTI ont donc dû implémenter les règles de ce langage de spécification qu'ils ont créé. Cette implémentation est faite dans le fichier `GenerateMessages.py`. Le générateur a alors la connaissance des règles du langage et peut, comme nous l'avons déjà dit, faire une analyse syntaxique de l'entrée. Ces règles implémentées dans le générateur peuvent être décrites de manière formelle. Pour faire cette description nous allons utiliser la *forme de Backus-Naur (BNF)* qui est un métalangage permettant de décrire les règles syntaxiques des langages de programmation.

Expliquons en quoi consiste la BNF (une présentation complète est donnée dans [11]). La BNF a pour rôle de fixer des règles à des compilateurs et non à des utilisateurs humains. En BNF, on distingue les *méta-symboles*, les *terminaux* et les *non terminaux*. Les méta-symboles sont tout simplement les symboles de la BNF. Les symboles non terminaux sont les noms des catégories que l'on définit, tandis que les terminaux sont des symboles du langage décrit. Prenons l'exemple de la règle définissant la structure `if` en langage C décrite avec la BNF :

<structure_if> : := if "(" <condition> ")" "{" <code> "}"

<structure_if>, <condition>, et <code> sont des **non terminaux**.

: := est un **méta-symbole** signifiant "est défini par".

if, "(", ")", "{" et "}" sont des **terminaux** du langage C.

Il arrive souvent qu'un non terminal puisse se définir de plusieurs façons. Dans ce cas on utilise le méta-symbole | qui signifie "ou". Par exemple : <catégorie> : := <un> | <deux> | ...

Une description complète d'un langage avec la BNF partira donc de règles de haut niveaux

utilisant uniquement des non terminaux et aboutira, par définitions successives de règles, à des terminaux du langage. La BNF offre aussi la possibilité de définir une régle de manière récursive, c'est-à-dire que le membre définit (à gauche de : :=) se retrouve dans la liste des membres qui les définissent (à droite de : :=).

Maintenant que l'utilisation de la BNF est connue, on peut présenter la description BNF du langage de spécification rédigée pendant le PIR :

```
1
2  <MessageSpecification> ::= <Statement>*
3
4  <Statement> ::= <Comment_block>
5                            | <Package>
6                            | <Version>
7                            | <Factory>
8                            | <Message>
9                            | <Enum>
10
11 <Comment_block> ::= <Comment_line>*
12 <Comment_line>::= ''// '' <texte>
13
14 <eol_Comment> ::= <Comment_line> | '' ''
15
16
17 <Package> ::= package  <Identifier>
18
19 <Version> ::= version <Version_identifier>
20
21 <Version_identifier> ::= <number> ''.'' <number>
22
23
24
25 <Factory> ::=  factory  <identifier> ''{'' <factory_creator> <factory_receiver> ''}''
26               | factory <identifier> ''{'' <factory_creator> ''}''
27
28 <factory_creator> ::= factoryCreator <identifier> <identifier>(<identifier>)
29
30 <factory_receiver> ::= factoryReceiver <identifier> <identifier>(<identifier>)
31
32 <Message> ::= <Native> |  <IntegralMessage>
33
34 <Native> ::= native <identifier> ''{'' <Native_line_list> ''}''
35
36 <Native_line_list> ::= (<Native_line> <eol_Comment>)*
37
38 <Native_line> ::= <Representation_line> | <Language_line>
39
40
41 <Representation_line> ::=  representation ( <basic_type> | combine )
42
43 <Language_line> ::= language <language_name> ''['' <texte>  '']''
44
45 <IntegralMessage> ::= message <message_name> '':'' merge <message_name>
46            ''{'' <field_list> ''}''
47                      | message <message_name> ''{'' <field_list> ''}''
48
49 <field_list> ::= (<field> <eol_Comment)*
50
51 <field> ::= <simple_field> | <combine_field>
52
```

```
53 <simple_field> ::= <qualifier> <type> <identifier> <eol_Comment>
54                   | <qualifier> <type> <identifier>
55          ''{'' default ''='' <value> ''}'' <eol_Comment>
56
57
58 <combine_field> ::= combine <identifier> ''{'' <field_list> ''}''
59
60 <qualifier> ::= required | repeated | optional
61
62 <type> ::=  <basic_type> | <Message>
63
64
65 <basic_type> ::= onoff | bool | string | byte | int8 | uint8 | int16 | uint16 | int32
66        | uint32 | int64 | uint64 |float | double
67
68 <value> ::= integer_value
69             | float_value
70             | bool_value
71             | string_value
72
73
74 <Enum> ::= enum <identifier> ''{'' <Enum_value_list> ''}''
75
76 <Enum_value_list> ::= <Enum_value> <eol_Comment>
77                       | <Enum_value> '','' <eol_Comment>
78                       | <Enum_value> '','' <eol_Comment> <Enum_value_list>
79
80 <Enum_value> ::= <identifier> ''='' <number> '',''
81
82
83
84 <identifier> ::= [a-zA-Z][ a-zA-Z0-9]*
85
86 <number> ::= [0-9]+
87
88 <message_name> ::= <identifier>
89 <langage_name> ::= <identifier>
```

Dans cette BNF le méta-symbole * signifie ”un ou plus”. On trouve notamment dans cette BNF la structure <Factory> qui permet de spécifier les méthodes `create` et `receive` dont on a déjà parlé et la structure <Message> qui permet de spécifier un objet `Message` en donnant ses champs. Il ne faut pas confondre la structure <Enum> qui permet de spécifier des enum souhaités par l'utilisateur avec l'enum automatiquement généré (et absent au niveau de la spécification) contenant les types pouvant être reconstruits par la méthode `create` (voir §1.3.1). Un autre point important est l'utilisation du mot `merge` pour signifier qu'un message hérite d'un autre message. Cette notion d'héritage est importante notamment pour la méthode `create` comme on l'a déjà vu. On pourra éventuellement faire référence à cette BNF dans la suite du rapport.

2.2 Vérification et amélioration du rapport d'erreur

Comme nous l'avons dit dans la partie 1.2, il y a deux étapes de vérification effectuées par le générateur avant de générer la sortie : d'abord l'analyse syntaxique de l'entrée réalisée par le parser et ensuite la vérification des propriétés de l'AST produit. Si des erreurs sont relevées lors de

ces étapes, il faut que le compilateur d'une part arrête le processus de génération et d'autre part envoie un rapport d'erreur le plus précis possible afin que le rédacteur du fichier de spécification puisse facilement corriger ses erreurs. Ce sont ces aspects du générateur que l'on veut tester ici. Pour cela on a écrit des fichiers `.msg` de spécification dans lesquels on a volontairement introduit des erreurs (ces erreurs constituent donc des tests unitaires du générateur). Puis on les a donnés au générateur afin de voir comment il réagissait. Puis on a modifié le générateur afin qu'il arrête la génération du code et qu'il envoie un rapport d'erreur précis (avec notamment la ou les lignes de code où se situe l'erreur) lorsqu'il trouve une erreur. Nous présentons ici les rapports d'erreur obtenus pour trois fichiers .msg après avoir apporté les modifications nécessaires au générateur. Voici les fichiers utilisés :

– le fichier `testMsg_undefined_type.msg` dont voici le contenu :

```
1  package certi
2  version 1.0
3
4  message Person {
5    required int id
6    required string name
7    optional string email
8  }
```

Ce fichier permet de voir le rapport d'erreur lorsqu'un champ d'un message a un type qui n'est pas défini. En effet ici le message `Person` a un champ de type `int`. Mais si on se réfère à la BNF, ce type `int` ne fait pas partie des types de base autorisés par le langage et défini dans la règle <basic_type>. Cette erreur n'est pas une erreur de syntaxe car la règle de syntaxe attend juste pour le type un identifiant qui peut être quelconque a priori. En revanche elle sera relevée lors de la vérification de l'AST car alors le générateur sait que cet identifiant ne peut pas être tout à fait quelconque mais doit soit faire partie d'une liste d'identifiants de base (les types de base), soit être un type défini lors de la spécification. Ici ce n'est ni l'un ni l'autre et c'est ce que dit le rapport d'erreur (voir figure 2.1).

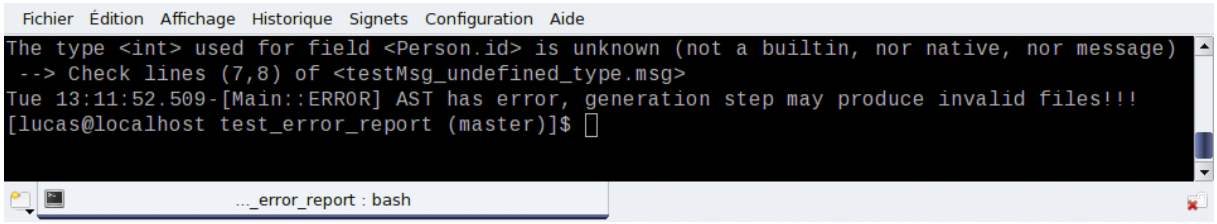

FIGURE *2.1 – Rapport d'erreur pour un type non défini*

– le fichier `testMsg_Syntax_error.msg` dont voici le contenu :

```
1  package certi
2  version 1.0
3
4  message Person {
5    requir string name
6    optional string email
7  }
```

Ce fichier permet de voir le rapport d'erreur pour un cas d'erreur de syntaxe (i.e. détectée lors de l'analyse syntaxique par le parser). En effet pour le champ `name` du message `Person`,

le parser suivant la règle de la BNF s'attend à recevoir en premier un <qualifier> qui est ou bien `repeated`, ou bien `optional`, ou bien `required` mais il reçoit le *token*[1] `requir` peut être dû à une faute de frappe. Toujours est-il qu'il l'interprète alors comme un <identifier> alors qu'il s'attend à recevoir un <qualifier>. C'est ce qui est dit dans la rapport d'erreur de la figure 2.2.

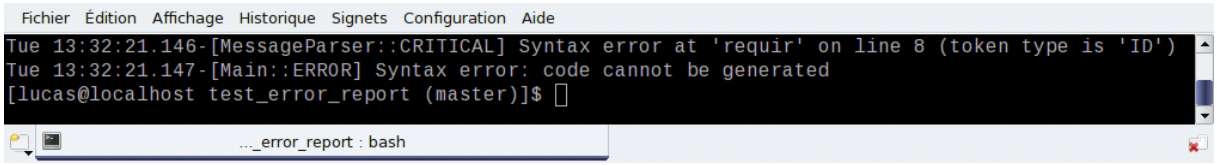

FIGURE *2.2 – Rapport d'erreur dans un cas particulier d'erreur de syntaxe*

– le fichier `testMsg_merge.msg` dont voici le contenu :

```
1  package tutorial
2  version 1.0
3
4  native MessageBuffer {
5    language CXX [#include "MessageBuffer.hh"]
6  }
7
8  message nmsg {
9    required uint32 type
10 }
11
12 message nmsgbis {
13   required uint32 type
14 }
15
16 enum PhoneType {
17   MOBILE = 0,
18   HOME = 1,
19   WORK = 2,
20 }
21
22
23 message PhoneNumber {
24   required string number
25   optional PhoneType type
26 }
27
28
29
30 message Person : merge nmsg {
31   required string name
32   required int32 id
33   optional string email
34   repeated PhoneNumber phone
35 }
36
37
38
39 message AddressBook : merge nmsgbis {
40   repeated Person person
41 }
```

1. un token est un léxème issu du lexer et traité par le parser

```
42
43 native M_Type {
44   language CXX [typedef uint32_t M_Type;]
45 }
46
47 native MStreamType {
48   language CXX [typedef char* MStreamType;]
49 }
50
51
52 factory M_Factory {
53   factoryCreator  nmsg  create(M_Type)
54   factoryReceiver nmsg  receive(MStreamType)
55 }
```

Ce fichier permet de voir le rapport d'erreur quand on essaie d'utiliser plus d'un type "racine" (ou ancêtre final) des autres messages. Plusieurs structures sont présentes dans ce fichier mais on ne s'intéresse ici qu'à la notion d'héritage (ou plutôt de merge comme on l'a déjà dit). En effet on s'est fixé la contrainte de n'avoir qu'un seul ancêtre final (i.e. qui a des descendants mais pas lui même d'ancêtre) qui doit être également le type de retour des méthodes `create` et `receive` de la `Factory`. On laisse la possibilité d'avoir différents ancêtres intermédiaires (spécifié derrière le mot `merge`) mais ils doivent tous se ramener à un unique ancêtre final. Tous les descendants directs ou indirects de cet ancêtre final sont alors susceptibles d'être reconstruits de façon polymorphe par la méthode `create`. Mais dans le cas présent on a deux ancêtres finaux :

– `nmsg` qui est le type de retour de la `Factory` et qui a pour descendant `Person`
– `nmsgbis` qui a pour descendant `AddressBook`

L'unicité de l'ancêtre final (*root merged type*) n'est donc pas respectée et ce que dit le rapport d'erreur sur la figure 2.3.

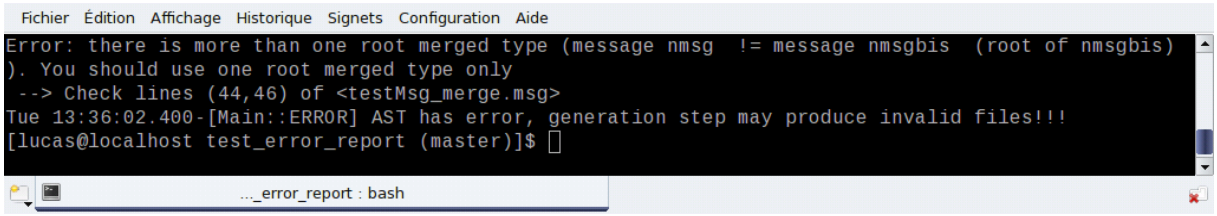

FIGURE *2.3 – Rapport pour une violation de l'unicité de l'ancêtre final*

2.3 Exemples d'application du générateur de code

2.3.1 Exemple 1

L'idéal pour comprendre l'ensemble des étapes par lesquelles on doit passer pour gérer des messages comme ceux de CERTI est de traiter un exemple. Cet exemple ne sera pas un message de CERTI mais sera une adaptation d'un exemple donné pour les Google Protocol Buffers et trouvé sur internet [12]. Plusieurs petits problèmes se sont posés au cours de la mise en oeuvre de cet exemple car au départ le générateur est conçu pour gérer uniquement les messages échangés

dans CERTI. On a donc dû modifier légèrement le générateur pour l'adapter à cet exemple et par la même occasion rendre son usage possible en dehors du projet CERTI. Dans cette exemple on a un fichier de spécification `addressbook.proto` (écrit en langage de spécification des Google Protocol Buffer) qui décrit les classes qui doivent permettre la réalisation d'une application "carnet d'adresses". Ce fichier .proto est donc fourni au compilateur des Google Protocol Buffer qui génère en sortie (en C++ par exemple) le code des classes spécifiées. Ce code peut ensuite être utilisé pour écrire à la main une application permettant de gérer un carnet d'adresse. Nous allons donc adapter cet exemple en prenant le fichier `addressbook.proto` et en créant son équivalent en langage msg sous forme d'un fichier `addressbook.msg` dont voici le contenu :

```
1  package tutorial
2  version 1.0
3
4  native MessageBuffer {
5    language CXX [#include "MessageBuffer.hh"]
6  }
7
8  message nmsg  {
9   required uint32 type
10  required string messageName
11 }
12
13 enum PhoneType {
14   MOBILE = 0,
15   HOME = 1,
16   WORK = 2,
17 }
18
19
20 message PhoneNumber  {
21   required string number
22   optional PhoneType number_type
23 }
24
25
26 message Person : merge nmsg {
27   required string name
28   required int32 id
29   optional string email
30   repeated PhoneNumber phone
31 }
32
33
34 message AddressBook : merge nmsg {
35   repeated Person person
36 }
37
38 native M_Type {
39   language CXX [typedef uint32_t M_Type;]
40 }
41
42
43 factory M_Factory {
44   factoryCreator  nmsg  create(M_Type)
45 }
```

Ce message définit donc un ancêtre final `nmsg` qui est le type de retour de la méthode `create` de la `Factory`. Cet ancêtre final a deux descendants (`Person` et `Addressbook`) susceptibles d'être reconstruits après envoi sur le réseau. Il faut préciser que dans cet exemple on n'étudiera pas l'envoi sur le réseau, mais seulement la sérialisation, reconstruction puis désérialisation d'un objet message de la classe `Person`. On n'a donc pas à spécifier de méthode `receive` dans la `Factory` et il a fallu pour que cela soit possible rendre optionnelle la méthode `receive` au niveau de la structure <Factory> de la BNF (et aussi au niveau de son implémentation dans le générateur). On définit également un enum `PhoneType` qui donne les différents types de numéro de téléphone.

La spécification contient également un message `native` nommé `MessageBuffer`. Le type `native` a été introduit au départ pour pouvoir gérer des classes de la norme HLA qui existaient déjà par ailleurs. Ces classes devaient donc être prises telles qu'elles étaient et ne pouvaient pas être modifiées. C'est justement le cas de la classe `MessageBuffer` implémentée dans `MessageBuffer.hh` et nécessaire pour pouvoir sérialiser et désérialiser un objet. Dans la spécification on se contente de préciser au générateur dans une <Language_line> (voir la BNF) ce qu'il doit faire (à savoir #`include "MessageBuffer.hh"`) dans le cas d'une sortie C++.

On peut ensuite donner ce fichier au générateur qui génère les fichiers `addressbook.hh` et `addressbook.cc` (on a choisi de générer du C++). Il n'est pas nécessaire de présenter ici le code de ces fichiers. Il faut surtout savoir qu'ils contiennent tout le code nécessaire à la réalisation de l'exemple d'application qui suit. Ces fichiers contiennent donc notamment la description de la classe `Person` avec entre autre le constructeur, les méthodes `serialize`, `deserialize` et `create`. On écrit ensuite un fichier `exemple_person.cc` dans lequel on instancie un objet `Person` puis on renseigne certains de ces attributs, on le sérialise et on le reconstruit de façon polymorphe via la méthode `create`. On affiche également les attributs du message envoyé et ceux du message reconstruit afin de vérifier qu'ils sont bien identiques. Le code de `exemple_person.cc` est donné ci-dessous :

```
1
2  #include <iostream>
3  #include <fstream>
4
5  #include "addressbook.hh"
6  #include "MessageBuffer.hh"
7
8  using namespace std;
9
10 int main() {
11
12   tutorial::Person p;
13   tutorial::nmsg*  p2;
14   libhla::MessageBuffer msgBuf;
15   tutorial::PhoneNumber phonenumber;
16   phonenumber.setNumber("1234567890");
17   phonenumber.setNumber_type(tutorial::addressbook::MOBILE);
18   p.setPhoneSize(1);
19   p.setPhone(phonenumber,0);
20   p.setName("John");
21   p.setEmail("john.doe@quisait.fr");
22   p.setId(456);
23
24   cout << "le message envoye" << endl;
```

```
25    cout << "␣" << endl;
26    p.show(cout);
27    cout << "␣" << endl;
28
29    p.serialize(msgBuf);
30    cout << "taille␣du␣message␣encode␣en␣octets␣=␣" << msgBuf.size() << endl;
31    cout << "␣" << endl;
32    //envoi sur le reseau
33    p2 = tutorial::M_Factory::create(p.getType());
34    p2->deserialize(msgBuf);
35    cout << "le␣message␣recu" << endl;
36    cout << "␣" << endl;
37    p2->show(cout);
38    delete p2;
39
40 }
```

Le résultat de l'exécution est présenté ci-dessous :

```
le␣message␣envoyé

[Person␣-Begin]
[nmsg␣-Begin]
␣type␣=␣1
␣messageName␣=␣Person
[nmsg␣-End]
␣name␣=␣John
␣id␣=␣456
(opt)␣email␣=john.doe@quisait.fr
␣␣␣␣phone␣[]␣=
[PhoneNumber␣-Begin]
␣number␣=␣1234567890
(opt)␣number_type␣=0
[PhoneNumber␣-End]
[Person␣-End]

taille␣du␣message␣encodé␣en␣octets␣=␣78

le␣message␣reçu

[Person␣-Begin]
[nmsg␣-Begin]
␣type␣=␣1
␣messageName␣=␣Person
[nmsg␣-End]
␣name␣=␣John
␣id␣=␣456
(opt)␣email␣=john.doe@quisait.fr
```

␣␣␣␣phone␣[]␣=
[PhoneNumber␣-Begin]
␣number␣=␣1234567890
(opt)␣number_type␣=0
[PhoneNumber␣-End]
[Person␣-End]

Dans cet exemple on réalise donc les étapes suivantes :

1. création d'un objet `p` de type `Person`.
2. remplissage de certains champs de cet objet (name, phonenumber...).
3. affichage de `p`.
4. sérialisation de `p`.
5. reconstruction polymorphe de `p` avec la méthode `create` qui attache un objet de type `Person` au pointeur `p2` sur le type `nmsg`.
6. remplissage de cet objet attaché par désérialisation [2]
7. affichage de l'objet attaché.

On voit donc que le message reçu (donc reconstruit) est identique au message envoyé ce qui prouve que les méthodes générées par le générateur de code remplissent bien leur fonction.

2.3.2 Exemple 2

L'exemple 1 nous a permis de comprendre comment on pouvait créer, sérialiser puis désérialiser un objet message en donnant au départ une spécification du message et en utilisant le code généré. Cet exemple 2 suivra les mêmes étapes, mais en essayant d'utiliser un maximum les fonctionnalités du langage de spécification. On écrit donc un fichier de spécification `testGlobalSpec.msg` sur le même principe que `addressbook.msg` et dont voici le contenu :

```
1
2  // Ce fichier a pour but de tester les fonctionnalites
3  // du langage de specification de CERTI
4  // L'ojectif de cette specification est donc d'utiliser un maximum de fonctionnalites
5
6  package test
7  version 1.0
8
9  enum Sexe {
10   HOMME = 0, //la personne est de sexe masculin
11   FEMME = 1, //la personne est de sexe feminin
12 }
13
14 enum PhoneType {
15   MOBILE = 0, //le numero est celui d'un portable
16   HOME = 1, //le numero est celui du domicile
17   WORK = 2, //le numero est celui du lieu de travail
18 }
```

2. C'est le principe de liaison dynamique qui s'applique ici. En effet lors de l'appel p2→deserialize(msgBuf), le compilateur va vérifier que la méthode deserialize existe bien dans le type de p2 (nmsg) puis il va aller chercher la méthode deserialize spécifique de l'objet attaché de type Person.

```
19
20 native MessageBuffer {
21   language CXX [#include "MessageBuffer.hh"]
22 }
23
24 message Factory_msg {
25   required uint32 type
26   required string messageName
27 }
28
29 message PhoneNumber {
30   required string number
31   optional PhoneType number_type
32 }
33
34 message Address {
35   required string country
36   required string city
37   required string street
38   required int32 number
39   required int32 postal_code
40 }
41
42
43 message Person : merge Factory_msg {
44   required Sexe sexe //le sexe de la personne
45   optional int32 age //l'age de la personne
46   required string name //le nom de la personne
47   required int32 id //l'identifiant unique de la personne (numero de carte d'éidentit)
48   required Address address //l'adresse de la personne
49   optional string email //le mail de la personne
50   repeated PhoneNumber phone //les numeros de telephone de la personne
51 }
52
53 message Grades_report_val {
54   required string subject
55   required double grade
56 }
57
58 message Grades_report {
59   repeated Grades_report_val grades_report_values
60 }
61
62 message Education_val {
63   required string year
64   required string formation
65   optional Grades_report grades
66 }
67
68 message Education {
69   repeated Education_val education_values
70 }
71
72
73 message Etudiant : merge Person {
74   optional Education education
75 }
76
77
78 native M_Type {
79   language CXX [typedef uint32_t M_Type;]
```

```
}

factory M_Factory {
  factoryCreator  Factory_msg  create(M_Type)
}
```

On voit que cette spécification est un peu plus compliquée que la précédente dans la mesure où on a des "merge" successifs (`Etudiant` merge `Person` merge `Factory_msg`) et où on a une plus grande variété de types définis et de champs utilisés. Ensuite, comme dans l'exemple 1, on utilise le code généré à partir de cette spécification dans un fichier d'application appelé ici `main.cpp` et dont voici le contenu :

```

#include <iostream>
#include <fstream>

#include "src/testGlobalSpec.hh"
#include "MessageBuffer.hh"

using namespace std;

int main() {

  test::Etudiant et1;
  test::Factory_msg* et2;

  libhla::MessageBuffer msgBuf;
  test::PhoneNumber phonenumber;
  phonenumber.setNumber("1234567890");
  phonenumber.setNumber_type(test::testGlobalSpec::MOBILE);

  et1.setPhoneSize(1);
  et1.setPhone(phonenumber,0);
  et1.setName("John");
  et1.setEmail("John.doe@quisait.fr");
  et1.setId(456);
  et1.setAge(25);
  et1.setSexe(test::testGlobalSpec::HOMME);

  test::Address address;
  address.setCountry("France");
  address.setCity("Toulouse");
  address.setPostal_code(31000);
  address.setStreet("Avenue Edouard Belin");
  address.setNumber(10);

  et1.setAddress(address);

  test::Education education;
  test::Education_val eduval;

  eduval.setYear("2009-2010");
  eduval.setFormation("2A Supaero");

  test::Grades_report grades_report;
  test::Grades_report_val aero;

  aero.setSubject("Aerodynamique");
  aero.setGrade(10.5);
```

```
48
49    grades_report.setGrades_report_valuesSize(1);
50    grades_report.setGrades_report_values(aero,0);
51
52    eduval.setGrades(grades_report);
53
54    education.setEducation_valuesSize(1);
55    education.setEducation_values(eduval,0);
56
57    et1.setEducation(education);
58
59    cout << "Le message envoye : " << endl;
60    cout << " " << endl;
61    et1.show(cout);
62    cout << " " << endl;
63
64    et1.serialize(msgBuf);
65    cout << "taille du message encode en octets = " << msgBuf.size() << endl;
66    cout << " " << endl;
67
68    // Envoi sur le reseau
69
70    et2 = test::M_Factory::create(et1.getType());
71    et2->deserialize(msgBuf);
72
73    cout << "le message recu : " << endl;
74    cout << " " << endl;
75
76    et2->show(cout);
77    delete et2;
78
79
80
81 }
```

Cet exemple 2 suit donc les mêmes grandes étapes que l'exemple 1 à la différence qu'ici la construction de l'objet `Etudiant` est plus compliquée à cause du grand nombre de champs qu'il contient. En dehors de ça le principe est le même. Le résultat de l'exécution est présenté ci-dessous :

```
␣Le␣message␣envoyé␣:

[Etudiant␣-Begin]
[Person␣-Begin]
[Factory_msg␣-Begin]
␣type␣=␣2
␣messageName␣=␣Etudiant
[Factory_msg␣-End]
␣sexe␣=␣0
(opt)␣age␣=25
␣name␣=␣John
␣id␣=␣456
[Address␣-Begin]
```

␣country␣=␣France
␣city␣=␣Toulouse
␣street␣=␣Avenue␣Edouard␣Belin
␣number␣=␣10
␣postal_code␣=␣31000
[Address␣-End]
␣address␣=␣0x805c6e4
(opt)␣email␣=John.doe@quisait.fr
␣␣␣␣phone␣[]␣=
[PhoneNumber␣-Begin]
␣number␣=␣1234567890
(opt)␣number_type␣=0
[PhoneNumber␣-End]
0x805c6e4
[Person␣-End]
[Education␣-Begin]
␣␣␣␣education_values␣[]␣=
[Education_val␣-Begin]
␣year␣=␣2009-2010
␣formation␣=␣2A␣Supaéro
[Grades_report␣-Begin]
␣␣␣␣grades_report_values␣[]␣=
[Grades_report_val␣-Begin]
␣subject␣=␣Aérodynamique
␣grade␣=␣10.5
[Grades_report_val␣-End]
0x805c6e4
[Grades_report␣-End]
(opt)␣grades␣=0x805c6e4
[Education_val␣-End]
0x805c6e4
[Education␣-End]
(opt)␣education␣=0x805c6e4
[Etudiant␣-End]

taille␣du␣message␣encodé␣en␣octets␣=␣207

le␣message␣reçu␣:

[Etudiant␣-Begin]
[Person␣-Begin]
[Factory_msg␣-Begin]

␣type␣=␣2
␣messageName␣=␣Etudiant
[Factory_msg␣-End]
␣sexe␣=␣0
(opt)␣age␣=25
␣name␣=␣John
␣id␣=␣456
[Address␣-Begin]
␣country␣=␣France
␣city␣=␣Toulouse
␣street␣=␣Avenue␣Edouard␣Belin
␣number␣=␣10
␣postal_code␣=␣31000
[Address␣-End]
␣address␣=␣0x805c6e4
(opt)␣email␣=John.doe@quisait.fr
␣␣␣␣phone␣[]␣=
[PhoneNumber␣-Begin]
␣number␣=␣1234567890
(opt)␣number_type␣=0
[PhoneNumber␣-End]
0x805c6e4
[Person␣-End]
[Education␣-Begin]
␣␣␣␣education_values␣[]␣=
[Education_val␣-Begin]
␣year␣=␣2009-2010
␣formation␣=␣2A␣Supaéro
[Grades_report␣-Begin]
␣␣␣␣grades_report_values␣[]␣=
[Grades_report_val␣-Begin]
␣subject␣=␣Aérodynamique
␣grade␣=␣10.5
[Grades_report_val␣-End]
0x805c6e4
[Grades_report␣-End]
(opt)␣grades␣=0x805c6e4
[Education_val␣-End]
0x805c6e4
[Education␣-End]
(opt)␣education␣=0x805c6e4
[Etudiant␣-End]

Là encore on voit que le message reçu et le message envoyé sont identiques. A l'exécution l'affichage des champs avec la méthode `show` est correct. Le seul petit problème est que la méthode `show` telle qu'elle est générée renvoie la référence d'un `ostream` (`std::ostream`& comme type de retour) et du coup cette référence (en l'occurence ici Ox805c6e4) est affichée à l'exécution alors qu'elle ne nous intéresse pas.

2.4 Documentation epydoc du fichier GenMsgAST.py du générateur

On revient dans ce paragraphe sur la notion d'AST vue en 1.2. Dans notre générateur de code, les classes permettant de créer et de vérifier ("check") les propriétés d'un AST se trouvent dans le module GenMsgAST.py. De la même manière qu'en Java avec javadoc, il est possible d'inclure une documentation dans un code python et d'en générer une mise en forme (html, pdf ...) avec un outil appelé **epydoc** (l'équivalent de javadoc). Une documentation epydoc a donc été écrite pour le fichier GenMsgAST.py au cours de ce PIR. On peut trouver une description complète d'epydoc sur [13] . Nous allons ici donner quelques exemples de la syntaxe epydoc. Considérons que l'on doit documenter une classe. D'abord on peut écrire un commentaire global sur la classe (description textuelle) juste après la déclaration du nom de la classe. Ce commentaire se met sous la forme :

"""commentaire"""

Ensuite pour documenter les méthodes de la classe voici la syntaxe :

signature méthode :
"""
Description de la méthode
@param paramètre : description d'un paramètre de la méthode (à répéter pour chaque paramètre)
@type paramètre : type du paramètre (à répéter pour chaque paramètre)
@return : la valeur retournée par la méthode
"""

En ce qui concerne les attributs de la classe, ils ne sont pas obligatoirement déclarés en Python (contrairement à C++ ou Java). Cependant on peut les documenter juste après leur affectation. Voici la syntaxe :

affectation de l'attribut :
"""
@ivar : description de l'attribut
@type : type de l'attribut
"""

Avec ces éléments de syntaxe epydoc on peut documenter toutes les classes de GenMsgAST.py. On peut voir un extrait de la documentation générée au format pdf en annexe A.

Conclusion

Au cours de ce PIR j'ai donc été amené à étudier le fonctionnement de CERTI pour ensuite pouvoir me consacrer au générateur de code. En effet parler du générateur de code sans s'intéresser d'abord à CERTI (et à HLA) serait un non sens dans la mesure où ce générateur est un outil permettant une gestion efficace des messages de CERTI. Il est donc par construction en interaction forte avec CERTI. C'est pour cela notamment que la partie sur le contexte de recherche occupe une place importante dans le rapport. Elle permet de poser les bases nécessaires à la compréhension de la mise en pratique au cours du PIR. Au final ma contribution au cours de ce PIR a principalement été :

- la rédaction de la BNF du langage de spécification des messages.
- l'amélioration du rapport d'erreur du générateur.
- la réalisation d'exemples d'application avec des objets messages extérieurs à CERTI (qui seront intégrés au projet).
- la documentation epydoc du fichier GenMsgAST.py.

D'un point de vue personnel, ce PIR m'a beaucoup apporté tant sur le plan théorique que pratique. Pour moi les principaux enseignements de ce PIR ont été :

- la découverte de HLA et de RTI.
- la découverte de CERTI et son fonctionnement par échange de messages.
- le mode de fonctionnement d'un compilateur.
- l'utilisation du langage Python.
- l'utilisation d'un système de construction logicielle comme **CMake**.
- l'utilisation d'un gestionnaire de version comme **Git**.
- la gestion d'un projet open source comme CERTI.

Ce PIR est aussi ouvert sur d'autres perspectives d'évolution du générateur. Voici quelques perspectives possibles :

♣ Générer du code vérificateur à l'exécution : en effet pour l'instant on peut spécifier des champs `required` (autrement dit obligatoire) dans un message mais rien ne dit que dans la partie écrite à la main par l'utilisateur ces champs seront bien renseignés. Il faudrait pouvoir faire cette vérification.

♣ Générer du code d'observation : il faudrait pouvoir adapter un outil comme *WireShark* (outil pour capturer du trafic réseau et analyser les paquets) aux messages de CERTI. On pourrait ainsi capturer et analyser les paquets des messages de CERTI.

♣ Ecrire le backend Python du générateur : en effet pour l'instant le générateur ne peut pas générer la sortie en langage Python.

♣ Etendre le langage de spécification pour décrire des machines à états dont les transitions seraient provoquées par la réception ou l'envoi des messages (il faut se rappeler que les services rendus par HLA peuvent être décrits par des diagrammes d'état). Une autre possibilité serait de faire une passerelle entre le langage de spécification msg et les langages existants permettant de décrire les machines à état.

Bibliographie

[1] Présentation de RTI HLA. Le 4 avril 2010. http://en.wikipedia.org/wiki/Run-Time_Infrastructure_(simulation).

[2] Documentation des Google Protocol Buffers. Le 6 avril 2010. http://code.google.com/apis/protocolbuffers/docs/overview.html.

[3] Eric NOULARD. CERTI Message specification language : definition, tools, applications and perspectives, 2010.

[4] 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.

[5] Le livre de référence sur la compilation : le "Dragon book". http://fr.wikipedia.org/wiki/Dragon_book.

[6] Tutoriel sur PLY. Le 4 mai 2010. http://www.dabeaz.com/ply/.

[7] Page Wikipedia sur l'analyse syntaxique LR. Le 5 mai 2010. http://fr.wikipedia.org/wiki/Analyse_LR.

[8] Encodage utilisé dans les Google Protocol Buffers. Le 26 mai 2010. http://code.google.com/intl/fr/apis/protocolbuffers/docs/encoding.html.

[9] Tutoriel Python. Le 4 avril 2010. http://www.siteduzero.com/tutoriel-3-223267-apprendre-python.html.

[10] Alex Martelli. *Python in a nutshell*. O'Reilly, 2006.

[11] Page Wikipedia sur la BNF. Le 7 avril 2010. http://fr.wikipedia.org/wiki/Backus_Naur_Form.

[12] Site des Google Protocol Buffers. Le 26 mai 2010. http://code.google.com.

[13] Site sur epydoc. le 10 juin 2010. http://epydoc.sourceforge.net.

A Extrait de la documentation epydoc de GenMsgAST.py