Data Modeling Technologies

User Manual

September 8, 2010

Semantix Information Technologies
ESA ESTEC Labs

Contents

List of Tables

List of Figures

Document Change Record

Issue Revision	Date	Affected Section/ Paragraph/Page	Reason for Change/Brief Description of Change
Internal	2008-07-29	All	Initial release, changes and bugfixes to the original ASSERT deliverable

Executive summary

This document started out as a deliverable of the ASSERT Project. In that project, Semantix Information Technologies was responsible for the development of the Data Modeling Technology; an important ASSERT component which allowed the utilization of many modeling tools and languages during the construction of a complete system.

The chapters that follow provide a complete description of both the *why* and the *how* of the Data Modeling technology. Starting from the requirements of the construction process, the design and hands-on usage of Semantix's toolchain are then presented (for Windows, Linux and the space-oriented embedded platforms with Leon). The content is presented as follows: *a*) First, the overall process that must be followed when utilizing more than one modeling tools is analyzed in detail. *b*) The interaction between the ASN.1 constructs that are used in the interface design and the modeling tools that will implement the subsystems (APLCs) is explained and futher detailed in the relevant appendix. *c*) Usage of the toolchain comes next, with hands-on examples of the tools as they are used from the command line to build the necessary system parts. *d*) The methods of integration with the VM are then analyzed in detail. *e*) The system orchestrator, which completely automates the process of building a system is described. Finally, *f*) the inner workings of the toolchain and the way it was actually designed and implemented are explained. Details of individual backends are also included in the Appendixes.

In conclusion, the work described here is providing a language- and modeling tool-neutral way of communication between subsystems, which utilizes established standards (ASN.1/AADL) and has culminated in demonstratable, automatically-built systems.

Chapter 1

Background

1.1 Introduction

In the context of the ASSERT project, ASN.1 [1] was used as the data modeling language for all the information exchanged between AP-Level containers (APLCs) [2]. The need therefore arose for at least two tasks: (a) mapping of the ASN.1 data models to the modeling languages used during functional modeling of subsystems and (b) weaving of the code generated via the modeling tools with the code generated via the ASN.1 tools.

These two tasks required a special Data Modeling toolchain, that would take into account all the necessary information and automatically generate all the code required. The following sections describe the challenges that had to be met and how they were addressed.

1.2 Bird's eye view - the main idea

There are many advantages to using modeling tools for functional modeling of subsystems. For one, modeling tools offer high-level constructs that abstract away the minute details that are common in low-level languages. The burden of actually representing the desired logic in e.g. C code, falls upon the tool itself, which can provide guarantees[1] of code correctness. Additionally, most modeling tools offer formal verification methods, which are equally important to their certified code generators. For example, a modeling tool can guarantee the correctness of a design in terms of individual components (e.g. if input A is within rangeA, and input B is within rangeB, then outputC will *never* exceed rangeC).

These advantages have driven many organizations to seriously consider (and use) modeling tools for the functional modeling of individual subsystems. After the completion of the functional modeling, however, the modeling tools use custom code generators that materialize the requested functionality in a specific implementation language (e.g. C). To use this functionality, the container's Provided Interfaces need to be accessible from other APLCs (and the Required Interfaces of other APLCs must be callable).

For these calls to take place, information needs to be exchanged amongst the code generated for the "communicating" APLCs. Unfortunately, the generated code is, by definition, quite different amongst different tools; each modeling tool has a very specific way of generating data structures and operational primitives, and mapping these data structures between them is a tedious and very error prone process - since it has to deal with many low level details.

By using ASN.1 as the center of a "star formation" in this communication process, the problem is reduced to mapping the data structures of the exchanged messages between those generated by the modeling tools and those generated by an ASN.1 compiler[2].

[1] SCADE, for example, has been qualified for DO-178B up to level A.

[2] In our case, Semantix's Space Certifiable Compiler, `asn1Scc` (http://www.semantix.gr/asn1scc/)

This process lends itself to a large degree of automation - and this is the task performed by the Data Modeling Toolchain: the automated (and error-proof) generation of the necessary mappings.

1.3 Data Modeling in the ASSERT process

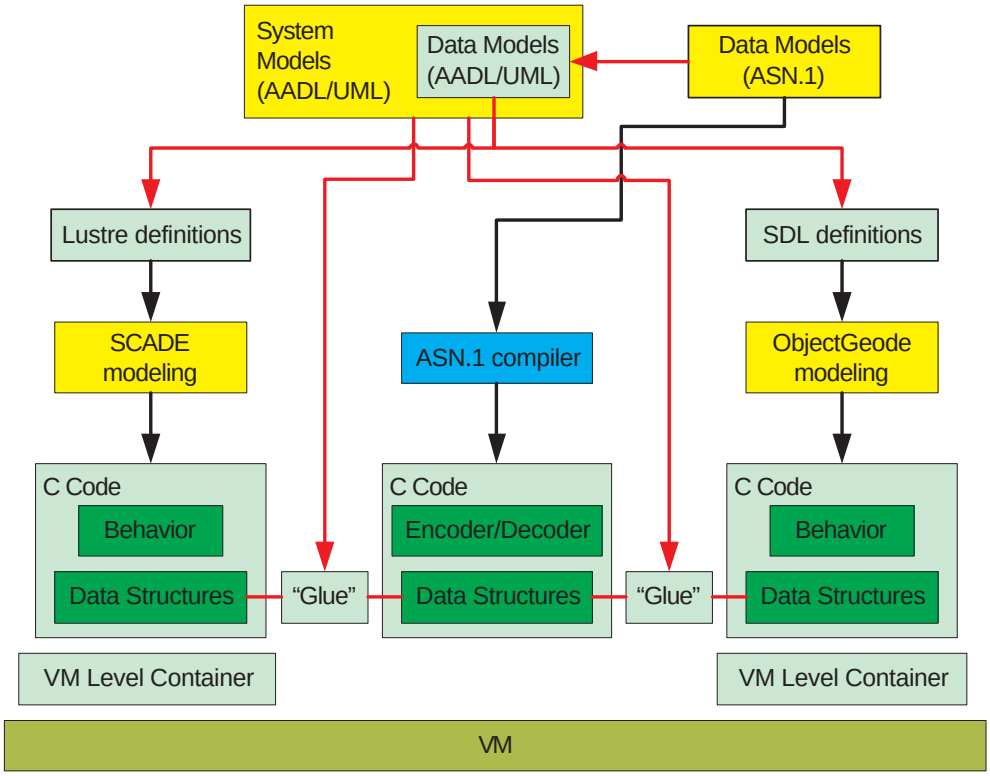

Figure 1.1: Data Modeling with ASN.1

Figure 1.1 displays a high level view of how ASSERT integrates data modeling in the overall system modeling. The yellow blocks depict stages where manual labour is required, and the green ones depict machine-generated entities.

1. The process begins with manual specification of the data models for the messages exchanged between subsystems (AP Level containers). This is where details about types and constraints of message members are specified. To be usable from within the system (AADL [3], UML [4]) specifications, these message definitions are translated into (AADL/UML) data definitions. These definitions are in turn used by the users doing the system modeling - they are referenced during the high level design of the system, in order to create the overall system model (in AADL/LabASSERT or UML/Eclipse).

2. The actual functional modeling of subsystems is next - but before it begins, the exchanged messages' descriptions are read, and semantically equivalent definitions of the data messages are created for each modeling tool's language (e.g. Lustre [5] definitions for SCADE [6] modeling, Simulink [7] definitions for MATLAB/Simulink modeling, etc). This way, the teams building the individual subsystems know that their message representations are semantically equivalent and that no loss of information can occur at subsystem borders.

3. Functional modeling is then done for the individual subsystems. The modeling uses the data definitions as they were generated in step 2.

4. When functional modeling is completed, the modeling tools' code generators are put to use, and C code is generated. Modeling tools generate code in different ways; even though (thanks to step 2) the data structures of the generated code across different modeling tools are carrying semantically equivalent information, the actual code generated cannot interoperate as is; error-prone manual labour is required to "glue" the pieces together. This is the source of many problems[3], which is why ASN.1 is used in ASSERT: by placing it as the center of a star formation amongst all modeling tools, the "glue-ing" can be done automatically.

5. The ASN.1 compiler is used to create encoders and decoders for the messages.

6. "glue" code is then generated, that maps (at runtime) the data from the data structures generated by the modeling tools to/from the data structures generated by the ASN.1 compiler.

7. Code from the ASN.1 compiler, code from the modeling tools and "glue" code are compiled together inside VM [2] Level containers.

8. The VM Level containers are executed on top of VM [8],[9].

The sections that follow elaborate on the steps of this process that relate to data modeling.

1.4 ASN.1 and modeling tools

Each modeling tool has its own idiosyncracies: the way it generates code depends on many things. To begin with, the language used to internally model data and processes has its own limitations in the range of values it can store in its data types; these limitations must be taken under consideration from the beginning - from the creation of the initial ASN.1 data models.

Its not only the modeling tools that impose limitations; the code generated by the ASN.1 compiler also adheres to specific rules: for example, the mapping used for ASN.1 INTEGERs dictates how large a number can be held inside the runtime data structures[4].

Appendix A [8] describes how individual ASN.1 grammar elements must be decorated to represent these limitations - allowing the code generators to use this information and verify that all components fit together well.

[3] Lost satellites being one of them.

[4] If *long* is used as the equivalent C data type, it would mean that under a 32-bit environment INTEGERs can only store numbers from -2147483648 to 2147483647.

Chapter 2

Toolchain usage

The Data Modeling toolchain is reading the AP Level Container information from the system level description in AADL[1] [3], [10]. The extracted information contains both the data types as well as their usage from Provided and Required interfaces.

2.1 Data type definitions

When the ASSERT process reaches the system modeling stage, the overall system design must reference the data definitions that are exchanged between AP-Level containers. These definitions must include references to ASN.1 grammars, as shown in this extract from an actual AADL Dataview:

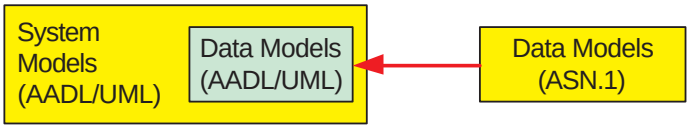

Figure 2.1: From ASN.1 to AADL/UML

```
PACKAGE DataTypes

PUBLIC

DATA POS
PROPERTIES
  -- name of the ASN.1 source file:
  Source_Text => ("PosData.asn1");
  -- Size of a buffer to cover ASN.1 representation:
  -- Real message size is 20; suggested aligned message buffer is...
  Source_Data_Size => 24 B;
  -- name of the corresponding data type in the ASN.1 file:
  Type_Source_Name => "TCLink";
END POS;

END DataTypes;
```

[1] If UML tools were used for the system model, the UML system models are automatically translated into the appropriate AADL declarations.

...and this one, from a UML Dataview:

```
<?xml version="1.0" encoding="UTF-8"?>
<rcm:Model xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:rcm="http://rcm" name="DataTypes" >
  <rootPackage>
    <ownedMember xsi:type="rcm:DataType" name="T_POS"
        Source_Text="PosData.asn1" Source_Data_Size="32"
        Type_Source_Name="TCLink" />
```

The information kept per type includes the specific ASN.1 type (in this extract, `TCLink`) as well as the specific ASN.1 grammar file ("PosData.asn1"). Also included is "glue" specific information (e.g. the size of the message representation in the selected ASN.1 encoding).

These files are of course impossible to write manually, so appropriate tools have been built to that purpose: `asn2aadlPlus` and `asn2uml`.

2.1.1 Creating AADL data definitions

The first step in including an ASN.1 data model in a system level description is the creation of the AADL package that defines the data types. `asn2aadlPlus` is the part of the Data Modeling Toolchain responsible for this. It employs a simple command line interface - it reads an ASN.1 file as input, and generates an AADL file as output:

```
AVALON:~/ASSERT$ asn2aadlPlus
Usage: asn2aadlPlus input.asn [input2.asn] [...] outputDataSpec.aadl
```

A sample usage scenario is depicted below:

```
AVALON:~/ASSERT$ ls -l
total 16
-rw-r--r-- 1 root root 14273 2007-02-28 10:10 messages.asn1
AVALON:~/ASSERT$ asn2aadlPlus messages.asn1 Data.aadl
AVALON:~/ASSERT$ ls -l
total 36
-rw-r--r-- 1 root root 14273 2007-02-28 10:10 messages.asn1
-rw-r--r-- 1 root root 19245 2007-02-28 10:11 Data.aadl
AVALON:~/ASSERT$
AVALON:~/ASSERT$ head -18 Data.aadl
package messages

public

DATA T_CONTROLLER_FILTER_STATE
PROPERTIES
    -- name of the ASN.1 source file:
    Source_Text => ("messages.asn1");
    -- Size of a buffer to cover ASN.1 representation:
    -- Real message size is 286; suggested aligned message buffer is...
    Source_Data_Size => 288 B;
    -- name of the corresponding data type in the source file:
    Type_Source_Name => "T-CONTROLLER-FILTER-STATE";
END T_CONTROLLER_FILTER_STATE;
```

As can be seen above, the utility will create an AADL package that contains data definitions for all ASN.1 types. It will also automatically translate ASN.1 type names to valid AADL identifiers (in the example above, dash (-) is not an allowed part of an identifier, so type `T-CONTROLLER-FILTER-STATE` has been translated as `T_CONTROLLER_FILTER_STATE` in its AADL definition). Finally, it will create the largest possible versions of the ASN.1 messages (using the largest posible values for primitive types and extending SEQUENCE OFs as much as possible) and encode/decode them, to figure out the message sizes and the memory requirements for using them. For this step to work, a valid `gcc` installation must exist, and `gcc` must be accesible from one of the destination directories in the PATH environment variable.

> If your installation environment lacks gcc, you can use an online version of the tool, from: *http://www.semantix.gr/assert*

2.1.2 Creating UML data definitions

System modeling can also be done in UML. As was the case for AADL, an automated way of data type mapping is required - for use during the system modeling. The results of this process are importable in Eclipse, the environment used when UML is used to perform the system modeling.

To that end, a similarly scoped tool was developed, named `asn2uml`:

```
AVALON:~/ASSERT$ ls -l
total 16
-rw-r--r-- 1 root root 14273 2007-02-28 10:10 messages.asn1
AVALON:~/ASSERT$ asn2uml messages.asn1 Data.rcm
AVALON:~/ASSERT$ ls -l
total 36
-rw-r--r-- 1 root root 14273 2007-02-28 10:10 messages.asn1
-rw-r--r-- 1 root root 28734 2007-02-28 10:11 Data.rcm
AVALON:~/ASSERT$
AVALON:~/ASSERT$ head Data.rcm
<?xml version="1.0" encoding="UTF-8"?>
<!-- File generated by asn2uml, DO NOT EDIT! -->
<rcm:Model xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:rcm="http://rcm" name="D_view" >
  <rootPackage>
    <ownedMember xsi:type="rcm:DataType" name="TypeNested"
        Source_Text="D_view.asn" Source_Data_Size="256"
        Type_Source_Name="TypeNested" />
    <ownedMember xsi:type="rcm:DataType" name="T_POS"
        Source_Text="D_view.asn" Source_Data_Size="4096"
        Type_Source_Name="T-POS2" />
```

> If your installation environment lacks gcc, you can use an online version of the tool, from: *http://www.semantix.gr/assert*

2.2 Usage of data types from interfaces

There were many candidate AADL element for use as APLC representatives. The toolchain supports usage of data types from `PROCESSes`, `THREAD`s and `SUBPROGRAM`s (as `DATA PORT`s and `EVENT DATA PORT`s in the first two cases and as `PARAMETER`s in the latter):

```
PACKAGE TestSystem
PUBLIC

SUBPROGRAM TC_Parser
FEATURES
     Id : IN PARAMETER DataTypes::POS;
     Param : OUT PARAMETER DataTypes::POS;
PROPERTIES
     Source_Language => "Lustre";
END TC_Parser;

END TestSystem;
```

The currently supported values for `Source_Language` are depicted in Table 2.1.

`Lustre5` (or, equivalently, `Lustre`)	for SCADE5 [6]
`Lustre6`	for SCADE6 [6]
`OG`	for functional modeling with ObjectGeode [11]
`Simulink`	for functional modeling with MATLAB/Simulink [7]
`C`	for functional modeling with manual C coding (also used from Rhapsody)
`Ada`	for functional modeling with manual Ada coding [12]

Table 2.1: Currently supported modeling tools

2.3 Model mapping and code generation

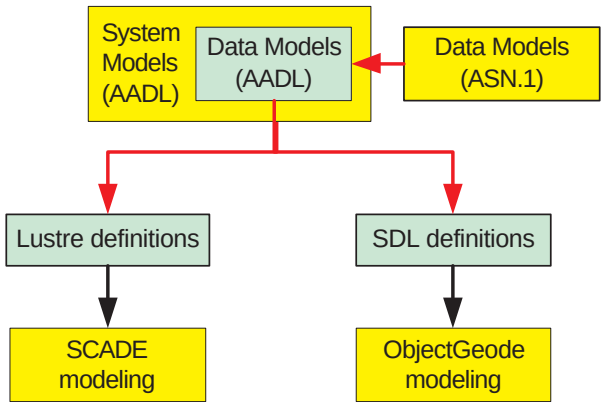

Figure 2.2: *From ASN.1 to modeling tool definitions*

As shown in section 1.3, the ASSERT process begins with ASN.1 data modeling of the messages exchanged between interfaces and the use of `asn2aadlPlus` and/or `asn2uml`. After these tools generate the AADL/UML definitions of the exchanged messages (based on the ASN.1 grammar that described the message contents), the remainder of the system modeling can take place, using the generated data definitions. This system modeling process, will eventually be concluded; at that point functional modeling of the individual building blocks (APLCs) must take place.

The functional modeling of the APLCs requires modeling-tool specific data models. The specification of these models has to be done in the modeling languages of their respective tools. Data models in ASN.1 must therefore be translated to their equivalent representations for each modeling tool (in the modeling language used by each tool - e.g. Lustre [5], SDL [13], etc).

The reason for this step becomes clear if we consider that possibly different teams will be performing the functional modeling of each building block (AP-Level container). This stage can introduce inconsistencies unless the teams working on the two sides of an interface have semantically equivalent representations of the messages exchanged, otherwise unpredictable side effects will take place at the communication borders

during runtime. The teams might be using different modeling technologies, so we need a way to guarantee semantic coherency between the different message representations used by each team.

`asn2dataModel` solves this problem, by reading the ASN.1 grammar that describes the exchanged messages, and generating the appropriate modeling-tool-specific definitions of the messages. It is invoked by each team developing a subsystem, to generate the desired data models in the modeling language that the team uses.

2.3.1 Data Modeling mapper

The following is a hands-on example of `asn2dataModel` usage:

```
AVALON:~/ASSERT$ asn2dataModel
Usage: asn2dataModel <options> input1.asn1 [input2.asn1]...
Where options are:
        -verbose                Increase verbosity of output
        -lexonly                Perform only lexical analysis
        -ignoreINTEGERranges    Don't check INTEGERs for mandatory constraints
        -ignoreREALranges       Don't check REALs for mandatory constraints
        -o dirname              Directory to place generated files
And one of:
        -toAda (for Ada)
        -toC (for C)
        -toOG (for SDL)
        -toSCADE5 (for SCADE5)
        -toSCADE6 (for SCADE6)
        -toSIMULINK (for Simulink)

AVALON:~/ASSERT$ ls -l
total 12
-rw-r--r-- 1 root root 716 2007-02-21 13:29 PosData.asn1

AVALON:~/ASSERT$ asn2dataModel -toSIMULINK PosData.asn1

AVALON:~/ASSERT$ ls -l
total 16
-rw-r--r--  1 root root    716 2008-07-29 13:29 PosData.asn1
-rw-r--r--  1 root root  10289 2008-07-29 16:17 Simulink.PosData.asn1.m

AVALON:~/ASSERT$ cat Simulink.PosData.asn1.m
WorldCoordinate = Simulink.AliasType;
WorldCoordinate.BaseType = 'uint8';
WorldCoordinate.Description = 'range is [0L, 10L]';
...
```

By simply invoking `asn2dataModel` and passing the ASN.1 grammar argument, the data model mapper will create the requested equivalent definitions of the ASN.1 types (in the example above, Simulink types, since `-toSIMULINK` was used).

2.3.2 Code mapper

Having semantically equivalent data type definitions, the teams doing the functional modeling of the AP-Level containers can proceed with their work. They rest assured that their input and output interfaces safely carry all the information required for communication with other AP Level containers.

When however, they reach the point that their modeling tool performs code generation, a different kind of problem appears: the generated code representing the exchanged messages' data structures is completely

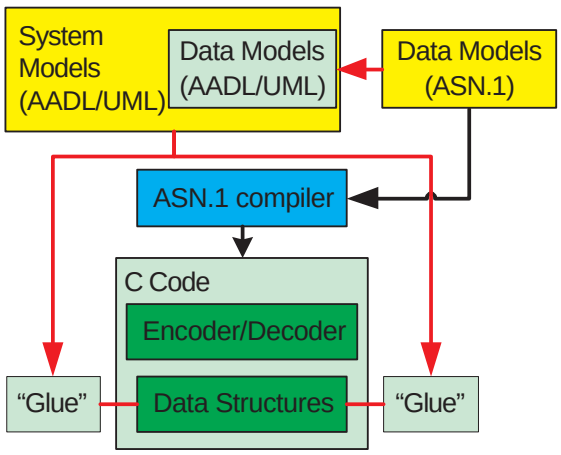

Figure 2.3: From system/data models to runtime "glue"

different, even though the structures themselves carry the same semantic content (`asn2dataModel` guarantees that).

Someone must translate the data between these different codes, and this is the job performed by `aadl2glueC`: it creates the necessary C[2] "glue" code for mapping the data between the data structures generated by `asn1c` and the data structures generated by the modeling tool. Notice that the work performed by this code is done at runtime, so it must be performed with accuracy and precision - and as fast as possible. In the case of SCADE for example, the code generated by `aadl2glueC` is using advanced MACROs provided by ESTEREL to read/write the fields from within the messages to the appropriate SCADE destinations (variables, object instances, etc):

```
AVALON:~/ASSERT$ ls -l
total 16
-rw-r--r-- 1 root root 235 2007-02-21 13:27 Data.aadl
-rw-r--r-- 1 root root 446 2007-02-21 13:32 Lustre5.PosData.asn1.lus
-rw-r--r-- 1 root root 716 2007-02-21 13:29 PosData.asn1
-rw-r--r-- 1 root root 213 2007-02-21 13:29 System.aadl

AVALON:~/ASSERT$ aadl2glueC Data.aadl System.aadl

AVALON:~/ASSERT$ ls -l
total 36
-rw-r--r-- 1 root root   235 2007-02-21 13:27 Data.aadl
-rw-r--r-- 1 root root   446 2007-02-21 13:32 Lustre5.PosData.asn1.lus
-rw-r--r-- 1 root root   716 2007-02-21 13:29 PosData.asn1
-rw-r--r-- 1 root root 12693 2007-02-21 13:41 PosData.asn1.TC_Parser.from.
asn1c.to.Lustre5.c
-rw-r--r-- 1 root root   291 2007-02-21 13:41 PosData.asn1.TC_Parser.from.
asn1c.to.Lustre5.h
-rw-r--r-- 1 root root   213 2007-02-21 13:29 System.aadl

AVALON:~/ASSERT$ cat PosData.asn1.TC_Parser.from.asn1c.to.Lustre5.h
#ifndef __LUSTREPOSDATAASN1_H__
#define __LUSTREPOSDATAASN1_H__
```

[2]In the case of integrating with manually written Ada code, it also creates the equivalent Ada data structures.

```
#include <stdlib.h> /* for size_t */

int Convert_From_ASN1C_To_TCLink_In__TC_Parser__Id(
    void *pBuffer, size_t iBufferSize);

int Convert_From_TCLink_To_ASN1C_In__TC_Parser__Param(
    void *pBuffer, size_t iMaxBufferSize);

...
```

As seen above, this code generator creates "glue" functions that are responsible for mapping the data between the interfaces.

The first, `Convert_From_ASN1C_To_TCLink_In_TC_Parser__Id` decodes an input ASN.1 stream (as provided by the pBuffer and iBufferSize parameters) of ASN.1 type `TC_Link` and stores its values inside the SCADE variables that have been generated for parameter `Id` of subprogram `TC_Parser`. The second one, `Convert_From_TCLink_To_ASN1C_In__TC_Parser__Param`, does the reverse: it reads the data used by SCADE to represent parameter `Param` of subprogram `TC_Parser` and encodes them in an ASN.1 stream of type `TC_Link`. The stream is stored inside the provided buffer arguments.

The implementation details on how the mapping between data stuctures takes place are stored inside `PosData.asn1.TC_Parser.from.asn1c.to.Lustre5.c`.

The generated functions don't stop at providing message translators; in the end, the outside world only needs access to two functions: an *Initialize_PIname* function that calls whatever initialization code is necessary (and must be called exactly once at the beginning of the system runtime) and an *Execute_PIname* which calls the appropriate message translation primitives for the input parameters, executes the functional code for the Provided Interface, and translates the outputs back into ASN.1 streams.

To put it simply, the code mapper creates a complete interface for usage of the APLC's code from ASSERT's VM, utilizing ASN.1 to encode and decode the messages it sends and receives.

2.4 Supported modeling tools

2.4.1 SCADE

SCADE [6] is a completely supported environment. The generated "glue" functions use ESTEREL-provided MACROs that abstract away the internal details of the code generated by SCADE. The VM can then use these generated functions to pass message buffers (i.e. buffers containing ASN.1 encoded messages) back and forth between itself and the SCADE generated C code [14]. Section 2.3.2 demonstrates the generation of glue functions and their call signatures for a specific SCADE example.

More information about the SCADE mappers are in Chapter 11.

2.4.2 MATLAB/Simulink

MATLAB/Simulink [7] is a completely supported environment. The generated "glue" functions target the Real-Time Workshop generated code, whose code generator is by far the best code generator in the MATLAB suite. The RTW generated code is very close to what a human coder would write, so the reverse-engineering necessary for creating the "glue" generators was straightforward - and easier than most other tools... Section 3.1.1 demonstrates the generation of glue functions and their call signatures for a specific MATLAB/Simulink example.

More information about the MATLAB/Simulink mappers are in Chapter 11.

2.4.3 ObjectGeode

ObjectGeode [11] is completely supported. To begin with, ObjectGeode offers the easiest possible mapping of data types: SDL [13] directly supports ASN.1 data types, so `asn2dataModel` is almost a no-op for this modeling tool. `aadl2glueC` on the other hand, has the normal workload for SDL (just as it did for SCADE, Rhapsody, Ada, etc). It generates a set of MACROs that are used by ObjectGeode-generated code, which allow for

- *Declarations and definitions of transfer buffers for each ASN.1 type*: These are used by the ObjectGeode code (`hpostdef.h`) to statically reserve the necessary space for the exchanged messages (at compile time).
- *ASN.1 Decoders/mappers and readers/ASN.1 encoders*: These are used by the message-sending / signal-receiving parts of the ObjectGeode code (`vm_if.c`, `hpostdef.h`) to perform the ASN.1 marshalling as well as the data mapping between the data structures of ObjectGeode and those generated by the ASN.1 compiler.

Using these "glue" layers, the ObjectGeode generated code is effortlessly communicating with the other APLCs over the VM.

More information about the ObjectGeode mappers are in Chapter 11.

2.4.4 Manual code in Ada

Ada [12] is also completely supported. `aadl2glueC` creates

- the necessary C code that uses the ASN.1 decoders and encoders to read and write the relevant ASN.1 messages,
- as well as the Ada code which gets the "translated" data through "bridge" data structures (in automatically generated Ada code); these "bridges" contain exactly the same information, in the language constructs expected by Ada developers.

This simply means that the Ada developer doing the functional implementation has a set of Ada data structures at his disposal, which are clearly mirrored into/from ASN.1 messages through the generated "bridge" functions (one per interface).

Table 2.2 describes the mapping from ASN.1 entities to the Ada "bridge" variables.

INTEGER	Integer
REAL	Long_Float
BOOLEAN	Integer
OCTET STRING	array (0..x) of Character
	plus an Integer xyz_len variable
ENUMERATED	Integer
SEQUENCE OF	array (0..x) of ...
	plus an Integer xyz_len variable
SEQUENCE	record ... end record

Table 2.2: From ASN.1 to Ada

2.4.5 Rhapsody / manual code in C/C++

These environments are also completely supported. `aadl2glueC` creates the necessary C code that reads and writes the relevant ASN.1 messages, utilizing asn1Scc's [15] "bridge" data structures in C that contain the semantically equivalent information. This considerably lessens the effort required to access the message

marshalling routines from within Rhapsody (or manually written C/C++ code), as the user has access to simple standalone[3] C data structures for all the messages exchanged.

INTEGER	long
REAL	double
BOOLEAN	int
OCTET STRING	char[x] plus an int xyz_len variable
ENUMERATED	long
SEQUENCE OF	typename[x] plus an int xyz_len variable
SEQUENCE	struct ...

Table 2.3: From ASN.1 to C

Table 2.3 describes the mapping from ASN.1 entities to the C "bridge" variables.

[3] i.e. not containing pointers to other entities.

Chapter 3

VM integration

The ASSERT Virtual Machine [8] is a key player in the inner workings of ASSERT. It is orchestrating the operation of the system as a whole, taking into account a number of parameters, such as timing and scheduling information.

As far as data modeling is concerned, the VM has a simple job to do: it acts - at runtime - as a "router" of messages between the orchestrated functional entities. To that end, the automatically generated runtime "bridges" (built from `aadl2glueC`) are put to use; decoding the incoming parameters (which are ASN.1 messages) and encoding the outgoing ones.

The VM itself is also automatically generated (from the system-level description). The following sections describe how the data modeling "bridges" are invoked at runtime from the automatically generated VM, and how this task breaks down for the two kinds of interfaces that exist in ASSERT: Synchronous and Asynchronous ones.

3.1 Synchronous modeling - how it works

Synchronous interfaces are very simple: they represent standalone functions, with (possibly) a set of input and (possibly) a set of output parameters. They are performing their work and generating their output without depending on any external functionality - that is, they don't call any other interfaces to do their work.

Assuming, for example that ...

- a system is formed from two subsystems (APLCs), A and B
- subsystem A uses a synchronous provided interface (named PI1) which is provided by subsystem B
- the PI1 interface expects one input parameter, described with a specific ASN.1 message (named M1)

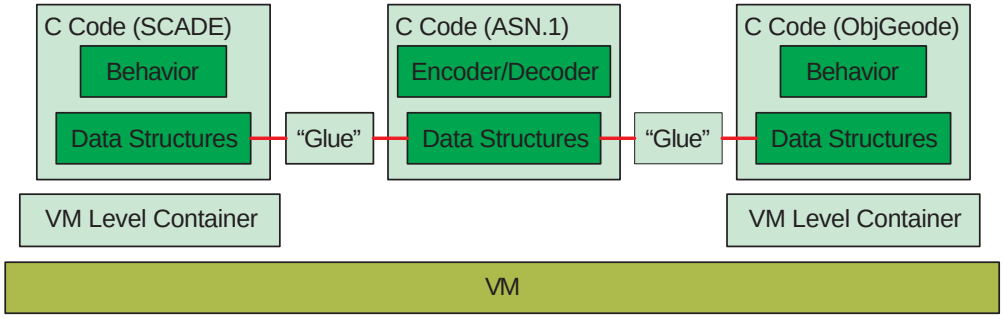

Figure 3.1: The ASSERT VM integrates everything

- subsystem B is functionally implemented through a MATLAB/Simulink model

The following happen during the VM's code generation stage ...

- the VM code generator knows - by parsing the system description in AADL - that subsystem B is functionally implemented through a Simulink model (`Source_Language => "Simulink";`).
- the VM code generator also knows that `aadl2glueC` has been called prior to its own code generation stages. During its invocation, `aadl2glueC` has generated the required "bridge" functions, which allow the VM to communicate with subsystem B's PI1 interface.

The VM code generator is therefore able to create code that performs the appropriate calls (at runtime) to call subsystem B's PI1 provided interface. In fact, based on the work done from `aadl2glueC`, the VM only has to perform a simple function call, to an appropriately named bridge function for the PI. This call is easy to introduce in the VM code, since the invocation of any PI (and thus, our example PI1 from subsystem A) is "routed" via a VM callback function.

The VM therefore, in the context of this callback, can ...

- get hold of the ASN.1 encoded data that describe the input parameter (as it is passed from subsystem A, that is, an encoded stream of ASN.1 type M1).
- the encoded data are accessible through a (pointer, size) pair, or more precisely, a pair of (`void *`, `size_t`). This pair provides access to the message data and the data size, respectively.
- The VM then calls the appropriate "bridge" function that was created by `aadl2glueC` for subsystem B's PI1 provided interface, passing in this pair...

This "bridge" function then (prepared from `aadl2glueC`) performs three tasks:

- it decodes the input parameter (of ASN.1 type M1) and places the decoded data inside the appropriate "bridge" Simulink variables.
- it invokes the PI1 entry function (the function generated by the Real Time Workshop component of MATLAB/Simulink) ; the functional code of PI1 then reads the M1 data from the bridge variables and operates based on them.
- if PI1 has any output parameters, their contents are read from the respective Real Time Workshop generated variables, and are encoded back into the appropriate output ASN.1 streams.

Notice that during this process, there is no modeling-tool-specific "agenda". The exact same steps are taken for all synchronous modeling tools (SCADE, MATLAB/Simulink, etc): the data modeling layer generates the appropriate "bridge" functions, and the VM only needs to invoke them, nothing else.

The names of these "bridge" functions follow simple naming conventions, and can be broken down in two layers

3.1.1 High-level bridges - VM interface

Each provided interface is equipped with an `FV_Name` AADL property. For example...

```
SUBPROGRAM mysimulink
FEATURES
        my_in:IN PARAMETER DataView::T_FOR_SIMULINK_IN {encoding=>UPER;};
        my_out:OUT PARAMETER DataView::T_FOR_SIMULINK_OUT {encoding=>UPER;};
END mysimulink;

SUBPROGRAM IMPLEMENTATION mysimulink.Simulink
PROPERTIES
        FV_Name => "mysimulink_fv";
        Source_Language => Simulink;
END mysimulink.Simulink;
```

Two "bridge" functions are generated per interface from `aadl2glueC`:

```
/* Initialize the PI */
void init_FVName();

/* Invoke the PI */
void FVName_SubPrgName(
    void *pInBuffer1, size_t iInBufferSize1,
    void *pInBuffer2, size_t iInBufferSize2,
    ...,
    void *pOutBuffer1, size_t *pOutBufferSize1,
    void *pOutBuffer2, size_t *pOutBufferSize2,
    ...);
```

`SubPrgName` is the name of the AADL SUBPROGRAM that is used for the implementation of the interface, e.g. `mysimulink` in this example. `aadl2glueC` would therefore create the following functions for this example:

```
void init_mysimulink_fv();

void mysimulink_fv_mysimulink(
    void *pmy_in, size_t size_my_in, void *pmy_out, size_t *pSize_my_out);
```

The VM code generator only needs to call these two functions per interface, to get access to any interface's functionality. The "init_FVName" must of course be called only once, to initialize the interface; the VM code calls it at startup.

3.1.2 Low-level bridges - internal interface between VM and data mappers

The high-level "bridges" use the following low-level functions to do their work:

- In the *incoming* direction, i.e. when the code needs the "glue" layer to decode the message and place its data inside the appropriate modeling-tool generated variables, the bridge function doing the work is:

  ```
  int
  Convert_From_uper_To_MsgType_In_SubPrgName_SbPrgImpl_ParamName(
      void *pBuffer,
      size_t iBufferSize);
  ```

  ...where
  - SubPrgName is the name of the subprogram (in the previous example. `mysimulink`)
  - SbPrgImpl is the name of the subprogram implementation (`Simulink`)
  - ParamName is the name of the parameter (`my_in`)

  These three entities unmistakeably identify the parameter in question. Their names are taken verbatimly from the AADL file, and if they contain a character not supported by C, '_' is used instead in its place.
  For our example above, the following function is generated:

  ```
  int
  Convert_From_uper_To_T_FOR_SIMULINK_IN_In_mysimulink_Simulink_my_in(
      void *pBuffer, size_t iBufferSize);
  ```

  Notice that this kind of bridge functions are called before the call to the actual PI implementation function - since the actual implementation expects its inputs to be set prior to its execution.

- The "real work" functionality of the PI is accessible through a function named as...

```
void Execute_SubPrgName_SbPrgImpl();
```

  This bridge function knows exactly how to call the appropriate functional code generated by each modeling tool. In the case of SCADE for example, it will issue a call to the `AADL2SCADE_PERFORM` macro generated by the ESTEREL code generator.
  In our example, this function is called...

```
void Execute_mysimulink_Simulink();
```

- In the *outgoing* direction, i.e. *after* the functional code completes, when the "glue" layer needs to read the data from the appropriate modeling-tool generated variables and encode their content as output messages, the corresponding bridge function is called...

```
int
Convert_From_MsgType_To_uper_In_SubPrgName_SbPrgImpl_ParamName(
    void *pBuffer,
    size_t iMaxBufferSize);
```

  The buffer passed-in must be able to hold the maximum configuration of the message. The VM code generator knows this maximum size, since it has parsed the AADL Dataview, which is automatically generated from `asn2aadlPlus` - and thus, has access to the maximum message sizes. Upon return of this function, the `pBuffer` is filled with the output message data, and the actual message size (always less than `iMaxBufferSize`) is returned.
  In our example, this function is called...

```
int
Convert_From_T_FOR_SIMULINK_OUT_To_uper_In_mysimulink_Simulink_my_out(
    void *pBuffer, size_t iMaxBufferSize);
```

These low-level support functions are called by the high-level ones, the ones that VM calls. From the VM's point of view, the process is completely automated: the VM ends up accessing functionality written in any synchronous modeling tool, regardless of parameters, by simply calling two functions... Access to the exchanged message details is not only unnecessary, it would also be hazardous, since it would "tie" the VM to message-specific details. Via the Data Modeling Toolchain, the VM can route any message to any interface, handling the message data as simple, opaque, sequences of bytes.

3.2 Asynchronous modeling - how it works

The approach described for synchronous tools, cannot cope with asynchronous modeling. In these tools, it is the subsystem itself that decides *when* a call will actually take place - the VM is a *callee* in this case, not a *caller*. Not only that, but the asynchronous call can't afford to wait for the call to return (and therefore, output parameters are meaningless). The "bridge" mechanisms must therefore be different for asynchronous modeling tools.

3.2.1 Provided Interfaces (PIs)

The case of PIs is similar in context to synchronous modeling tools's code: the available functionality that corresponds to the PI must be invoked; therefore, all that is required is an *entry function* into the PI, that takes the required parameters as *function arguments*.

There is an important difference, however, when comparing asynchronous code to synchronous code: OUT parameters are not allowed in PIs provided by APLCs that are modeled in asynchronous tools. By definition,

OUT parameters are supposed to be "ready" at the end of the PI invocation; in contrast, asynchronous tools give no such guarantee - there is no way to know *when* the output (if any) is ready, except by waiting for the state machine inside them to respond by generating a message (and possibly, switching current state).

This is where synchronous and asynchronous tools diverge - and hence, the corresponding "glue" also diverges: in asynchronous tools, the responses are not OUT parameters; the responses are *callbacks* into the VM, which are in fact invocations of RIs. This is how asynchronous tools provide answers; they call back RIs (through the VM), and provide their output as the input parameters to these RIs.

Here is an example of the relevant AADL section for an asynchronous PI:

```
SUBPROGRAM tcommand
FEATURES
        hltc:IN PARAMETER DataView::T_HLTC_PLUS {encoding=>UPER;};
END tcommand;

SUBPROGRAM IMPLEMENTATION tcommand.SDL
PROPERTIES
        FV_Name => "basic_fv";
        Source_Language => OG; -- Modeled in ObjectGeode
END tcommand.SDL;
```

When the VM needs to invoke a PI that is implemented with an asynchronous modeling tool, it must invoke a bridge function, named...

```
void
FV_Name_SubPrgName(
    char *pInBuffer1, int iBufferSize1,
    char *pInBuffer2, int iBufferSize2,
    ...);
```

As with the synchronous tools, `SubPrgName` is the name of the SUBPROGRAM, and `FV_Name` is the additional AADL property set in the `SUBPROGRAM IMPLEMENTATION` section. Any non alpha-numeric characters in these names are replaced by '_'. Depending on the number of parameters expected by the PI, the procedure can have 0, 2, 4, or generally, 2xN arguments. For each PI input parameter, two arguments appear in the procedure: a pointer to the ASN.1 encoded data of the parameter, and a length (how many bytes the pointer points to).

These bridge functions are generated inside `vm_if.c` (which is automatically generated for each asynchronous PI, during the "glue" generation process).

3.2.2 Required Interfaces (RIs)

As said above, RIs are called by the state machines; the VM (or, more specifically, the appropriate VMLC) is in this case the *callee*, not the caller. For this to work, the code generation process of the VM must provide entry points for the RIs, named

```
void
vm_FV_Name_SubPrgName(
    char *pInBuffer1, int iBufferSize1,
    char *pInBuffer2, int iBufferSize2,
    ...);
```

Just as with PIs, the parameters of the RI are translated in pairs of (pointer, length) arguments in this callback. The VM will get the specific values of the invocation parameters as encoded ASN.1 streams, pointed to by these pairs. It must delegate the call to the appropriate PI, passing the parameters exactly as it received them.

As an example, the code generated from ObjectGeode calls these functions through special MACROs, defined inside `hpostdef.h`. The appropriate function declarations are placed during the "glue" generation process inside file `hpredef.h` (see the details of these in Chapter 11).

Chapter 4

Automation

Just like any other process, the ASSERT process can be viewed as a "black box", operating on a number of inputs and producing a single output: the compiled, executable parts of the system.

These are the inputs required for the actual compilation of the system:

- The ASN.1 grammar describing the messages exchanged between APLCs (over Provided and Required Interfaces)
- The overall system model's Interface view (in AADL [3])
- The overall system model's Concurrency view (in AADL [3])
- The functional models for the APLCs of this system (SCADE, MATLAB/Simulink, ObjectGeode, etc)

Due to the complex build process required, a central "orchestrator" is necessary: automated logic that takes care of all the details of performing each individual step in the ASSERT process. The Data Modeling Toolchain includes the `assert-builder`...

```
Usage: assert-builder.py <options>
Where <options> are:

-f, --fast
        Skip waiting for ENTER between stages

-n, --nokalva
        Use OSS Nokalva for ASN.1 compiler (if missing, asn1Scc is used)

-o, --output <outputDir>
        Directory with generated sources and code

-a, --asn <asn1Grammar.asn>
        ASN.1 grammar with the messages sent between subsystems

-i, --interfaceView <i_view.aadl>
        The interface view in AADL

-c, --concurrencyView <c_view.aadl>
        The concurrency view in AADL (as generated by the VT)

-S, --subSCADE name:<zipFile>
```

```
        a zip file with the SCADE generated C code for a subsystem
        with the AADL name of the subsystem before the ':'

-M, --subSIMULINK name:<zipFile>
        a zip file with the SIMULINK/ERT generated C code for a subsystem
        with the AADL name of the subsystem before the ':'

-C, --subC name:<zipFile>
        a zip file with the C code for a subsystem
        with the AADL name of the subsystem before the ':'

-A, --subAda name:<zipFile>
        a zip file with the Ada code for a subsystem
        with the AADL name of the subsystem before the ':'

-G, --subOG name:file1.pr<,file2.pr,...>
        ObjectGeode PR files for a subsystem
        with the AADL name of the subsystem before the ':'
```

For example, when the script is used to create ASSERT's PFS Pilot Project, it is invoked like this:

```
bash$ assert-builder.py \
        --fast \
        -o output/ \
        -a newPackages/D_view.asn1 \
        -i newPackages/MSU_Thread.aadl \
        -c newPackages/MSU_Thread_conc.aadl \
        -G newPackages/MSU_Threads_Basic.pr \
        -G newPackages/MSU_Threads_Control.pr \
        -G newPackages/MSU_Threads_Cyclic.pr \
        -S newPackages/ScadeBlock.zip
```

The "orchestrator" completely takes care of the following:

- Translating the ASN.1 grammar into AADL DATA definitions (`asn2aadlPlus`)
- Creating the modeling tool-specific type definitions (`asn2dataModel`)
- Creating the appropriate "glue" for each APLC, after parsing the system model (`aadl2glueC`)
- Creating the ASN.1 encoders/decoders (invoking `asn1Scc` or Nokalva to generate the source code, via the ASN.1 Grammar)
- Unpacking the .zip files containing the code for the subsystems and invoking the directly supported modeling tool code generators (ObjectGeode)
- Invoking Ocarina to generate the VM
- Compiling (via `gnatmake`) the VM sources (Ada)
- Compiling (via `gcc`) the ASN.1 encoders/decoders
- Compiling (via `gcc`) the source code for the SCADE/MATLAB/etc synchronous models
- Compiling (via `gcc`) the source code for the ObjectGeode/C/Ada/etc asynchronous models
- Auto-resolving the (possibly) conflicting symbols that might appear in the object files
- Linking it all together into a working executable (or executables, if the deployment so dictates)

Needless to say, this automation immensely helps the - otherwise very tedious and error prone - integration phase of ASSERT projects. Also note that parts of this process are directly available through online gateways [10].

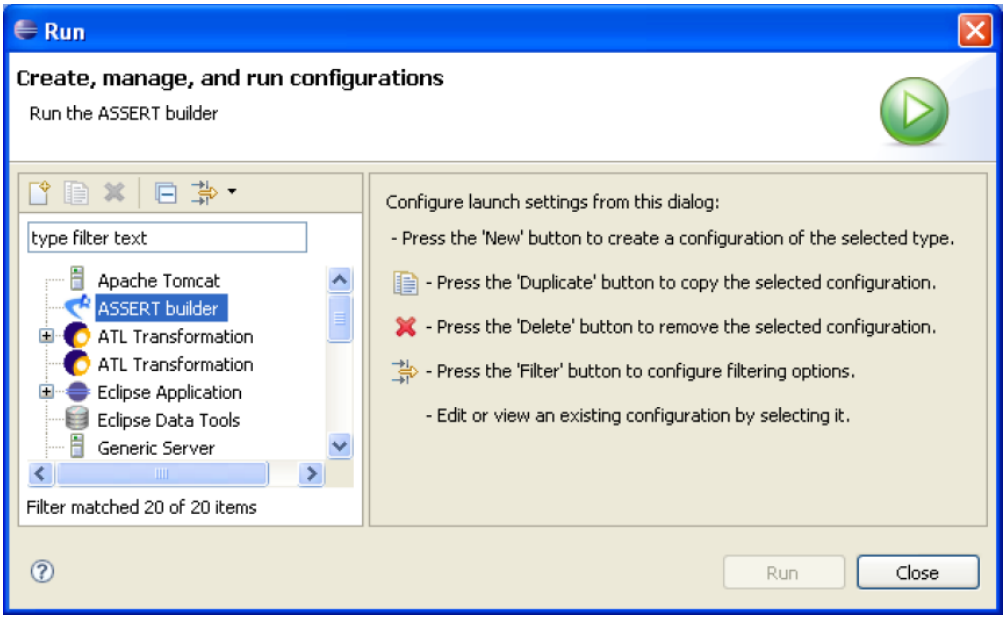

Figure 4.1: Integration in the UML modeling tool

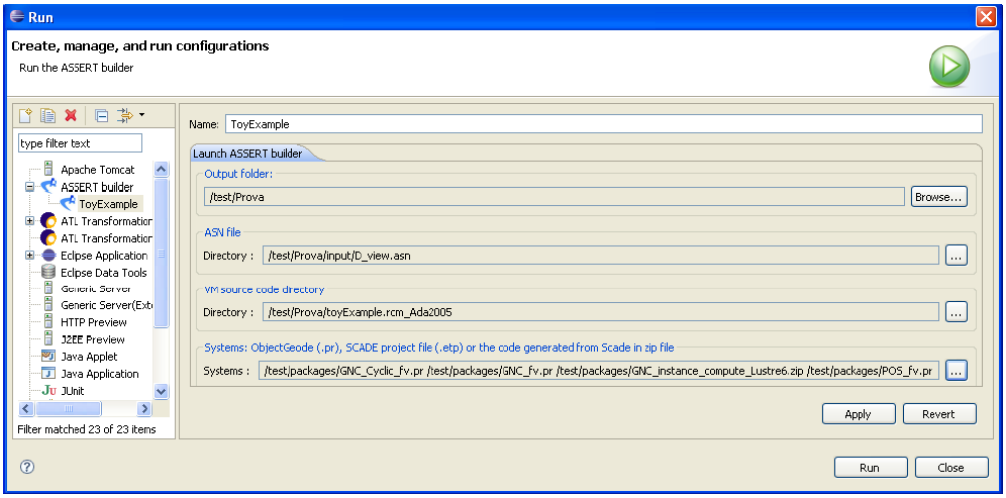

Figure 4.2: Selecting the inputs

Chapter 5

Toolchain internals

5.1 Design

The Data Modeling Toolchain can be basically summarized as a (potentially large) set of code generators. Code generators, in turn, are known to usually suffer from a serious problem: rigidity. Since they handle the issue of automatic code generation at a meta-level (at the level of "notions" that drive the code genesis) it is difficult to change their behaviour and extend them. In fact, the whole process is usually so delicate that in most cases only the creator of the code generator knows enough to modify it - even within the walls of the company building it, the people "allowed" to work on it are usually a select *elite*.

This could not be tolerated in ASSERT - in fact, the project leader (ESA) immediately highlighted the importance of "tweakability": the code generators were not only supposed to cover the functional requirements of the project, they had to do this in an extendable and modular way. In fact, the desire was expressed for a scaffolding that would allow the *end user* - and not just the developer - of the toolchain to easily modify existing code generators or create new ones.

To accomodate this, a domain specific language was initially adopted. This language was a simple extension of the ubiquitous Python scripting language. It allowed the developer of the code generators to access information on all the entities stored in the AADL and ASN.1 parse trees, and use any Python constructs to process them and generate output (if/elif/else, try/except/, for loops, etc).

As the code generators for specific backends (SCADE, ObjectGeode, etc) were being written, new needs arose: it became clear that static code analysis as well as statement coverage would provide significant help in identifying errors and hidden traps. These however could only be executed in the project's alloted timeframe if the domain specific language stopped being 'almost Python' and became true Python (which comes with ready made versions of these tools included).

5.2 Mappers and Code generators: Python modules

The architecture selected for the code generators can be seen in Figure 5.1. It is basically split in (a) the frontend parsers (AADL and ASN.1) and (b) backend generators (for SCADE, ObjectGeode, etc).

`asn2dataModel` directly parses the input ASN.1 grammar. `aadl2glueC` starts with parsing of the AADL system description. The AADL parser is fed in turn with each one of the input AADL files, and Data Modeling specific information is extracted. The AADL Data view, in particular, has been generated from `asn2aadlPlus`, and includes references to the ASN.1 grammars used. Each one of these ASN.1 grammar files is then passed to the ASN.1 parser (as is done for `asn2dataModel`.

At the end of these stages, two Abstract Syntax Trees (ASTs) contain the overall system information: The system AST and the data types AST. The system AST contains all the Data Modeling related information which is stored in the AADL files, while the data types AST contains all the ASN.1 specific information stored in the ASN.1 file referenced by the Data View.

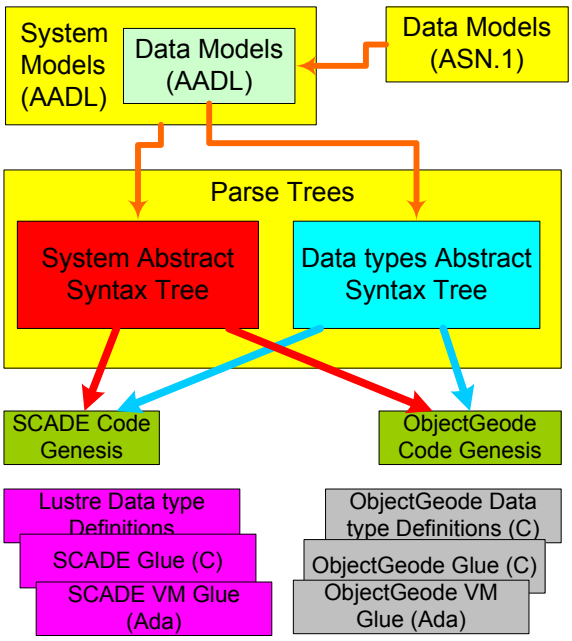

Figure 5.1: Architecture of Code Generators

What happens next (labeled in in Figure 5.1 as "Code Genesis") can be described in pseudocode like this:

```
for each one of the AP Level Containers that need "glue"
   identify the implementation language they are made of
   (e.g. Lustre/SCADE, SDL/ObjectGeode, manual C/Ada)
   Load the appropriate backend module for the language
   for each one of their input and output parameters
      call the loaded module's function
      for handling the specific parameter's ASN.1 type
      (e.g. OnBasic, OnSequence, OnSequenceOf, etc)
```

This allows for a completely modular architecture:

- The backend module that will be loaded for a specific AP Level Container follows a simple naming convention: if the implementation language (`Source_Language`) is `XYZ`, then the module must be named either `xyz_A_mapper.py` (for mapping to tool-specific data types) or `xyz_B_mapper.py` (for the specification of the "glue" code genesis). The toolchain currently includes support for Lustre5, Lustre6+, MATLAB/Simulink, ObjectGeode, Ada and C mappers, in the form of ...
  - `lustre5_A_mapper.py` and `lustre5_B_mapper.py` (for SCADE 5)
  - `lustre6_A_mapper.py` and `lustre6_B_mapper.py` (for SCADE 6)
  - `simulink_A_mapper.py` and `simulink_B_mapper.py` (for MATLAB/Simulink)
  - `og_A_mapper.py` and `og_B_mapper.py` (for ObjectGeode)
  - `c_A_mapper.py` and `c_B_mapper.py` (for manual C coding, or Rhapsody/C++)
  - `ada_A_mapper.py` and `ada_B_mapper.py` (for manual Ada coding)
- Each one of the backends includes a set of functions that describe what actions to take for each kind of ASN.1 construct:

  - `OnBasic` handles all the basic ASN.1 types: INTEGER, REAL, BOOLEAN and OCTET STRING and its variants.
  - `OnEnumerated` handles the ASN.1 ENUMERATED types
  - `OnSequence` handles the ASN.1 SEQUENCEs (i.e. the *structures* of information)
  - `OnSequenceOf` handles the ASN.1 SEQUENCE OFs (i.e. the *arrays* of a type)
  - `OnChoice` handles the ASN.1 CHOICE OFs (i.e. the *polymophic* types)
  - `OnSet` handles the ASN.1 SETs (i.e. the *unordered structures* of information)
  - `OnSetOf` handles the ASN.1 SET OFs (i.e. the *sets* of a type)
- The function arguments provide access to the ASTs, so that the developer of the backend can use whatever information he/she needs from the type (e.g. RANGE information on INTEGERs, options available in ENUMERATEDs, etc).
- `OnStartup` and `OnShutdown` functions act as "constructors" and "destructors" (e.g. open the output files, etc). They are called for initialization and final work.

Notice that each parameter is of a specific ASN.1 type, which explains why all the OnXYZ functions take a `nodeTypename` string argument, as well as a `node` argument that provides them with access to the ASN.1 AST. Additional function arguments are also provided to offer "shortcuts" and additional information through the ASTs (see below).

The "Hello world" of type A backends (i.e. mappers to data types) is therefore this:

```
def Version():
    print "HelloWorld backend of type A (asn2dataModel), Version 1.0"

def OnStartup(modelingLanguage, asnFile, outputDir):
    print "Starting up processing..."

def OnBasic(nodeTypename, node, leafTypeDict):
    print nodeTypename

def OnSequence(nodeTypename, node, leafTypeDict):
...

def OnShutdown():
    print "Ending up processing..."
```

...while the "Hello world" of type B backends (i.e. "glue" generators) is this:

```
def Version():
    print "HelloWorld backend B (aadl2glueC), Version 1.0"

def OnStartup(modelingLanguage, asnFile, subProgram, sPrImpl, outputDir):
    print "Starting up processing..."

def OnBasic(nodeTypename, node, subProgram, sPrImpl, param, \
        leafTypeDict, names):
    print nodeTypename
    ...

def OnShutdown(modelingLanguage, asnFile, subProgram, \
        subProgramImplementation):
    print "Ending up processing..."
```

All mappers and "glue" generators of ASSERT are implemented under this scheme. Since the backends are real Python modules, they are amenable to static analysis (through PyChecker[1]) and equally important, statement coverage[2].

Also note, that Appendix B [9] includes details on what kind of information is carried by the ASN.1 Abstract Syntax Tree (the `node` parameters in the callbacks shown above).

[1] http://pychecker.sourceforge.net/

[2] By using the *-m trace* option of Python.

Chapter 6

Getting the toolchain

The latest version of the toolchain and its documentation is always available at:
http://www.semantix.gr/assert/.

Chapter 7

Conclusion

The Data Modeling Toolchain, as evidenced in its use by both the AADL and the UML tracks in the ASSERT project, was hugely successful in solving the interworking between the ASSERT VM and the code generated by the modeling tools. By using ASN.1 definitions as the "data contracts", it succeeded in establishing the necessary guarantees that no information will ever get lost in the otherwise tedious and error-prone process of translating data (at runtime) between code generated by different modeling tools. It thus manages to completely eliminate a very troubling - and rather wide - category of errors that relate to the marshalling of data; and more importantly, it enforces this at DESIGN time, not at runtime.

Equally important is how this was accomplished: not in a rigid, black-box technology; the mapping backends allow anyone who spends the time to read the relevant appendixes, to be able to create any backend for his/her own modeling tool, irrespective of the specific details of the mapping. The architecture is thus completely flexible and expandable. Semantix is already working in tandem with ESA, in creating backends for additional modeling tools (beyond those supported in ASSERT).

Finally, the implementation did not just stop in simple "proof-only" examples; it became field tested, when it was applied in two multi-modeling tool scenarios, with the MA3S/PFS Pilot Project being the most prominent one. The resulting binaries were successfully downloaded and executed on the real embedded processors that ESA is using (LEON [16]), proving in a conclusive manner that ASSERT is not just a set of theoretical concepts; it's a real implementation of state of the art concepts, in the multi-modeling tool domain (for both monolithic and distributed platforms).

Chapter 8

Appendix A - ASN.1 usage guidelines

This appendix describes how individual ASN.1 grammar elements must be decorated to represent the limitations that modeling tools impose on the ASSERT process. These decorations, in turn, allow the code generators to use the information and verify that all components fit together well (in the code generating stages that will follow).

8.1 BOOLEANs

Supported as per the standard - use something like this ...

```
MyBooleanType ::= BOOLEAN
```

... to describe a standalone boolean type, or ...

```
fieldName BOOLEAN
```

... to describe a member of a SEQUENCE.

8.2 INTEGERs

`INTEGER` ranges cannot be open: for example, as members in a SEQUENCE, ...

```
value INTEGER
```

... or ...

```
value INTEGER (0..MAX)
```

are not allowed, since `MAX` can have ASN.1 tool-specific semantics. To that end, the only `INTEGER` statements allowed are the ones with explicit ranges declared, i.e.

```
value INTEGER (123..789)
```

This is also necessary for another reason: the code generated for `INTEGER`s by an ASN.1 compiler (in our case, `asn1Scc`[1]) is able to carry specific ranges of `INTEGER`s. In our case, the target C type is `long`. Depending on the implementation platform, `long` can carry anywhere from 32 to 128 bits; the Data Modeling Toolchain knowns the target platform, and can therefore check whether the range specified in the ASN.1 grammar is covered. If it is not, a fatal error will be displayed during the translation, ...

[1] http://www.semantix.gr/asn1scc/

```
INTEGER (in ...) must have a range constraint inside
ASN.1, or else we might lose accuracy during runtime!
```

... safeguarding the system from undefined runtime behavior which would occur if an attempt to store a value larger than `long` is made somewhere during the modeling process.

> This error can be disabled by passing `-ignoreINTEGERranges` to the code generator.

8.3 REALs

For the same reasons as with `INTEGER`s, `REAL`s need their range specified as well:

```
angleInDegrees REAL (0.0 .. 360.0)
```

If no range is specified, you'll get an error:

```
REAL (in ...) must have a range constraint inside
ASN.1, or else we might lose accuracy during runtime!
```

> This error can be disabled by passing `-ignoreREALranges` to the code generator.

The representable values are those defined by the IEEE754 double precision standard.

8.4 ENUMERATED

Use integer values for each enumerant:

```
Enum ::= ENUMERATED
{
    one(1),
    two(2)
}
```

Otherwise, you'll get ...

```
ENUMERATED must have integer values for each enum! (...)
```

The definition of values for each enum is mandatory, to prevent from ambiguous representations; by using INTEGER values, there is a guarantee that all code using the ENUMERATED has the same understanding on which value means which enum.

8.5 OCTET STRINGs

The "glue" mapping generates statically sized arrays of bytes for OCTET STRINGs; in addition, SCADE will only generate C code for character arrays it knows the length of; therefore, open ended `OCTET STRINGs`, `UTF8Strings`, etc are unusable: The following ...

```
description OCTET STRING
```

... would cause this error:

```
string (in ...) must have a SIZE range set!
```

Something like this must be used instead:

```
description OCTET STRING (SIZE(1..10))
```

This error can't be bypassed; it blocks the generation of code.

Also note that OCTET STRINGs are a "last resort"; the content inside them can be usually broken down in simple (base) ASN.1 types, and that is the prefered approach in ASN.1, not "black-box" types.

8.6 BITSTRINGs

Copying verbatim from Olivier Dubuisson's reference work[17], ...

> More generally, the use of the OCTET STRING and BITSTRING types should always be the last resort once all the other ASN.1 types have been discarded as inappropriate models of the problem in hand.

The purpose of ASN.1 is to *abstract* the data structures, releaving the designer from the burden of how they are actually represented. In other words, if the desired data structure is an array of boolean flags, then by all means use a `SEQUENCE OF BOOLEAN` ; don't use `BITSTRING`s. The desired output encoding has nothing to do with the data structure ; it is done through the ASN.1 encodings (DER [18], CER [18], PER [19], XER [20] etc). `BITSTRING`s are not supported by the Data Modeling Toolchain.

8.7 SEQUENCEs

Supported, but without OPTIONAL fields and DEFAULT values. The former makes no sense in the context of ASSERT, while the latter (DEFAULT values) is provided in the functional modeling code, not in the bridging ASN.1 structures.

8.8 CHOICEs

`CHOICE`s are supported fully, with the introduction of an extra `choiceIdx` variable in the generated code, which is filled with an integer value (0 means that the first choice was used, 1 means the second, etc). The mapping code therefore always checks this value first and acts accordingly.

8.9 SEQUENCE OF

For the same reason mentioned in OCTET STRINGs, SEQUENCE OFs need SIZE constraints:

```
Matrix ::= SEQUENCE (SIZE(12)) OF Line
```

... otherwise, an error occurs:

```
SequenceOf (in ...) must have a SIZE range set!
```

Notice that if you use base types in the SEQUENCE OF, the previously mentioned base type constraints must also exist, so this is a valid example:

```
ArrayOfInt ::= SEQUENCE (SIZE(1 .. 12)) OF INTEGER (0 .. 10)
```

... describing an array of up to 12 integers, each one of which can contain values from 0 to 10.

8.10 SETs and SET OFs

Supported, but as `SEQUENCE`s and `SEQUENCE OF`s respectively. In other words, the handling of the unordered nature of SET fields is left to the code generated by the ASN.1 compiler (which represents them in the generated code as normal, "fixed" C structs), while the SET OF is handled as a simple array of the contained type: the extra knowledge that no two elements can be the same is enforced by the calling code, not by the toolchain. Keep in mind, though, that... (Copying again from [17]):

> It is recommended to use the SEQUENCE OF type as much as possible: indeed, when using canonical encoding rules such as the CER or the canonical PER (see Part III on page 391), the SET OF type requires dynamically sorting the encoding of every value before transmitting it!

Chapter 9

Appendix B - The ASN.1 Abstract Syntax Tree

This appendix includes details on the information carried by the ASN.1 Abstract Syntax Tree. This is the information carried around by the `node` parameters in the mapping backends. They therefore provide access to all that is necessary during the mapping process.

As seen in the listings of section 5.2, the functions acting on each APLC parameter take at least two arguments:

- The ASN.1 typename (as a simple string)
- The ASN.1 node in the ASN.1 Abstract Syntax Tree

The second one points to the node-specific information in the ASN.1 AST. This can be one of the following types, each one with its own, type-specific information:

9.1 AsnBool

```
class AsnBool(AsnBasicNode)
   This class stores the semantic content of an ASN.1 BOOLEAN.
   Members:
       _name : the name of the type (or var)
       _isOptional : if True, node is optional
       _bDefaultValue : one of True,False,None.
```

9.2 AsnInt

```
class AsnInt(AsnBasicNode)
   This class stores the semantic content of an ASN.1 INTEGER.
   Members:
       _name : the name of the type (or var)
       _isOptional : if True, node is optional
       _range : a tuple containing the valid range for the integer or []
       _iDefaultValue : either None, or the default value for this integer
```

9.3 AsnReal

```
class AsnReal(AsnBasicNode)
```

```
   This class stores the semantic content of an ASN.1 REAL.
   Members:
       _name : the name of the type (or var)
       _isOptional : if True, node is optional
       _range : a tuple containing the valid range for the integer or []
       _baseRange,
       _mantissaRange,
       _exponentRange  : single or double element tuples containing
                         the allowed ranges for respective values.
                         Or [].
       _dbDefaultValue : either None, or the default value for this real
```

9.4 AsnString

```
class AsnString(AsnBasicNode)
   This class stores the semantic content of an ASN.1 String.
   Members:
       _name : the name of the type (or var)
       _isOptional : if True, node is optional
       _range : a tuple containing the allowed string size or []
```

9.5 AsnEnumerated

```
class AsnEnumerated(AsnComplexNode)
   This class stores the semantic content of an ASN.1 enumeration.
   Members:
       _name : the name of the type
       _isOptional : if True, node is optional
       _members : a tuple of all the allowed values for the enumeration.
                  Each value is itself a tuple, containing the name
                  and the integer value associated with it (or None,
                  if it is ommited)
       _default : if one of the values of the enumeration is the default,
                  it is contained in this member
```

9.6 AsnSequence

```
class AsnSequence(AsnComplexNode)
   This class stores the semantic content of an ASN.1 SEQUENCE.
   Members:
       _name : the name of the type
       _isOptional : if True, node is optional
       _members    : a tuple of all child elements. Each tuple contains
                     two elements: the name of the variable and the
                     type itself (as an AsnInt, AsnReal, ... or an
                     AsnMetaMember).
```

9.7 AsnSequenceOf

```
class AsnSequenceOf(AsnComplexNode)
```

```
    This class stores the semantic content of an ASN.1 SEQUENCEOF.
    Members:
        _name : the name of the type
        _isOptional : if True, node is optional
        _containedType : the contained element (either a string or AsnNode)
        _range : [] or a tuple with the allowed size range.
```

9.8 AsnMetaMember

```
class AsnMetaMember(AsnNode)
    This class stores the semantic content of a member type of a
    CHOICE or SEQUENCE.
    Members:
        _isOptional : if True, node is optional
        _containedType : the contained element as a string (type name)
```

All nodes of the ASN.1 Abstract Syntax Tree inherit from AsnNode, which offers the `Location()` member: if something must be reported that pertains to the specifics of the ASN.1 type, the backends can report the whereabouts of the type by printing the string return by `Location()`.

Chapter 10

Appendix C - The online gateways

The `asn2aadlPlus` tool of the Data Modeling toolchain was developed on two paths, almost from the start.

The first one was the "traditional" one, where the end user has to install the toolchain in his working machine (Windows/Linux), install the necessary support software, and proceed with the invocations.

It became immediately apparent, however, that there was another alternative, which would significantly ease the process: offering the same functionality over Web gateways, where the only thing required was a Web browser (Internet Explorer, Mozilla Firefox, etc). In this manner, the end users simply accessed our appropriately setup page, and were able to perform the translations of their ASN.1 grammars to both AADL [10] and UML [4] definitions.

From the main site (`http://www.semantix.gr/assert/`), the end user navigated to the Downloads section, and from there, two links offered direct access to:

- The ASN.1 to AADL gateway, which translated the ASN.1 definitions into DATA definitions suitable for use from within LabASSERT (as described in 2.1).
- The ASN.1 to UML gateway, which translated the ASN.1 definitions into RCM definitions suitable for use from within Eclipse (as described in 2.1).

The following screenshots show a characteristic usage of the gateways:

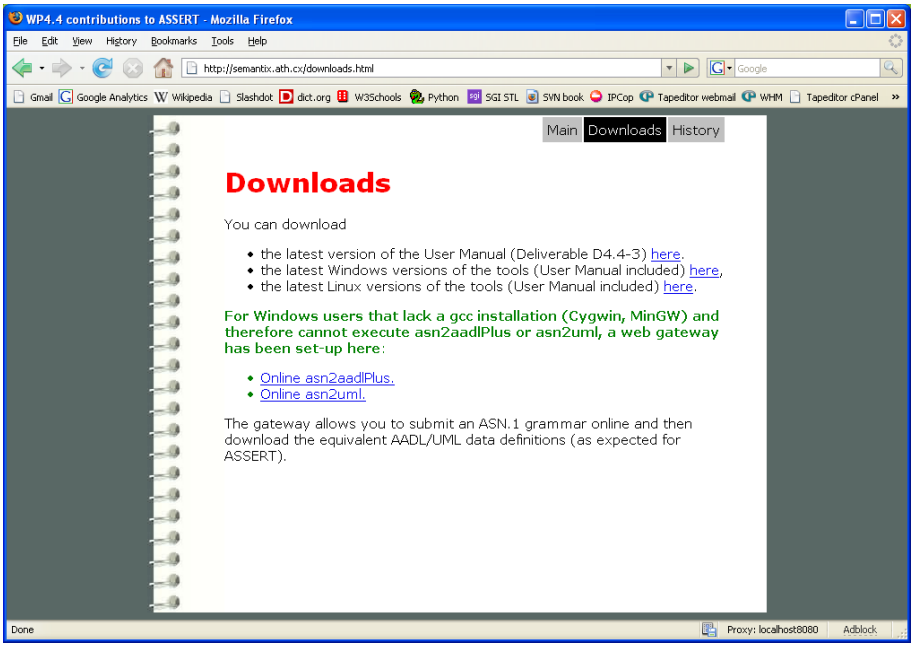

Figure 10.1: Accessing the gateways

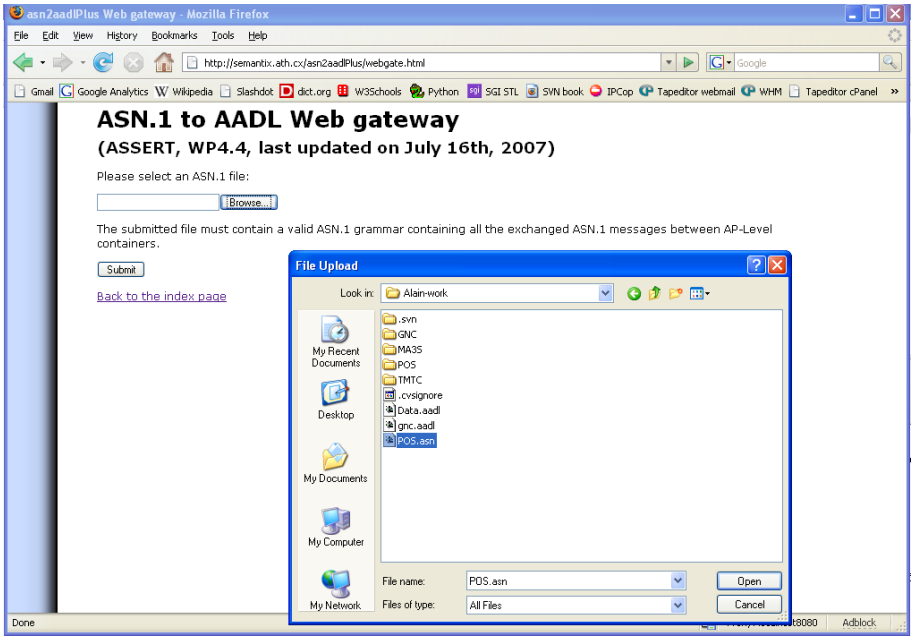

Figure 10.2: Submitting an ASN.1 grammar

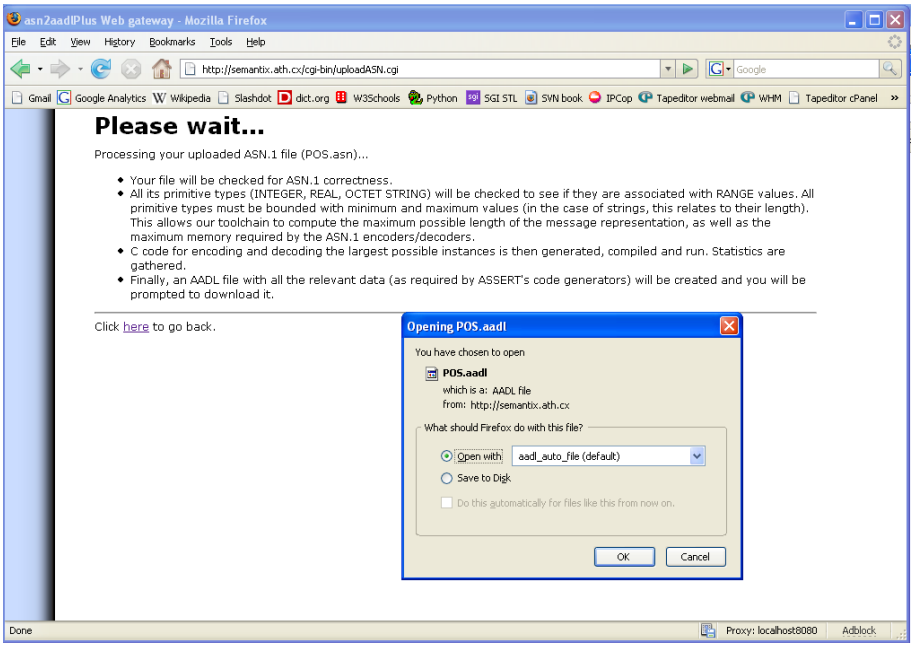

Figure 10.3: Obtaining a result

Chapter 11

Appendix D - Technical notes on the backends

11.1 Model level mappers

11.1.1 ObjectGeode profile

ObjectGeode's model mapper logic is stored inside file `og_A_mapper.py`. Since ObjectGeode is an SDL tool, it includes native support for ASN.1; that is, ASN.1 grammars can be used in their original form inside the .pr files that define an SDL subsystem.

Unfortunately, this isn't entirely accurate... The SDL parser used by ObjectGeode imposes some minor restrictions in the way ASN.1 content appears, and this mapper (`og_A_mapper.py`) takes extra care to handle this.

- When used inside an ObjectGeode SDL file (a .pr file), ASN.1 definitions are not allowed to use hyphens, since hyphens are considered by ObjectGeode's parser to denote subtraction (or negation). The mapper therefore "patches" the original ASN.1 definitions into a form more suitable for consumption from ObjectGeode (i.e. replacing '-' with '_')
- For the same reason, comments - which start with '--' - have to be eliminated
- As per ESA's request, the definitions of ASN.1 types must reside in a section named 'Datamodel' ; the mapper replaces the original name with this.
- Obviously, hyphens are not changed when found inside range constraints

The mapper uses regular expressions to locate the appropriate places and apply these rules in the text of the original ASN.1 grammars. To be able to cope with all possible ASN.1 grammars, a full fledged ASN.1 parser would be required; however, this regular-expression based implementation covers almost all cases without requiring the effort necessary for a full-blown ASN.1 parser.

11.1.2 Matlab/SIMULINK profile

The Matlab/SIMULINK mapper is stored inside file `simulink_A_mapper.py`. The logic behind this mapper was built in close co-ordination with ESA Simulink experts.

- BOOLEANs are directly mapped to `boolean`s, inside `AliasType`s.
- Simulink offers 6 distinct target types for INTEGERs: `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`. They constitute the different combinations of signed and unsigned integers of 8, 16 and 32 bits - and the

mapper takes the ASN.1 constraints of INTEGERs into account when creating the appropriate target SIMULINK type (`AliasType`). The `Description` attribute also offers the exact range constraint used.

- REALs are mapped to `double`s (inside `AliasType`s). The `Description` attribute also offers the exact range constraint used.
- Any ENUMERATED is mapped to an `int32`. The originating ASN.1 type is specified in a special comment accompagnying the new type ("values of ENUMERATED EnumTypeName"). The values themselves are available as Simulink values, named `EnumTypeName_value_OptionName`, where EnumTypeName is the ASN.1 type name and OptionName is the name of the enumerant option.
- SEQUENCEs are mapped to Simulink `Bus`es. The `Elements` attribute is used to enumerate all the comprising types, so the necessary types are defined by the mapper before the actual `Bus` definition. Fields using standard types get "temporary" types: For example, any ASN.1 string field gets a corresponding "`octet_string_X`" pseudo-type, with X being an increasing number (per string), and each temporary type reflecting any possible string specific size constraint.
- CHOICEs are mapped just like SEQUENCEs, but with an extra `choiceIdx` of type `uint8`. This index will point to which option is actually used in the Choice (the index starts at 1 for the first option).
- SEQUENCE OFs are represented with a number of individual elements of the contained type, named `TypeName_member_X`.
- SETs and SET OFs are handled just as SEQUENCE and SEQUENCE OFs.

11.1.3 SCADE profile

The SCADE mappers are stored in files `scade5_A_mapper.py` and `scade6_A_mapper.py`. Only minor differences exist between the two mappers, to accomodate for some type-specific changes introduced in SCADE6.

All type declarations are placed in a `System_Types` SCADE package.

- BOOLEANs are mapped to `bool`s.
- INTEGERs are mapped to `int`s.
- REALs are mapped to `real`s.
- Any ENUMERATED is mapped as a SCADE enumeration (with the possible value for the option enclosed in brackets, after the option name).
- SEQUENCEs are mapped to compound SCADE structures (defined by opening and closing braces in SCADE6, or opening and closing brackets in SCADE5). The field names used are the same ones as those used in the ASN.1 grammar, with any non-alphanumeric character changed into '_'.
- CHOICEs are handled just like SEQUENCEs, but with an extra field (`choiceIdx` of type `int` that denotes which option is actually used in the Choice (the index starts at 1 for the first option).
- SEQUENCE OFs are mapped using the '^' syntax, adding the cardinality immediately after the carret (taken from the maximum size contraint of the SEQUENCE OF).
- SETs and SET OFs are handled just as SEQUENCE and SEQUENCE OFs.

11.1.4 Pragmadev RTDS profile

RTDS is an SDL tool, so it must include native support for ASN.1. The company (Pragmadev) is in fact working with ESA right now to introduce this support in the tool. If this inclusion of ASN.1 support requires any additional work on the model mapper side, then `rtds_A_mapper.py` will be enhanced to do so. Currently, it is doing nothing.

11.2 Code level mappers

11.2.1 Synchronous/Asynchronous APIs

During the development phase of the ASSERT code mappers, it became clear that many parts of the building process were repeated every time a new mapper was built. This repetitive work was both tedious and error-prone. If a bug was identified in the repeating code, it had to be fixed in all synchronous backends. Moreover, the copy/paste involved was causing errors difficult to identify; minor changes were lost in copying and their side-effects could only be identified through extensive regression tests.

To address this challenge, the common aspects of the build process were investigated to identify their recurring patterns and to make use of them during the building of the mappers. This resulted in the construction of two APIs, the Synchronous and the Asynchronous one, which now form the bases that all mappers must adhere to.

Synchronous API

As described in section 3.1, Synchronous modeling tools are used to create synchronous, cycle-driven systems. The backbone of mappers for such tools is common, and consists of:

HeadersOnStartup: tool-specific code that must be placed in the output files (e.g. special `#include`s, etc)

InitializeBlock: code that calls the initialization code for the subsystem

ExecuteBlock: code that calls the execution code ("Perform One Cycle") for the subsystem

SourceVar: code that identifies the source variable to map from

TargetVar: code that identifies the target variable to map to

FromToolToASN1SCC, FromToolToOSS, FromASN1SCCtoTool, FromOSStoTool: code that recursively descends into the ASN.1 Abstract Syntax Tree and creates runtime mappings for each ASN.1 type met, between the pertinent data structure generated by an ASN.1 compiler and the pertinent data structure generated by a modeling tool

The rest of the synchronous mappers consists of patterns repeating for every tool: generation of Encode and Decode function declarations and definitions, invocations of ASN.1 compiler specific encoders and decoders (Semantix ASN1SCC or OSS/Nokalva), etc. All backends for synchronous tools make use of these repeating patterns by way of inheritance. More precisely, they...

1. Define a `isAsynchronous = False` global variable in the mapper's global scope
2. Define the four distinct recursive mapper engines:
   (a) From tool to ASN1SCC (`FromToolToASN1SCC`)
   (b) From tool to OSS/Nokalva (`FromToolToOSS`)
   (c) From ASN1SCC to tool (`FromASN1SCCtoTool`)
   (d) From OSS/Nokalva to tool (`FromOSStoTool`)

   These four engines also make use of a base class, called `RecursiveMapper`, which performs the recursive descent into the ASN.1 AST, and calls mapper-defined primitives for each ASN.1 Type (see Table 11.1)
3. Define a version reporting function (`Version`)
4. Define a function that writes any additional startup work (`HeadersOnStartup`)
5. Define a function that returns the source variable for the mappers (`SourceVar`)

`MapBoolean(srcVar, destVar, node, leafTypeDict, names)`
`MapInteger(srcVar, destVar, node, leafTypeDict, names)`
`MapReal(srcVar, destVar, node, leafTypeDict, names)`
`MapOctetString(srcVar, destVar, node, leafTypeDict, names)`
`MapEnumerated(srcVar, destVar, node, leafTypeDict, names)`
`MapSequence(srcVar, destVar, node, leafTypeDict, names)`
`MapSet(srcVar, destVar, node, leafTypeDict, names)`
`MapChoice(srcVar, destVar, node, leafTypeDict, names)`
`MapSequenceOf(srcVar, destVar, node, leafTypeDict, names)`
`MapSetOf(srcVar, destVar, node, leafTypeDict, names)`
srcVar and destVar: the string representations of the source and destination variables
node: the Abstract Syntax Tree node (see Chapter 9)
leafTypeDict: dictionary specifying the leaf type (string) of each node
names: dictionary that provides AST node from typename (string)

Table 11.1: Callbacks invoked from within `RecursiveMapper`

6. Define a function that returns the target variable for the mappers (`TargetVar`)
7. Define a function that writes the code for subsystem initialization (`InitializeBlock`)
8. Define a function that writes the code for subsystem cycle execution (`ExecuteBlock`)

This API greatly reduced the code inside the synchronous code mappers described in the following sections. It also made them easier to develop, less error-prone, and easier to maintain.

Asynchronous API

Just as their synchronous counterparts, asynchronous modeling tools also share common attributes. These common parts were reflected in the Asynchronous API for the code mappers, which is as follows:

HeadersOnStartup: tool-specific code that must be placed in the output files (e.g. special `#includes`, etc)

Encoder: tool-specific code that outputs code per ASN.1 Type. The code generated reads data from pertinent modeling-tool generated data structures and encodes them into caller-specified buffers using either Unaligned Packed Encoding Rules or memory dumps of ASN1SCC generated pointerless data structures.

Decoder: tool-specific code that outputs code per ASN.1 Type. The code generated reads data from caller supplied buffers and decodes them into pertinent modeling-tool generated data structures. The source data can be either bitstreams generated using Unaligned Packed Encoding Rules or memory dumps of ASN1SCC generated pointerless data structures.

Compared to the Synchronous API, the extent of this API appears to be somewhat smaller. The reason for this difference can be traced to the nature of Asynchronous tools: They are supposed to be used to model asynchronous processes, which can call (and be called) whenever they choose to. The only guaranteed procedure in the context of these calls is the translation that takes place between the tool-generated data structures and the corresponding ASN.1 compiler generated data structures. The implementations of the `Encoder` and `Decoder` methods can also employ re-use by inheriting from the `RecursiveMapper` (and thus having the recursive descent into the ASN.1 Abstract Syntax Tree and the callbacks of simple mapping methods for free); still, Asynchronous tools are indeed less amenable to "categorization" and "templating" than Synchronous ones.

Calls to type specific mappers are done from within the base `RecursiveMapper` class, and are thus exactly the same as in Table 11.1.

11.2.2 ObjectGeode profile

ObjectGeode is an SDL tool and is, by design, used to create asynchronous systems. It therefore falls into the Asynchronous API described in section 3.2. The code for the mapper is inside file `og_B_mapper.py`.

Combined with the work done inside `buildSupport`, the mapper is responsible for writing the code for the following macros (per ASN.1 Type):

DECODE_UPER_TypeName: Works on an input stream, decoding the Unaligned Packed Encoding Rules bitstream into ASN.1 structures. These structures have been generated by ASN1SCC (Semantix's Space Certifiable ASN.1 Compiler) or OSS Nokalva's ASN.1 compiler. It then proceeds to map the data, field by field, to the appropriate destination data structures as generated by ObjectGeode's code generator. The macro syntax is `DECODE_UPER_TypeName(pBuffer, iBufferSize, pSdlVar)`, with pBuffer and iBufferSize pointing to the input buffer and input buffer size, respectively. pSdlVar points to the ObjectGeode variable that will receive the data. The choice between OSS and Semantix's compiler is done on the command line of `aadl2glueC` (depending on the existence or absence of the argument `-useOSS`).

DECODE_NATIVE_TypeName: When not crossing process boundaries, communication across APLCs need not suffer the unnecessary encoding/decoding of ASN.1 streams. In that case, the self-contained (pointerless) structures generated by ASN1SCC are used as carriers of semantic content. This macro operates on said structures, reading the passed-in information and mapping it, field by field, to the appropriate destination data structures as generated by ObjectGeode's code generator. The macro syntax is `DECODE_NATIVE_TypeName(pBuffer, iBufferSize, pSdlVar)`, with pBuffer and iBufferSize pointing to the input buffer and input buffer size, respectively. pSdlVar points to the ObjectGeode variable that will receive the data. Notice that the data pointed to by pBuffer are in fact the memory dump of the corresponding ASN1SCC generated data structures.

ENCODE_UPER_TypeName: The reverse of DECODE_UPER_TypeName: encodes the contents of an ObjectGeode variable into the data structures generated by an ASN.1 Compiler (Semantix's ASN1SCC or OSS/Nokalva's). It then proceeds to call the ASN.1 encoder function, and stores the resulting bitstream into the destination buffer. The macro syntax is `ENCODE_UPER_NAME(varName, param1)`, with varName being the output buffer (declared with `DECLARE` and `DEFINE_Typename`, see below) and param1 pointing to the source ObjectGeode variable.

ENCODE_NATIVE_Typename: The reverse of DECODE_NATIVE_Typename: encodes the contents of an ObjectGeode variable into the data structure generated for the ASN.1 Typename by Semantix's ASN1SCC ASN.1 compiler. The memory dump of the structure is then taken and copied in the destination buffer. The macro syntax is `ENCODE_NATIVE_NAME(varName, param1)`, with varName being the output buffer (declared with `DECLARE` and `DEFINE_Typename`, see below) and param1 pointing to the source ObjectGeode variable.

DECLARE_Typename: This macro is used to declare enough buffer space (statically allocated) for an ASN.1 type. The macro syntax is `DECLARE_Typename(varName)` which translates to an appropriate declaration for a buffer named `varname`.

DEFINE_Typename: This macro is used to reserve (define) enough buffer space (statically allocated) for an ASN.1 type. The macro syntax is `DEFINE_Typename(varName)` which translates to an appropriate definition for a buffer named `varname`.

The above macros are put to use by the signal sending and receiving macros of ObjectGeode. Their use is limited to the files `vm_if.c`, `hpostdef.h` and `hpredef.h`. More specifically:

vm_if.c: For each Provided Interface which is provided by the modeled APLC, a "bridge" function is created inside this file. This function is in one of two forms:

1. if the PI has no associated PARAMETER with it, then this function is simply doing

(a) `G2S_OUTPUT(...);`

(b) `sdl_loop_FVNAME();`

2. if the PI has an associated PARAMETER with it, then this function has to...

   (a) Decode the ASN.1 message (incoming arguments: pointer,length)

   (b) place the decoded values in the appropriate ObjectGeode generated global variable (for this PI)

   (c) `G2S_OUTPUT(...);`

   (d) `sdl_loop_FVNAME();`

hpostdef.h: For each type used by this APLC, a call to `DEFINE_ASN1TYPENAME(name_of_variable)` must be issued, to reserve space for the variable. The macro knows (by construction) what size to reserve to accomodate the maximum configuration of this message.

Also, for each Required Interface used by the modeled APLC, this header file must also declare a C MACRO:

```
#define riname(param1, param2, ...)
```

...which will:

1. encode all the input data from the input params, via the ENCODE_ macros, into the reserved spaces (the DEFINE_ASN1TYPENAME ones)
2. call the vm callback to access the RI (vm_fvname_riname)
3. decode all the output data from the output params, via the DECODE_ macros.

This MACRO will be called by the ObjectGeode generated code whenever the RI is to be called.

hpredef.h: Contains the prototypes (extern declarations) of the vm callbacks.

11.2.3 Matlab/SIMULINK profile

The code for this mapper resides in `simulink_B_mapper.py`. It follows the synchronous API paradigm (see section 3.1). In plain terms, this means that access to this kind of subsystems is offered via two functions: one that initializes the subsystem, readying it for use, and a second one that executes it: the "cycle" function, which reads the input parameters, acts upon them and (possibly) populates output parameters.

The way the Real-Time Workshop (RTW) code generator operates makes this task easier than other tools: for one, both initiliazation and execution functions are already there, called SubsystemName_initialize and SubsystemName_step. For another, input and output parameters are easily accessible as global variables, named SubsystemName_Y and SubsystemName_U respectively.

The mapper therefore has a clear task: to use the appropriate recursive mapping rules when assigning data into and outof the RTW-generated data structures. The source/target data structures can again be either the Semantix ASN1SCC or the OSS/Nokalva generated data structures. When using Native encodings, just as with ObjectGeode, we use memory dumps of the ASN1SCC data structures, since they are self-contained (pointerless).

Code generated by RTW is very close to the one humans would actually write, and this applies to the data structures as well. The recursive descent into the ASN.1 AST doesn't need to address any special needs for temporary types or anything else: the mapping is very straightforward for RTW code: in around 60 lines we have a complete mapper - and this applies for all directions: OSS to Simulink, Simulink to ASN1SCC, etc (see classes `FromOSStoSimulink`, `FromSimulinkToASN1SCC`, etc).

11.2.4 SCADE5/6 profile

The code for this mapper resides in `lustre_B_mapper.py`. It follows the synchronous API paradigm (see section 3.1). In plain terms, this means that access to this kind of subsystems is offered via two functions: one that initializes the subsystem, readying it for use, and a second one that executes it: the "cycle" function, which reads the input parameters, acts upon them and (possibly) populates output parameters.

This mapper utilizes a macro support layer written by Esterel Technologies specifically for the utilization of SCADE in the ASSERT project. The SCADE code generator was tweaked towards that goal, and enhanced with a new set of access macros. This made our task straightforward: both initiliazation and execution functionality is accessible via ready-made macros, called AADL2SCADE_INIT and AADL2SCADE_PERFORM. The input and output parameters are also accessible via macros, named AADL2SCADE_INPUT_PARAMETER and AADL2SCADE_OUTPUT_PARAMETER.

The mapper therefore has a clear task: to use the appropriate recursive mapping rules when assigning data into and outof the SCADE-generated data structures. The source/target data structures can again be either the Semantix ASN1SCC or the OSS/Nokalva generated data structures. When using Native encodings, just as with ObjectGeode, we use memory dumps of the ASN1SCC data structures, since they are self-contained (pointerless).

SCADE, just like RTW, is also using straightforward mapping rules for structure fields. The recursive descent into the ASN.1 AST doesn't need to address any special needs for temporary types or anything else: It takes around 65 lines of code to complete a full recursive mapper for SCADE generated code (see classes `FromOSStoSCADE`, `FromSCADEtoASN1SCC`, etc).

11.2.5 Pragmadev RTDS profile

The code for this mapper resides in `rtds_B_mapper.py`. It follows the asynchronous API paradigm (see section 3.2).

In contast to the ObjectGeode profile (which is also an SDL-based one), this mapper does not generate macros. The way RTDS generates its structures allows for a cleaner interface between the RTDS and ASN1SCC code.

In its current form (which is rapidly evolving) the generated glue code maps all the fields (one-to-one) between the corresponding fields of the two categories of data structures, via functions named like this:

Convert_TypeName_from_RTDS_to_ASN1SCC: These functions...

Convert_TypeName_from_ASN1SCC_to_RTDS: ...and these functions, are combined in the code generated by buildsupport, to transfer the semantic content back and forth between the Pragmadev code and the ASN1SCC code.

11.3 ASN.1 Compilers

The code mapping process is heavily influenced by the code generation details of the ASN.1 compilers that are used. In particular, when native encodings are used, the message buffers are carrying simple memory dumps of Semantix ASN1SCC generated data structures. When using Unaligned PER (Packed Encoding Rules), a choice appears: the end system can either use OSS/Nokalva's ASN.1 compiler or Semantix's ASN1SCC.

11.3.1 Semantix/asn1scc

This compiler was designed and built from scratch to address the needs of software in the space domain. The code it generates has no need for a memory heap (it doesn't invoke any system calls for dynamical allocation of memory), and can therefore be used in memory-deprived platforms (e.g. embedded space designs).

Specific details about the code generation strategies employed by ASN1SCC, as well as user manuals and the relevant software are available [15].

11.3.2 OSS/Nokalva

OSS/Nokalva's ASN.1 compiler can also be used by the Data Modeling toolchain. To that end, the ASN.1 grammar is automatically *decorated* during the system build process with the following OSS-specific directives:

- `--<OSS.PDU>--`
- `--<OSS.ARRAY IA5String>--`
- `--<OSS.ARRAY BITSTRING>--`
- `--<OSS.ARRAY OCTETSTRING>--`
- `--<OSS.ARRAY SET OF>--`
- `--<OSS.ARRAY SEQUENCE OF>--`

These directives instruct the OSS ASN.1 compiler to generate (to the extent it can) self-contained data structures that don't include pointers.

The code mappers, as discussed in section 11.2.1, include recursive descent mapping logic for both ASN1SCC and OSS/Nokalva. Comparing the implementations, it is clear that both tools try to mimic (to the extent possible) the original ASN.1 field and type declarations - they however diverge in some other aspects:

- `OCTET STRING`s are represented as compound structures, with one field storing the length (`nCount` in ASN1SCC, `length` in OSS) and another field storing the data (`arr` and `value` respectively).
- Both tools use C unions to represent CHOICEs, and store the selected option via a C enumeration (named `kind` in ASN1SCC and `choice` in OSS). The enumerants are named `whatever_PRESENT` in ASN1SCC and `OSS_whatever_chosen` in OSS.
- `SEQUENCEOF`s are represented as C arrays (fields named `arr` and `value` respectively), and their length is stored in separate integer variables (`nCount` for ASN1SCC and `count` for OSS).

Overall, the impact of using another ASN.1 compiler is not as big as one would expect, since most of the mapping logic ends up being very similar. The APIs themselves (11.2.1 were also designed from the start to cope with this.

11.4 Automatically created GUIs

When applied to real systems, the tools and techniques described in this user manual distill down to...

- an overall system-level description (in AADL) that includes the system-level interfaces
- an overall message-level description (in ASN.1) that includes all the messages exchanged between subsystems

This kind of information offers an unexpected bonus: by having access to both the subsystem interfaces and the ASN.1 types they are expecting, a code generator would be able to automatically create graphical user interfaces offering interactive access to subsystem interfaces (see Figure 11.1). This code generator was built and proved to be a very welcome addition to the toolchain.

Through access to the ASN.1 grammar, the code generator can create code with the appropriate GUI widgets and code constructs to...

- Create tree-like dialog controls that offer access to interface parameters

- Allow the operator to populate them, while checking for correctness (appropriate data used for INTEGER/REAL edit controls, validations of ASN.1 constraints, etc)
- Allow the operator to re-use message data, by Saving and Loading them (easily implemented via the ASN.1 encoders and decoders)
- Allow the operator to invoke the interface at runtime, thus interacting with a real running system

WxWidgets/C++ was chosen as the code generator's target. WxWidgets is a cross-platform GUI and tools library for MS Windows, GTK (Linux/FreeBSD/etc) and MacOS. This, in effect, allows for compilation and execution of the generated GUI code under any modern operating system.

Invocation of the interfaces depends on some method of communication between the GUI and the running system. A number of Inter Process Communication mechanisms (IPC) exist for this purpose; currently, the implementation uses named message queues to transfer data between the GUI and the running system. For

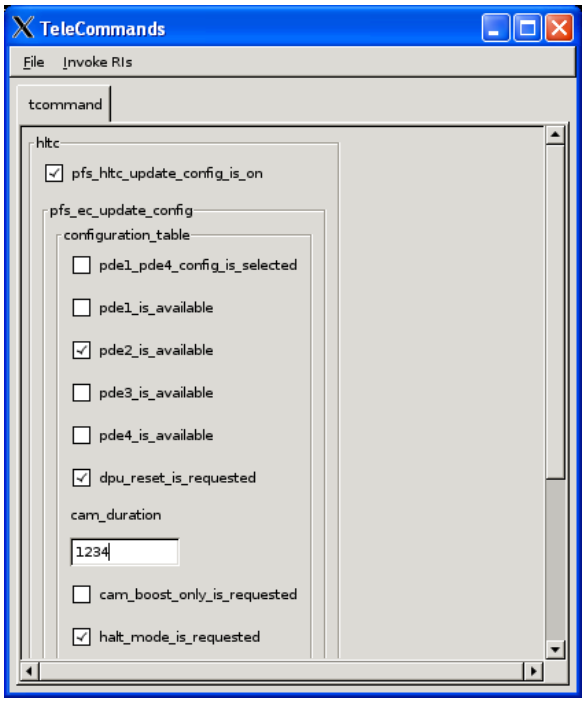

Figure 11.1: Automatic GUIs for system interfaces

this to work, both of these applications must be running in the same machine (POSIX message queues can not generally be used for communication between different machines). Eventually, the more powerful ASSERT VM technology may be used to allow for these invocations over any of the ASSERT sanctioned buses (Spacewire, Ethernet, etc).

11.5 Automatically created Python bridges

The principles behind automatically written GUIs 11.4 were also put to use in building automatically-generated Python bridges.

The reasoning behind using Python (or any other interpreted language) is very simple: Testing the (usually complex) logic inside space systems requires big regression checking suites. In the absence of support for interpreted languages, the checking code has to use the low-level APIs, which is a tedious and error-prone process.

The ASSERT tools, however, automatically create Python bridges that offer direct access to the contents of the ASN.1 parameters, as well as direct runtime access to the TM/TCs offered by the system. All that the user needs to do to create his set of regression checks, is to write simple Python scripts, that exercise any behavioural aspect of the system. As an example, the automatically generated Python interface can be used to write a scenario like this:

> When I send a TC with value X in param Y, then I expect a TM after a max waiting of Z seconds, with the value K in the incoming param L

This kind of scenario can be expressed in less than 10 lines of Python code, that is, an order of magnitude less work than the corresponding C code.

The code below is a simple (but complete) example that illustrates how ASN.1 types can be handled from within Python:

```
#!/usr/bin/env python
import sys

#
# From this ASN.1 grammar:
# (file DataView.asn)
#
#     DataView DEFINITIONS AUTOMATIC TAGS ::= BEGIN
#
#     T-REAL ::= REAL( -10000.0 .. 10000.0 )
#
#     END
#
# You process the ASN.1 file like this:
#
# bash$ export ASN2DATAMODEL=/path/to/asn2dataModel/asn2dataModel
# bash$ mkdir output
# bash$ $ASN2DATAMODEL -o output -toPython DataView.asn
# bash$ cd output/
# bash$ make -f Makefile.python
#
# After that, you can run this script...

from DataView_asn import *

def testReal(val):
```

```
    # Create a stream buffer to host the encoded data
    d1 = DataStream(DV.T_REAL_REQUIRED_BYTES_FOR_ENCODING)
    # Create a T_REAL
    f1 = T_REAL()
    # Set the value
    f1.Set(val)
    try:
        # Encode the value of f1 into the buffer d1
        f1.Encode(d1)
    except:
        print "Encoding failed..."
        sys.exit(1)

    binaryData = d1.GetPyString() # Get the encoded stream bytes
                                  # as a Python string. You can
                                  # pass these data over sockets,
                                  # save them to files, etc

    # Create a second stream buffer, put the encoded data
    # inside it, and decode from it.
    # Another way (one that avoids a separate DataStream)
    # is to re-use d1, calling d1.Reset (which "rewinds"
    # the stream indexes to the start, thus making it ready
    # for access by the Decoders)
    d2 = DataStream(DV.T_REAL_REQUIRED_BYTES_FOR_ENCODING)
    d2.SetFromPyString(binaryData)
    f2 = T_REAL()
    f2.Decode(d2)
    #f2.PrintAll()
    assert(abs(val - f2.Get())<1e-5)

if __name__ == "__main__":
    i = -5.0
    while i<5.0:
        testReal(i)
        i+=0.001
```

... and the following code is an example of TM/TCs communication with an actual system:

```
import time, sys

# DV, DataView_asn and PythonController are automatically written
# by the ASSERT tools (in this example)

import DV
from DataView_asn import *
from PythonController import *

cnt = 0
try:
# Poll_mygui is a class automatically written by ASSERT tools, which
# is imported from PythonController and polls for arriving TMs
# in a thread, reporting their arrivals...
```

```
    thread = Poll_mygui()
    thread.start()
    while True:
        # While polling in a thread for arriving TMs,
        # we will also send a TC every two seconds:
        print "Sleeping for two seconds..."
        time.sleep(2)
        print "Invoking TC"

        # Direct access to ASN.1 types from Python:
        a = TC_T()
        a.action.kind.Set(DV.display_PRESENT)
        cnt += 1
        a.action.display.SetFromPyString("abc" + str(cnt))
        a.destination.Set(Destination_T.displayer)
# Show the contents of the ASN.1 message before you send it
        a.PrintAll()
# automatically written accessor to the TC, imported from PythonController
        Invoke_router_put_tc(a)
except:
    sys.exit(1)
```

Chapter 12

Abbreviations

AADL	Architecture Analysis and Design Language
APLC	AP-Level Container
ASN.1	Abstract Syntax Notation number One
ASSERT	Automated proof-based System and Software Engineering for Real-Time applications
AST	Abstract Syntax Tree
CER	Canonical Encoding Rules
DER	Distinguished Encoding Rules
ENST	Ecole National Superieure des Tlcommunications
ESA	European Space Agency
ESTEREL	Esterel Technologies
HOOD	Hierarchical Object Oriented Design
IEEE	Institute of Electrical and Electronics Engineers, Inc.
PBSE	Proof Based System Engineering
PER	Packed Encoding Rules
PI	Provided Interface
PP	Pilot Project
RCM	Ravenscar Computational Model
RI	Required Interface
SCADE	Safety Critical Application Development Environment
SDL	Specification and Description Language
TN	Technical Note
UML	Unified Modeling Language
UPD	Padua University
VM	Virtual Machine
VMLC	VM Level Container

Bibliography

[1] ITU-T: Rec. X.680-X.683, ISO/IEC: Abstract Syntax Notation One (ASN.1). (2002)

[2] APLCs: Daniela Cancila, Tullio Vardanega, AP-Level Containers: a Survival Kit. (ASSERT Technical Note.)

[3] AADL: Architecture Analysis and Design Language. (http://www.aadl.info)

[4] UML: The Unified Modeling Language. (http://www.uml.org)

[5] Lustre: Declarative synchronous language, kernel language of SCADE. (http://www-verimag.imag.fr/~synchron/index.php?page=lang-design)

[6] SCADE: Safety Critical Application Development Environment. (http://www.esterel-technologies.com/products/scade-suite/)

[7] Mathworks: MATLAB/Simulink, a tool for modeling, simulating and analyzing multidomain dynamic systems. (http://www.mathworks.com/products/simulink/)

[8] Jerome Hugues, Laurent Pautet, Khaled Barbaria, Bechir Zalila (ENST), Juan Zamorano, Santiago Uruea, Jos Pulido, Juan Antonio de la Puente (UPM), Stuart D. Fowell (SciSys Ltd): D3.3.2-2: Virtual Machine Architecture Definition. (July 23rd, 2007)

[9] Daniela Cancila, Tullio Vardanega (UPD), Irfan Hamid, Elie Najm (ENST): D3.1.2-1 + D3.1.1-3 An HRT-UML/RCM Interface Grammar for AP-Level Modeling. (November 19th, 2006)

[10] Peter H. Feiler, David P.Gluch, John J. Hudak: The Architecture Analysis and Design Language (AADL): An Introduction. (February 2006)

[11] ObjectGeode: Verilog ObjectGeode, a tool set dedicated to analysis, design, verification and validation. (http://www.spacetools.com/tools4/space/213.htm)

[12] ISO SC22/WG9: Ada Reference Manual. Language and Standard Libraries. Consolidated Standard ISO/IEC 8652:1995(E) with Technical Corrigendum 1 and Amendment 1. (2005)

[13] SDL: Specification and Description Language. (http://www.sdl-forum.org/)

[14] ESA: ASSERT System Family Independent Middleware. System Inputs to the ASSERT Middleware Definition. (March 21st, 2005)

[15] asn1Scc: Semantix Space Certifiable ASN.1 compiler, http://www.semantix.gr/asn1scc/. (2002-2007)

[16] LEON: The LEON processor, a 32-bit synthesisable processor core. (http://www.gaisler.com/leonmain.html)

[17] Olivier Dubuisson: ASN.1 - Communication between heterogeneous systems, http://www.oss.com/asn1/dubuisson.html. (2000)

[18] ITU-T: Rec. X.690, ISO/IEC: ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER). (2002)

[19] ITU-T: Rec. X.691, ISO/IEC: ASN.1 encoding rules: Specification of Packed Encoding Rules (PER). (2002)

[20] ITU-T: Rec. X.693, ISO/IEC: ASN.1 encoding rules: XML Encoding Rules (XER). (2002)