Contents

Chapter 1

Terms and Conventions

The most important general terms, shortcuts and other words defined for the purpose of this document.

VHDL VHDL-AMS

model VHDL-AMS description of system

whole model

- main (top) entity that instantiates other subentities
- compiled main entity integrated with Simulink, according to a context

entity general specification of VHDL-AMS unit and its interface

architecture implementation of the entity, its behaviour

instantiated entity component of the upper entity

quantity analog value of a specified type, scalar or vector

class Java class of a translator framework used for parsed data and persistent objects

OOP Object Oriented Programming, a high level abstraction that encapsulates data and the code together in objects and reuses them

structure C++ OOP data type in developed libraries, generated one per entity

analog analog quantities integrated by DOT attribute

ODE ordinary differential equations

API Application Program Interface

compiler VHDL-AMS-RT translator (parser and generator), and its framework classes

integration libraries uniform C++ constructions, the interface between Simulink and the top entity layer

tools programs that integrate a translated model with integration libraries and Simulink

method Java code encapsulated in object—member function of a class

persistent binary or text representation of objects and their relations—references that can be saved and read from a file

serialized Java synonym for persistent

designer designer of VHDL-AMS models, user of the compiler and integration tools

developer analyst, designer and programmer of the compiler and the tools

RT real time

LRM VHDL-AMS language reference manual

DSP digital signal processor used for fast operations on analog/digital data flow

Format conventions used:

name_extension() JavaCC BNF production with Trans return type and possible parameters, if not specified otherwise. Its 'call' stands for its use—reference from other production.

REAL_VECTOR VHDL keyword or important standard identifier

Chapter 2

Introduction

Engineers need VHDL-AMS RT simulation tool integrated with such a known industrial modeling framework as Simulink. There are powerful simulation environments as, for example, ADVance MS from Mentor Graphics, Corp, and TheHDL and VeriasHDL from Analogy-Avanti, Inc. They have a lot of usefull features. This makes the overhead very long for RT simulation. hAMSter free simulator has a medial support for VHDL-AMS. REMOVE: However, its execution limitations do not fit for RT either.

Instead of a large compiler and own simulation environment, a translator of the most used language subset to C++ is developed. It uses an interface to Simulink. High speed of C++ and Simulink, stability, reliability and well structured API between them suit for a fast and robust RT simulation. Existing Simulink modules and libraries are used and actuated from others. An existing support of A/D cards allows the use of Hardware-In-the-Loop (HIL).

Chapter 3

VHDL-AMS RT

VHDL-AMS (IEEE 1076.1) is **V**ery High Speed Integrated Circuit **H**ardware **D**escription **L**anguage for **A**nalog and **M**ixed **S**ignals. It extends VHDL (IEEE 1076) that describes digital circuits. It has efficient and powerful capabilities for analog and mixed signal models [3]. This extension includes full VHDL. It enhances formal description of electrical, mechanical, hydraulical and other systems and processes.

VHDL-AMS is modular, structurised and extensible. The separated formal interface and implementation supports team work. It is easy to use by non-programmers and no-IT specific engineers. The development process is fast, the model is robust and safe. It defines what system does, instead of how it works. Models are platform independent. Designers can choose from several simulation tools. They profit from a wide range of libraries that function also as documentation and reference for manufactured products.

VHDL-AMS offers a lot of language constructions with powerful expressiveness. The behaviour of modeled systems is very detailed and thus suitable for REAL-TIME simulation.

VHDL-AMS-RT is VHDL-AMS Subset for REAL-TIME [4, Chapter 0]. It eliminates parts that do not fit the requirements of Real Time simulation.

Chapter 4

Brief Review of used Technologies

4.1 Matlab

Matlab stands for **Mat**rix **Lab**oratory, a framework from Mathworks, Inc. It is widely used by engineers, scientists and mathematicians. It has its own programming language with few structural statements, very efficient notation and implementation of vector and matrix operations. Reusable blocks of code can be defined in M file, which is automatically precompiled on its first use. Matlab is available for several platforms.

4.2 Simulink

Extensible modeling and simulation software environment integrated with Matlab. It has a lot of visualisation and debugging possibilities. The user interface is easy to use and intuitive. A large set of toolboxes—standard, customized and 3^{rd} party modules—is used to build models.

A fast and specialized model can be defined as S-Function, a module written in other programming language. It represents one subsystem, model or process, that can contain its own objects, structures, threads and functions. S-Function is implemented as an user's library of specified functions that are called from Simulink on each simulation step and interact together. It can be a binary library compiled from C/C++, Ada and Fortran sources, or

precompiled M file.

Simulink is known, used, stable and fast. It has the interface to A/D cards that are easily connected to any Simulink model and used in RT simulation.

4.3 C and C++

C is a general programming language providing both high level abstraction structures and low level efficient operators. It is ANSI and an industrial standard used for interface specification of extensible software moduls—open systems. The source is easily portable to different platforms with minimal or non changes. The compiled binary code is machine-executed, optimized and fast.

C++ adds a large set of Object Oriented Programming (OOP) features to C, still allowing to write efficient low-level programs. Classes and structs are OOP types that encapsulate data and the code. The object is an instance—a variable of a class or a struct. Member functions are used to access object's variables that can be protected from a direct use by object's environment. The child class can inherit from the parent class, however obtaining an access only to specified subset of its variables and functions. The child can implement or specialise virtual functions defined by the parent.

C++ allows to write a prototyped, easily readable, abstract and efficient program. The features are: classes, overloaded—redefined functions and operators according to argument types, templates—generic constructions (classes, functions and operators) and exceptions. The types are checked more than in C, so the code is safer. Syntax is simple, clean, short and easy to read, inherited by other successful languages.

4.4 Java

Java is a modern OOP programming language defined by Sun Microsystems, Inc. It is used for standalone, Web/Internet and embedded device applications, being an industrial standard for the two latter areas. It is usefull for

tool development, as it is stable, well-tested, free and widely used. It supports UNICODE 16 bit international characters. Its slower speed is not as important as the quality and reliability. A generated platform independent byte code is interpreted and frequently used parts are compiled on the fly by former Just-In-Time (JIT) and today's HotSpot technologies. It can be run, debugged and reused on any platform without any need for source code distribution, which suits the industrial and commercial use. Native functions defined in C/C++ are also possible. Java objects can be used directly from Matlab. All this speeds up the development process, future integration options, safety and robustness of designed applications.

The language is based on C/C++, but much simplier and thus easier to learn and use. It provides a constrained set of OOP focused on a clear and uniform use of objects, reducing possibilities for errornous, hard readable and ambiguous constructions. The developer profits from automatic garbage collecting, language support for multithreading and standard powerful API libraries.

4.5 Parsers

The grammar formally describes the syntax of a language. A low-level lexical specification—tokens are terminals defined by regular expressions. BNF (Backus-Naur Form) productions are rules, that provide higher abstractions. They consist of grouped, optional and repeated referenced—'called' rules and tokens.

Parser (translator, compiler) is a program that accepts inputs of given language. It is created by 'Compiler Compiler' from grammar definition. This must conform to specified restrictions, so parser can be machine-generated. In general, left recursion must be transformed.

Parser reads input, analyzes it and generates AST (Abstract Syntax Tree) for it. There exist different parsing technologies. Several Compiler Compilers, such as lex, yacc, bison, PCCTS/ANTLR, JavaCC, are used. They generate parsers—stack machines that accept sentences of a given language, translators—substitution engines, and compilers to the binary code.

4.6 JavaCC and Jjtree

JavaCC is a state-of-art parser/translator generator. The source JJ file is a grammar specification consisting of tokens, BNF and other definitions. Arguments can be passed to productions and returned values can be assigned to variables. Standard Java code can be added around BNF expansions inside *{}* pair, and it is executed when those rules are parsed. This code can be used to execute—interpret or translate/transform parsed source. It is not called during lookahead evaluation (when decision is made at choice points). Basic grammar defines:

parser class Java definition of the parser class that is extended by JavaCC generated code for tokens and productions

parser options case sensitivity for tokens, default lookahead (number of tokens to look ahead at a parsing choice), debugging and UNICODE characters

skip characters to be ignored, white space

tokens Terminal regular expressions. Less complex then BNF, they can consist of other tokens. Character lists and intervals can be used. Tokens are defined separately for different lexical states that function as a simple low-level parser state machine. Special tokens are unimportant for parsing, they are usually comments.

productions BNF has symbolical and more structured rules than token definitions. BNF's structure is that of *expansion-return-type name-of-production (optional-parameter-list) : {java-block} {expansion-rules}.* Java block is usually used to define local variables. The production can optionally declare and throw exceptions. Expansion rules are grouped in parentheses:

[] or ()? optional appearance

()* zero or more repetitions

()+ one or more repetitions

other local syntax/semantical lookahead specifications, JavaCode productions and JavaCC API give full control over the parsing process

JavaCC generates (but does not override) several Java classes. The object of the parser class represents the whole translator. It is attached to the input stream, possibly from a file, that is parsed by a chosen root production method. Its output (tranformed input) and byside effects (generated files...) are the result of translation. It can be embedded in other systems and gains all the advantages of Java program. Its classes, usually Token, ErrorHandler and TokenManager, can be customized. Tool Jjdoc generates very usefull HTML documentation of grammar BNF tree. See [9].

Jjtree is JavaCC extension, that generates JavaCC grammar and AST tree classes from JJT definition. Its grammar source has few additional parts to JavaCC grammar. Generated AST classes can be customized and can use additional general API to access parsed ordered nodes of any parsed rule. This can be used for context dependent and automated use of parsed input. AST classes are defined separately from JJT source and need to be synchronized with the changes to JJT grammar structure. Because nodes are indexed and optional/repeated appearance must be checked 'manually', it does not fit for large and complicated grammars.

Chapter 5

State of the Art in AMS and RT Simulation

AMS Simulation of abstract models and virtual prototypes speeds up the product development and finds possible problems. Ability to simulate them together with real hardware gains designer faster, safer and accurate results. This places hard restrictions on simulator.

5.1 Analogy-Avanti, Inc.

TheHDL and single-kernel environment VeriasHDL support modeling of MAST, SPICE, VHDL-AMS and Verilog-AMS languages. Their products are used in Automotive, Communications, Military, Aircraft and Power Electronics industries.

There are several ways to execute the mixed signal simulation: glued connects analog and logic simulator, native has one simulator, the native version connected to logic simulator is mixed [6]. Those configurations have their own synchronization and logical event queue approaches. The most known is the patented Calaveras algorithm of native simulator. It is a robust industrial simulation framework.

5.2 Mentor Graphics, Corp.

Mentor is focused on the world of digital integrated circuits. Simulators ADVance MS (ADMS), Eldo and Eldo RF are used for mixed signals. ADMS supports systems defined in VHDL-AMS, Verilog-AMS, VHDL with Vital, Verilog, SPICE and C. Fast and efficient ModelSim solver is used for the digital part, and analog portions are simulated by one of several algorithms.

5.3 hAMSter

Simulator HAMSTER is for MS Windows only. It has its own editor, simulation environment, limited ODE and unlinear solver. Only scalar quantities are allowed. Restricted processes are supported. Integer and enumeration types, several predefined attributes, libraries and several packages can be used. It is focused on the analog part.

Chapter 6

Research Problem Statement

6.1 Aim

- Define most usefull subset within VHDL-AMS RT with focus on analog and mixed signals, suitable for the implementation by this project
- Analyze strategies
  - integrating VHDL-AMS model with Simulink, the interface between Simulink and the main entity
  - connecting inner VHDL entities to each other and handling their analog signal flow to Simulink, that passes by the main entity
  - transforming VHDL-AMS code to C++ code
  - leaving semantical checks to C++ compiler up to maximal level—no symbol table, type control...
  - language mapping—definitions of C++ constructions for corresponding VHDL-AMS statements
  - processing and implementation of special features and language constructions
  - extensibility in the future
- Create the framework

- general interface between the entity and its environment
- libraries and interface between the main entity layer and Simulink
- entity translator to C++
- model integration tool that generates integration layer of the top entity

6.2 Need

Software models, formal specification and prototypes increase the productivity of development, design and manufacturing processes. The industry follows VHDL-AMS standard to get maximal compatibility of product descriptions and specifications. Users can choose from more simulation environments for given standard, one that suits their needs.

RT simulation of models connected to real existing systems is a necessity today. Hardware-In-The-Loop (HIL) simulation shows hidden weaknesses, increases the development speed, quality, robustness and safety of the product. It is not sufficient to test analog and digital parts separately, because they influence each other in unpredictable ways. This is critical in the aircraft, automotive, electronics, communications and other industries.

There is presently no VHDL-AMS-RT Simulator on the market. There is a scientific and industrial need for such a framework that offers possibilities to use existing models.

6.3 Value

The project gives designers a possibility to test simple mixed signal models in the real time. Simulated model is integrated to Simulink, so it uses vectors, efficient configurable ODE solver, simulation control options and commands, and an intuitive user-friendly interface. Inputs and outputs of the model can be connected to standard source/display items, any other models and external hardware interfaces as A/D, D/A converters. The model is translated to C++ and compiled with Simulink libraries, or processed to C language

and compiled for DSP. The simulation is very fast and usable in RT. The designer uses all the power from Matlab and Simulink.

This pilot project finds a possible way, the strengths and problems of a small-scale VHDL-AMS-RT implementation. The proposed interface and architecture are open, modular and easily extensible in chosen limits with the focus on mixed signals. A support for complex numbers and complex vectors, processes, signals and other control flow statements can be added. A connection layer to other simulator environments can be defined by standard IPC or shared library communication. The translator leaves lexical checking to target C++ compiler; in this way the development process is more focused on the implementation of larger VHDL-AMS subset with its special features. Digital operations are easy to add, some of them are implemented, or the interface can be defined to existing digital simulators.

Chapter 7

Sources

7.1 VHDL-AMS Grammar from Grimm

There are pure VHDL grammars for Yacc, that do not include AMS extensions. The available VHDL-AMS grammar was from Christopher Grimm for Jjtree. It confirms original LRM specification, using the same production names. It is clean, easy to use and to compare with LRM. It does some symbolical error checking and syntax error recovery of blocks. Productions that require semantical lookahead based on symbol tables and local context, are not used during the parsing. Consequently, respective VHDL-AMS code parts are parsed by different, syntactically equivalent rules. See [1].

7.2 VHDL-AMS translator from Vince & Dudáš

This framework generates one DSC file per entity and one M S-function per each of its instantiations. DSC is a commented text-format description that keeps an information about the entity's interface. One Simulink MDL file is created, and it encapsulates all entities as separate S-functions, connected together by multiplexors and demultiplexors. This powerful and structured hierarchy copies a tree of instantiated entities.

Thus the whole model is reusable and easy to debug in Simulink, because the user can access and set any signal between entities. Instantiation

dependencies are checked recursively, required entities are processed first.

The translator is based on Grimm's JJT grammar. The transformation is performed by methods added to Jjtree generated AST classes. Statements are collected and grouped as: generics, REAL ports and internal quantities, components with generic and port maps, derivations and other equations. Those lists are saved to DSC. Structured M S-function with uniform skeleton is generated per each entity. It defines other functions for different simulation tasks. Additional Simulink ports are defined to instantiated entities. MdlInitializeSizes sets number of ports, states and simulation options. The execution part (MdlOutputs) receives integrated values from Simulink, does equations and returns the output ports to Simulink. MdlDerivatives evaluates derivations directly in assignment to the vector that is sent to Simulink. Actual generics are set in the two latter functions. Three other trivial or empty functions are generated. Equations—simple_simultaneous_statements with basic mathematical operators—are the only supported execution statements. See [2].

Chapter 8

Targets

8.1 Simulink and RT

> The required calculation time for each time step must stay below some predictable maximum time [4, Chapter 0].

Simulink calls several user-implemented C functions, on the initialization, each step and the end of the simulation. It offers several fixed-step ODE solution methods that fit for RT. Variable-step solvers are used to get fast and more accurate results in the off-line simulation.

Even RTW (Real-Time Workshop) is designed for RT simulation, it does not suit our purpose. It uses just a subset of Simulink interface. Especially, RTW allows only vector inputs/outputs of width known before the model is initialized. Our solution needs an access to full Simulink API. However, this does not restrict the generated model from being used in RT in any way.

8.2 C++ MEX

A model code translated to C++ is fast, robust, of high structurality and portability. It can be used in other frameworks than Simulink with only minimal changes. Such a model uses 3^{rd} party and customer's modules accessing them through their interfaces defined in C/C++. The code is preprocessed

to C and compiled for DSP, because generally there are no C++ compilers for them.

The support for multifunction AD/DA and IO card Lab-PC+ from National Instruments [7] can be added. C++ drivers from National Instruments Data aquisition utilities NI-DAQ [8] for Borland C++ and Microsoft/Visual C++ are available.

Chapter 9

Strategy analysis

After general decisions, specific issues were analyzed during the development of the implementation. Those two processes were tied and difficult to separate. Analysis depend on grammar and behaviour specification. The structure of grammar that makes a skeleton and is a part of the translator was changed and restricted during the development. Other behaviour constraints and relaxations were also defined according to existing and future implementation needs. Thus descriptions of respective production rules are here instead of presenting them separately.

9.1 Translation methodology

9.1.1 Traversing AST

This was tried by [2]. Jjtree grammar is used. Generated AST classes are customized by those methods that perform translation tasks.

The main problem is the separation of JJT grammar and AST code. The general access to any parsed node is realised by API functions and node indices rather than by their names. They must be retyped and in this way give a possibility for runtime errors that are difficult to find. The checking of optional and repeated nodes must be done by Java conditional and loop commands. That all represents a long trivial code, that only copies existing grammar structure and depends on it. The developer is a counter

and source code comparator. If grammar is reorganized, respective AST code must be searched for and changes made. This is much more complicated in the presence of a lot (186) AST classes defined in their own files. The maintenance of such a project becomes a nightmare.

Such an approach creates an illusion that it supports a team work. The source is distributed to number of different files that are not updated automatically on the grammar change. Ties to JJT grammar must be checked manually. It is not flexible and incremental development is slow. Even if grammar is already well structured, optimized and unchanged in the future, little 'active' part and big skeleton does not seem the best solution for large projects. A lot of trivial—waste code must be written.

Problems could be reduced, but not eliminated by the use of JTB, another AST possibility for JavaCC. It offers type-safe access to parsed nodes by means of symbols. JTB definitions depend less on grammar changes, however, it is still distributed in a net of files. It is usefull if grammar is in the final form.

The author added a translation of simultaneous_if_statement to this framework using Trans container. The combination of AST and collecting—separating parsed statements solves only few language features. It is not general, can not grow lineary and is very limited to extend. None of this code is used in presented translator to C++. The approach was analyzed and the results influenced decisions and solutions of this project. The idea of description files is enhanced by the use of Java serialization.

9.1.2 Substitution production rules

The decision not to use Jjtree was chosen. JavaCC production rules do the translation, substitution and integration tasks. This is what JavaCC productions suit for—to transform and pass up (return) an usefull representation of parsed input. Tasks of former AST classes can be easily and automatically done by Java block inside {}. It is called directly from JavaCC rule at the point where it is parsed. This code can be in grouped, optional and repeated statements and is executed each time the rule accepts an input. Results of

'called' productions are assigned to local Java variables in the type-safe way. The symbolical use of parsed nodes and the automated execution of a specified code enhance the simplicity and quality assurance. The translator source is very expressive and resembles a high-level data-flow driven program.

BNF methods generate and return a container object capable to hold concatenated strings, tokens or any printable objects. This Java container class, called Trans, is created to hold any printable objects, possibly nested Trans. The content is printed with a default or user-specified indentation. This by-side product is fully independent from this project and JavaCC, and it can be used for format transformators, macro and template engines.

An easier and safer translator development process represents an important improvement. The translator code is compact, the main parts in one JJ file. Although most of the code is centralized, it fits for team work (see the documentation of Trans). No dependency checks between grammar rules and generation methods are needed when some of them were changed, as they are all at one place. The code is type safe and very easy to modify. The development control was more efficient and less painfull. The necessary semantical checking for ports and generics is done by passed arguments and returned objects of specialized Java classes from selected productions. Those customized objects carry an aditional information used for checks performed.

9.1.3 Main grammar changes

The translator is based on Grimm's grammar. All Jjtree-specific code is removed, so it's a pure JavaCC grammar. A lot of semantical grammar production rules—descriptive only when used without symbol tables—are skipped and removed, split or merged. 334 original productions are transformed to 234. The grammar is more compact for our purpose. The translator source code is reduced from JJT source, 186 decentralized AST and 4 JJT additional classes (generated in unique files) to JJ file and 9 framework classes. Other 2 of 4 classes originally defined by Grimm are used. This dramatically increases the development speed. The startup and the execution of translation process are faster.

Because VHDL-AMS is case unsensitive, and C/C++ case sensitive, all identifiers are transformed to the lowercase. C++ layer and integration library identifiers consist of the uppercase, which prevents the conflict with the user specified VHDL-AMS code. C++ definitions used directly by VHDL-AMS code are in the lowercase. They're in a different case in this document only to be highlighted, as 'REAL_VECTOR'.

9.2 Development process

JavaCC source indentation and structuring prevents hard-to-find JavaCC errors . Those are often reported at wrong places and only one at time. The time-consuming JavaCC processing must be repeated. BNF HTML documentation allows fast orientation in the grammar.

If the translation code breaks the parsing process, unusual runtime parser errors raise for correct VHD input. They are all eliminated by use of simple rules. Generally, created container object must be returned from the main block of the production, or from its end. See the documentation of Trans.

The compilation of Java classes finds other errors uncaught by JavaCC. The report and the generated compiler class resemble JJ code, so more errors are fixed at the same time.

Most of BNF productions do a trivial transformation from VHDL to C++. Complicated rules and special features have by-side effects, such as collecting DOTed quantities, ports and generics and writing to description files. Framework persistent classes are used. They carry symbolical information, that is read during the translation and the integration of the upper entity. The semantical checking that can not be left for C++ is done.

C++ general entity interface is defined. It is implemented by the translated entity code in H files. The integration CPP file is generated for the top entity, that defines functions for construction and destruction of a model object, its initalisation and simulation. They use libraries—C++ layer interacting with Simulink.

Framework classes and libraries are first tested independently from the translator, then together when translating and simulating the model.

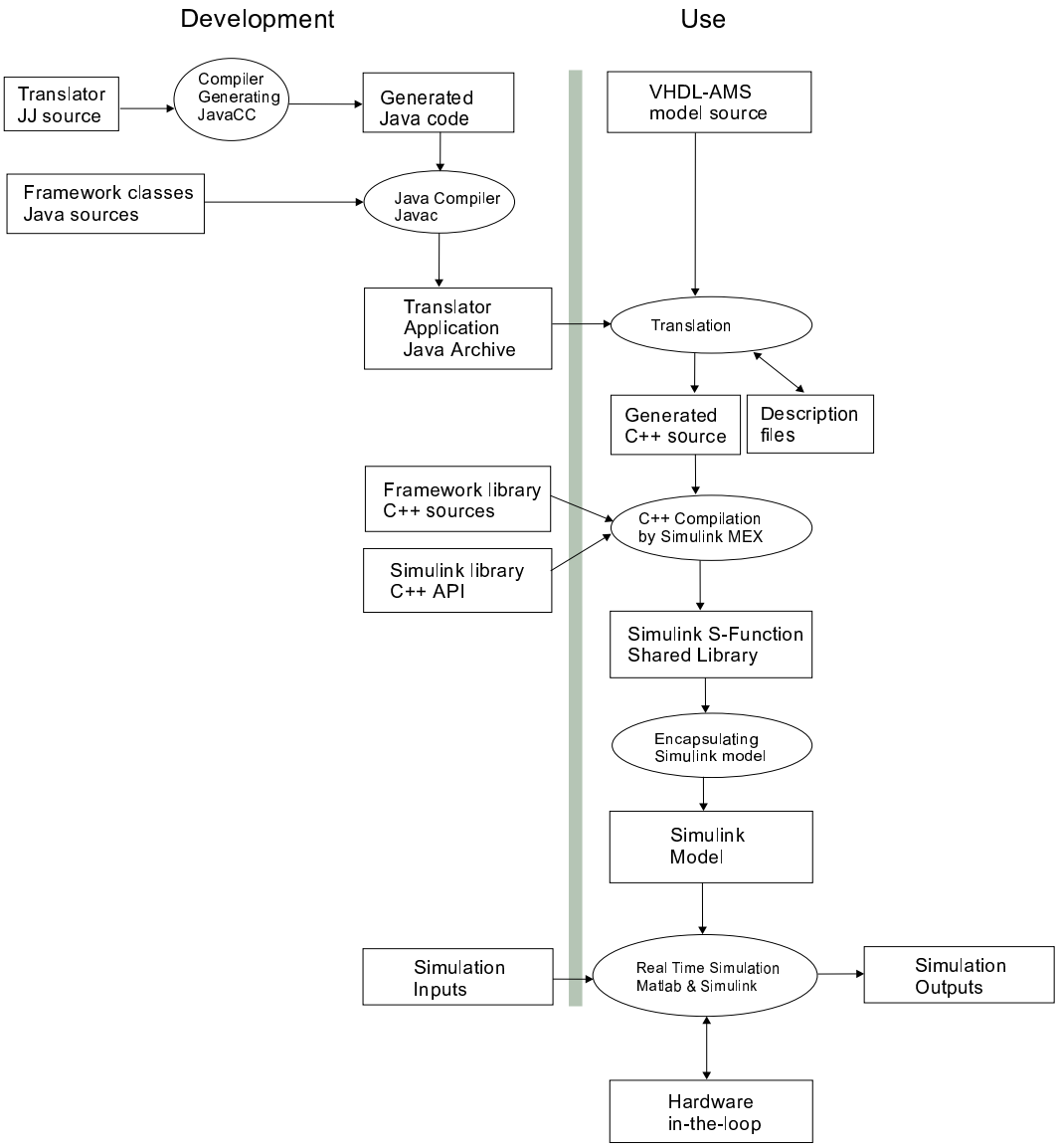

Figure 9.1: Development and use process

9.3 Integration with Simulink

9.3.1 Open box

Framework [2] generates well structured MDL model that copies a tree of instantiated entities with their mapped ports. This gives the designer very

usefull possibilities (chapt. 7.2). Unique M S-Function generated for each instantiated entity has direct access to ODE solver with its own list of derivations and integrals. That fits for VHDL-AMS feature of DOTed quantities specified 'directly on place', in the architecture behaviour definition.

The price to be payed is the complexity of model generation process and the slow execution. If the designer had a lot of small, structured reusable entities, most of the simulation time would be spared for the signal passing between entities and Simulink 'channels'.

Creating one C++ S-function for each entity, all of them connected on Simulink side, was not much more efficient. The small C++ part could be of the same speed as M S-functions, that are pre-compiled by Matlab to a faster format on the first use.

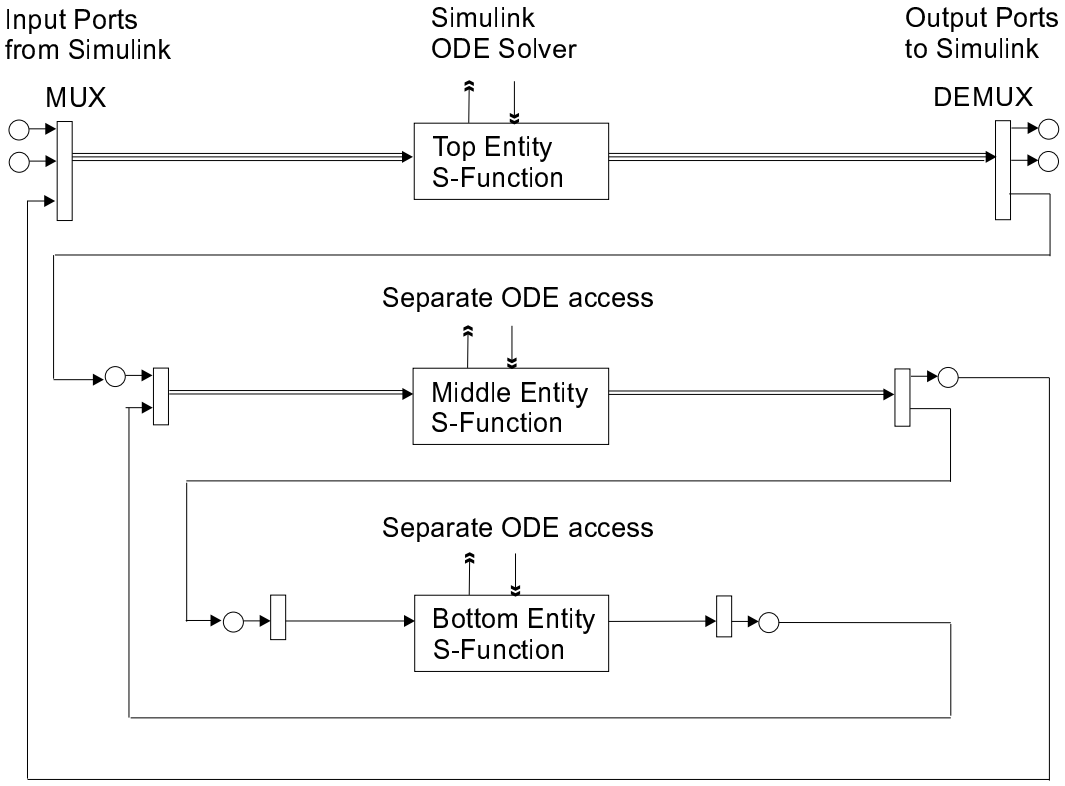

Figure 9.2: Several S-Functions in one hierarchical Simulink model

9.3.2 Black box

The translator generates one C++ S-function for the whole model—the main entity and all its subentities. C++ code created for each entity uses a simple interface, is automatically indented and well structured. S-function handling code called by Simulink is separated in another, stable layer in file model.cpp. Thus the translation result is better to be read than M code and the user can find semantical and other errors easily. C++ S-function gain developers full Simulink API, not available to M S-functions. Analyzes, programming, tunning, integration and future development are safer, easier and faster. A high simulation speed is obtained.

Such a separation was impossible by M code, because it does not have comparable structural possibilities. Matlab can not pass parameters to M functions by reference, global variables must be redeclared in the functions that use them. OOP encapsulation of the code by objects and operator overloading is not possible. This makes its extensible use in a large project very difficult.

The designer can debug an inner entity separately by making a different model for it. An additional debugging support can be implemented by C/C++ assertion macros and functions that are called directly from VHDL-AMS code. Those can be automatically filtered or commented after tests, when in the production use.

Dependencies between entities are not checked. The translation of entities must be processed starting from down to the main entity. It is left up to the user or an automation tool as 'make'. Simulink model (MDL file) is not generated, because it is platform and version specific.

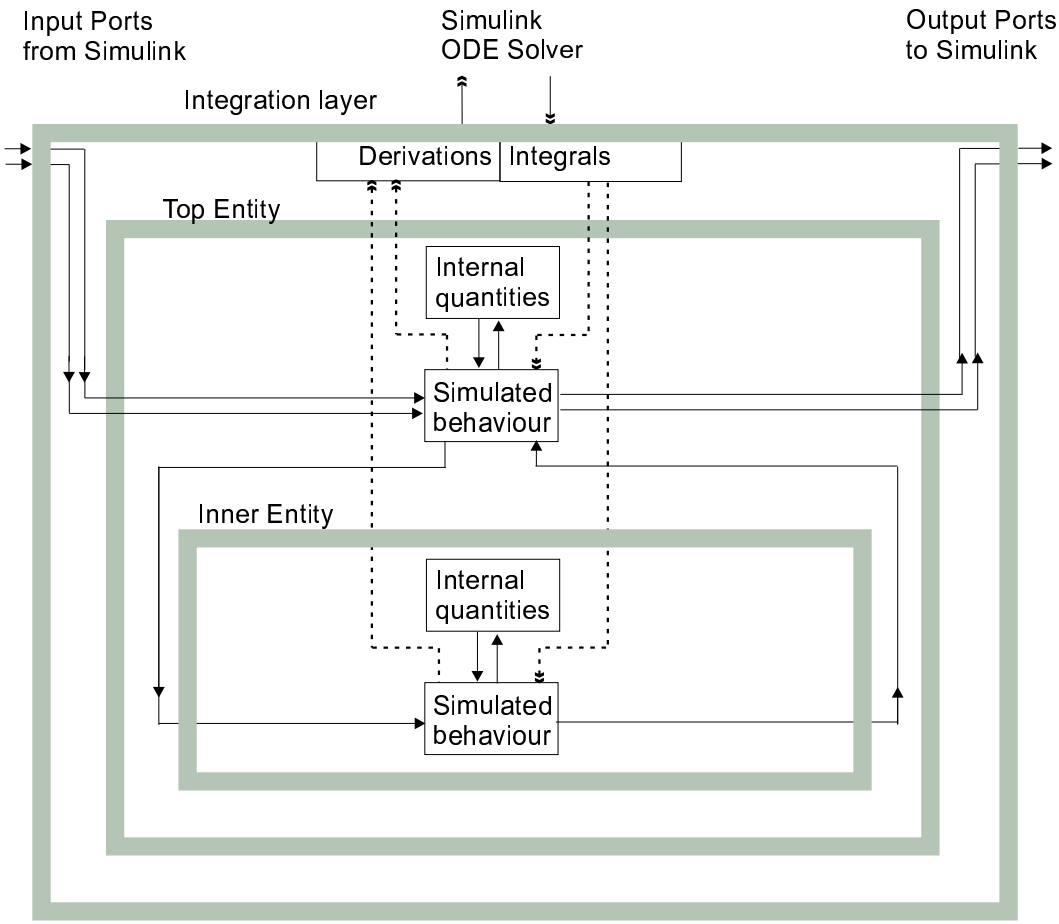

Figure 9.3: One S-Function for the whole model

9.4 General interface of architecture

The entity as an unit of a larger system interacts with its environment in a clean way by generics and ports only. It can be instantiated—owned and used by other, 'upper' entity. The top, main entity represents the whole system and passes ports to/from simulation environment—Simulink.

The entity can integrate quantities (internal or from the port) directly in the code, distributed 'on place'. Simulink requires all integrated values and their derivations to be passed in a special way together in the same time. Those are 'collected' from the source and 'registered' at the simulation initialization. Then they are passed to and from Matlab on each simulation

step.

Uniform MatlabInterface allows a connection to other entities and librares that communicate to Simulink. It defines a general way to send and receive ports, derivations and analog values and their widths to/from Simulink. It is implemented by types REAL and REAL_VECTOR.

9.4.1 Connection to Simulink

The connection to Simulink is used for ports and analog quantities. Nonintegrated analog and digital or discrete signals are implemented on C++ side. The number of analog quantities (states) and their default values are sent to Simulink during the initialization. Derivations and output ports are sent to Simulink, integrated values and input ports are read at each simulation step.

No more than one use of the same whole model is allowed in Simulink. That is because of static variables and fixed generated S-function's name 'model'. Otherwise an unique file model.cpp had to be generated. The designer can encapsulate several instantiated entities in the simple higher entity that becomes a new model.

S-Function is implemented as a dynamic library. If already used by running Matlab and Simulink, those must exit before the model is recompiled. Multi-step methods can cause problems (see chapter 9.7).

9.4.2 Hierarchical entities and generics

Both actual mapped generics and ports of the instantiated entity—only names or the order and the number are checked; type-checking is left for C++ compiler. Port and generic definitions (types, names, modes, default values, widths) are collected in lists and saved in ENT file at entity_declaration(). Class Port is used. This information is read and used if the entity is instantiated or integrated. Default generics are used if actual values are omitted. Named ports and generics at element_association() are processed correctly. Actual values of generics are set in a call to its constructor when the entity object is created in the upper entity or the integration layer.

To keep the whole approach simple, there is no posibility to set generics from Simulink. The main entity can not have any generics, not even with default values. The designer can instantiate it in the trivial higher entity that specifies and passes generic values and connects the ports.

9.4.3 Ports

Input vector ports of main the entity need width specification. Simulink allows dynamically sized ports, but restricts the process of the parameter exchange at the initialization of S-function. It must know the number of analog states, before it passes the actual width of input ports. Since analog quantity vectors can be assigned from input ports, the translated model needs their size first. Therefore all port widths of the main entity must be specified. However, instantiated entities can have unconstrained input and output ports, which size is set in the runtime from the owner entity. Unconstrained internal quantity vectors must be set before use.

9.5 Vectors

Vector quantities and ports are implemented by C++ template—generic structure VeCtor<class T>. It has constructors, assignment operators and element accessor—operator (*index*). In this way a vector of any element type can be easily defined in C++ and used by models. Standard type REAL_VECTOR has $+ \; - \; * \; /$ operators that perform those operations on its items, where one parameter can be scalar REAL. An underlying array is always allocated dynamically in the vector constructor (if width is known) or on the first assignment (if unconstrained).

VeCtor is a subclass of MatlabInterface and implements its functions. Items of the vector are sent to/from Simulink ports. DOTed vectors have their derivations sent to, and integrals received/sent from Simulink as a whole. Derivations and values of REAL_VECTOR are implemented as separated vectors because of the efficiency. Those derivations can be accessed only in the way *vector_one'DOT=vector_two*, no: *vector_one'DOT(0)=vector_two(0)*

or *vector_one(0)'DOT=vector_two(0).*

General REAL_VECTOR and REAL_VECTOR(size) declarations are allowed only (and similar for any defined vector types). The latter is defined by subtype_indication() that sets a flag InSubtypeIndication. It calls name() and name_extension() that handle parsed data in a special way. The vector size is assigned to variable TypeParameters and then carried in the object of class Port, rather then passed to C++ as a part of the vector type. It becomes a default or the first parameter to the vector constructor. Vectors of unspecified widths have a memory allocated on the assignment. Sizes are always checked in runtime and shorter vectors can be assigned to longer ones.

The element access parenthesis operator (*index*) uses name() and name_extension(). The parsed index value is put to generated C++ code as it was.

9.5.1 Vector literals

Vector constant—literal is a special form of the aggregate statement: *(expr0, expr1, ..., exprN).* This is translated to C++ code *(VeCt(N,expr0),expr1,...,exprN).* The aggregates are used in other constructions according to their context, as literals of composite types—records, or their parts—subaggregates. Those are not implementable by this framework, because we do not check types of expressions and the context of their use, possibly nested.

Translated C++ code calls a template function *VeCt(Size, FirsElement)* that creates an empty vector with elements of a specified type, size and sets its first item. Then the operator comma is called that adds successive items to the vector, the index of the last added item is remembered between those calls. Enclosing parenthesis are required because of a low precedence of the comma operator.

The access operator (*index*) is not distinguished from other uses of parenthesis by syntax rules. This involves a need of symbolical tables for the detection of function and procedure calls.

9.5.2 Cached vectors

Nested operations produce intermediary results, new vectors. They are not needed anymore after the evaluation of the whole expression. A repeated dynamic allocation and disposion is time-consuming and can not be used in RT.

Local arrays reserved on the stack in a very fast way can be used only directly in the generated function go(...), or in a macro. They can not be returned from other functions or operators to go(...), because the reserved stack is not valid later. This restricts the use of C++ structural possibilities and involves an analysis of the evaluation tree.

The chosen way is to dynamically allocate intermediary arrays on the first need, and then reuse them. They are returned from functions and overloaded operators. A queue of unused 'cached' vectors of possibly different widths is maintained. They are deleted at the model destruction. This keeps the whole approach simple and gains from C++ high level abstractions. Cached vectors are defined by template TempVector<class T>.

The first pass that involves the allocation is much longer. However, it is called at Simulink initialization before the first simulation cycle. More cached arrays can be used later, in other code branch—at conditional IF, CASE and BREAK statements. Those are allocated when needed, thus delaying the simulation step. An unimplemented solution is to duplicate the list of cached vectors after the initialization pass. This should make the reserve high enough for demand peaks.

9.6 Collecting DOTs and ports

The derivation of quantity—for integration purpose only [4]—is parsed by name() and name_extension(). Those are used in other ways, so DOT is checked and handled specially. 'Exotic' use of name_extension(), as subtype_declaration(), are limited. *Quantity'DOT* is transformed to C++ code *DOT(Quantity).* This is the function defined for types REAL and REAL_VECTOR.

The entity defines analog and discrete quantities together with no distinction. Their behaviour is known from the body of architecture. Scalar and vector internal quantities and output ports can be changed by DOT. Simulink gives a limited control over analog values and their derivations. S-function is required to send and receive them in one vector of the length known at a model initialization time. However, DOTs are specified directly at the code. Thus they are 'collected' by the parser and registred in the constructor of architecture. Their list is maintained during the simulation, containing integrated internal and port quantities from all instantiated entities.

When derivations are passed to Simulink, they are zeroed. That is because in the next simulation step, a different branch of the code can be executed (IF, CASE, BREAK statements) and it can set those analog values directly, rather than integrate them.

Only the first order DOTs are allowed, according to [4]. DOTs of record type elements, vector elements were translated in an experimental version. Higher DOTs were transformed to form DOT(Quantity, Order). They are not implemented because of VHDL-AMS grammar ambiguity and complexity. Rather than those, vector literals and the vector item access operator are implemented.

Integrated entity ports can not be registred in the constructor of this entity, because they are not its part—they are only passed as arguments to entity's go(..) function. Class Architecture keeps the list of DOTed ports, saved in ARC file. They are registered in its owner—the upper entity or the integration layer, or passed up again.

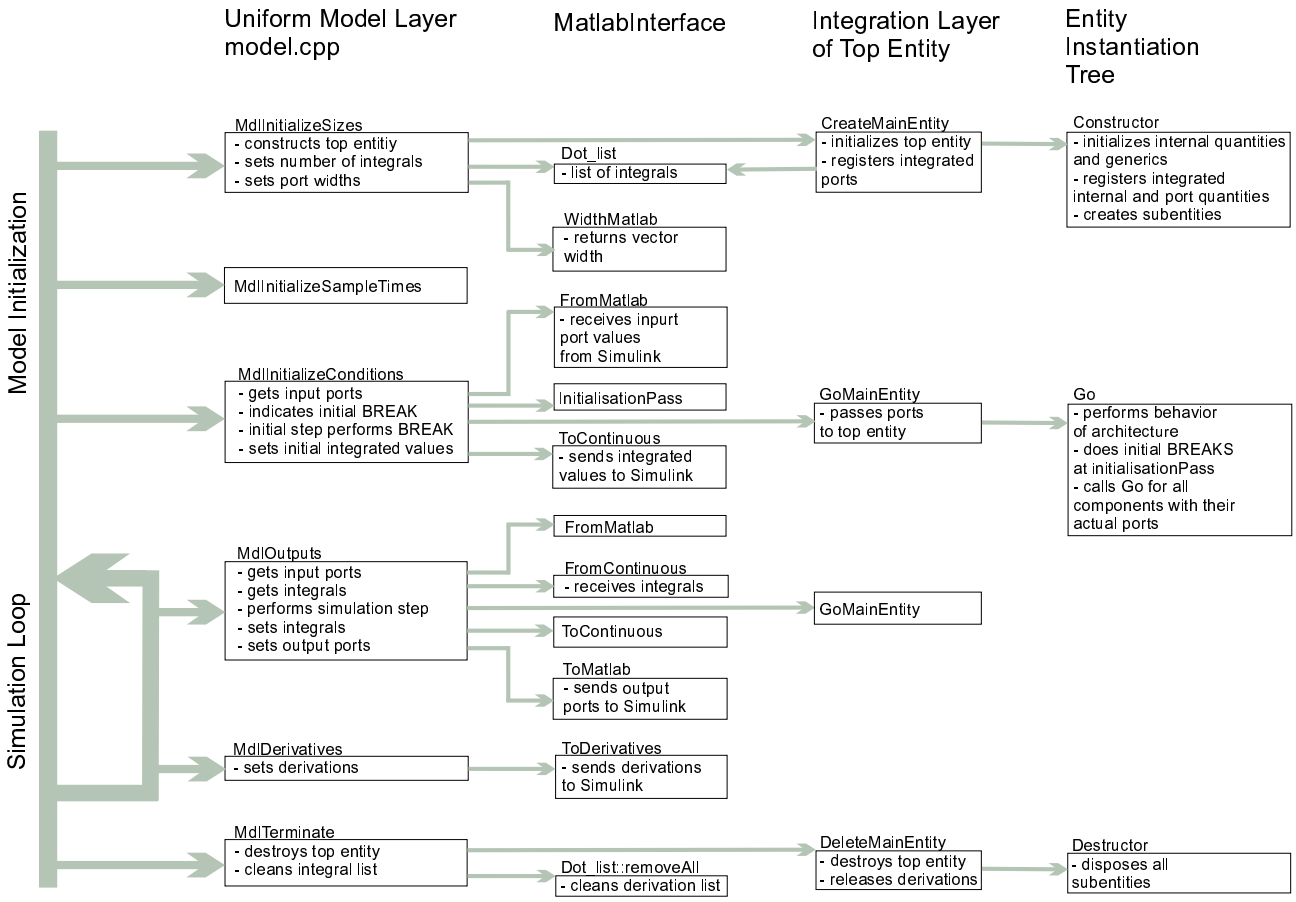

Figure 9.4: Simulation process

9.7 BREAK

Initialization BREAK—without sensitivity_clause()—sets integrated analog values on the initial call to the entity's go(..) function. Input ports are passed from Simulink already and can be used, as well as generics.

Break statements without a break list only indicate an augmentation—a discontinuity of analog signals. They are accepted but not translated, because this is not required in our approach. They could interact with multistep ODE solvers to set the new analog values and prevent the integration approximation and unstability. Simulink API does not allow this.

Conditional breaks that have a break list of *quantity=>value* pairs are

translated to C++ *if()* statement. Assignments are performed when a condition occures.

9.8 Architecture

Generics and ports are saved to file EntityName.ENT at entity_declaration(). This is used when architectures and instantiated entities are translated. No C++ code is generated for an entity. At architecture_body(), the architecture name is checked (if it appears after END token). A header file EntityArchitecture.H is generated and it includes H files of all components. Library_unit() cleares lists of DOTs, generics, ports, quantities and components. Those are filled during the parsing entity/architecture, checked and reported to user.

There is one C++ structure generated per each architecture. An uniform skeleton is used, parts are separated and commented, the transformed code is easy to read. The architecture has parts that are performed in different tasks. C++ constructions generated for them are called separately according to the simulation process execution flow.

The structure has quantities, generics, instantiated entities and a flag InitializationPass as its member variables. They are initialized by constructor's parameters or their default values outside of the constructor's body. C++ default argument values are not used because of VHDL-AMS' enhanced mapping mechanism. Ordered, named and default actual values are compared to their formal definitions from ENT description. Unconnected and wrong connected ports/generics and wrong number of ports and generics are checked and reported.

The main initialization is processed by mdlInitializeSizes(S) that calls GoMainEntity(). A tree of instantiated entities is constructed and their internal and port quantities are set to default values or zero.

The constructor's body sets InitializationPass to true and registers analog quantities and components' ports (that are not mapped to the port of this entity) by function integrated(). Those were collected from architecture_statement_part and read from ARC descriptions of instantiated enti-

ties. DOTed ports of this entity are saved to EntityArchitecture.ARC file. Unknown DOTed quantities are reported.

Function void go(..) with ports as arguments is generated. OUT and INOUT ports are passed by a reference. IN ports are passed by a value. It calls go(..) for all components with their actual ports first. Then translated architecture_statement_part() appears. It executes initial breaks on its first pass only, called from mdlInitializeConditions(S). The true value of variable InitializationPass indicates this first execution and InitializationPass is set to false at the end of go(..). Then initial values of all integrated quantities are sent to Simulink. go(..) is called in every simulation cycle and performs behaviour of the architecture.

9.9 Trivial statements

- Algebraic, relational and logical operators are transformed to their C++ equivalents.
- Expression() checks whether NAND and NOR are in a sequence, which is not allowed.
- Simple_simultaneous_statement is transformed to C++ assignment.
- IF .. USE .. ELSIF .. ELSIF.. ELSE .. END USE
- Only FOR version of generate_statement is implemented, translated to C++ for(..) loop. Thus the block declarative part can not be used. The iteration identifier is of INTEGER type and with nonegative values only, that fits for the use with our implementation of vectors.
- Quantity'ABOVE(Value) is transformed to C++ function Above(Quantity, Value).
- Vector'LENGTH, Vector'HIGH, Vector'RIGHT, Vector'LOW, Vector'LEFT. They are transformed to their respective C++ functions, defined for one dimension vectors only.

Chapter 10

Limits and possibilities

The subset depends on implementation approach restrictions. Main and general implemented parts were chosen during the analyzis at the beginning. Other constraints appeared at the development process and next 'local' decisions. Also some relaxations to VHDL-AMS-RT are available. For details, see [4], implementation description, translation output and Vhdl.jj source. For supported language features, see Appendix C.

10.1 Difficult tasks

All those tasks are difficult to implement by the chosen approach. The use of symbol tables could make some of them possible.

Algebraic loop detection It requires symbol tables.

Matrices Through Matlab works perfectly with matrices, there is no option to pass them to and from Simulink. Operator () is transformed to VeCtor constructing method. This is unefficient for matrices.

Aggregates Operator () is allowed for vectors only, unless symbol tables are used. Different possible scalar, vector, record types and contexts need complicated expression tree analysis.

10.2 Easy extensions

Libraries and Packages can be implemented by included C++ headers. Standard STD package and its NOW function can return Simulink the simulation time.

Operator redefinitions can be defined in C++ as overloaded operators

Type definitions can be transformed to C++ struct or class

10.2.1 Processes, signals, delay mechanism

What VHDL-AMS calls processes, are threads in IT terminology. They share the same data (address space), and have a parallel execution flow. They are not implemented because they work in existing VHDL simulators, not specific to VHDL-AMS RT.

The transformation to Simulink model and using its possibilities for discrete signals can be applied. This involves the separation of analog and discrete values. Registration, passing, checking of those signals and Simulink restrictions involve more complicated constructions than MatlabInterface.

MS Windows libraries have Win32 thread API, not portable to other systems. Hooks, timers, event logging are available. There is a good support for external devices on Windows platform. Process.h, stdef.h with *__beginthread (thread_code.....)* and *__endthread()* are used.

Unix systems use POSIX.1 threads (ISO/IEC 9945-1:1996) with mutex objects for locking and synchronization. Those are portable except Win32 platform. Linux has two implementations, 'green' lightweight threads that suit for high-computation applications. Native threads, processes with the same address space, are efficient for high I/O. Special HW as A/D cards is less supported.

DSPs have a special process with restricted paralelism.

The wait statement introduces a list of remaining times or deadlines. More efficient C++ static local variables defined in go(..) function can not be used for it because one entity can be instantiated several times. New struc-

ture for each instantiated entity as in [2] solves this problem. Delayed signals can be implemented by buffers defined in C++, faster then on Simulink side.

10.3 Relaxations to VHDL-AMS RT

Functions and procedures that are easy to implement can have side effects. Simultaneous_if_statement has independent number of substatements in each of its branches. See [4, 0.5.2].

Chapter 11

Conclusion

The presented solution is easily extensible within boundaries given by the chosen approach. Most of usefull VHDL-AMS behavioral statements can be implemented by a simple transformation to C++ code. Some structural parts are easy to add. A lot of 'exotic' features can not be used, because of the wide and ambiguous syntax.

A support for other simulation environments and ODE solvers than Simulink is possible. General MatlabInterface is not hard dependant on Simulink and can be changed to interact with any modular API. This could relax restrictions on the initalisation process and allow unconstrained ports of the main entity.

The language gives the designer a large set of possibilities. 'What you write is what you mean' approach allows a short code that uses several structural and behavioral features. The feature of DOTed quantities specified 'directly on place of use' hides the necessary processing. This is hard to implement in an efficient way. The context-dependency and the ambiguous syntax are much bigger problems then the paralel behaviour. The user can not remember all of those constructions and probably uses just the preferred subset. However, a correct implementation must accept and process all the possibilities. The question is whether it is worth. When reading the source, one must look at different parts to check whether the symbol is a function, variable... Even simplier languages, based on Pascal and C, have more pos-

sibilities to distinguish a different use. Unparameterized functions must be called with () pair, the array access is by []. Thus the source code is easier to read, complain about errors and translate.

The incremental increasing of language features makes an illusion that it is powerfull. It teaches What To Do, instead of How To Do. Industrial standards as C/C++ and Java make profit because of clean contract—simple, easy-to-implement language. The extension is realised by standard and custom reusable moduls.

Appendix A

Example

```
-- File model.vhd. Encapsulates Ball entity.
-- s - position, v - speed, a - acceleration

ENTITY Model IS
  PORT ( QUANTITY start, ground : IN REAL_VECTOR(2);
         QUANTITY s, v, a       :OUT REAL_VECTOR(2) );
END ENTITY Model;

ARCHITECTURE First OF Model IS
BEGIN
  MyBall: ENTITY Ball(Simple)
          PORT MAP ( start, ground, s, v, a );
END ARCHITECTURE First;

-- File ball.vhd. Instantiated in top entity Model.
-- Use a very small/variable step and an absolute tolerance 1e-3.

ENTITY Ball IS
  GENERIC ( gravity : REAL := 9.81 );
  PORT ( QUANTITY start, ground :  IN REAL_VECTOR;
         QUANTITY s,v,a         : OUT REAL_VECTOR );
END ENTITY Ball;
```

```
ARCHITECTURE Simple OF Ball IS
BEGIN
  BREAK s => start; --Initialization BREAK

  Cycle: FOR i IN 0 TO s'HIGH GENERATE
    IF s(i) > ground(i) -- 'ABOVE not implemented for vector items
    USE
      a(i) == -gravity;
    ELSE
      a(i) == -gravity - 200.0*v(i) - 10000000.0*( s(i)-ground(i) );
    END USE;
    BREAK ON s(i)'ABOVE( ground(i) ); --Descriptive only
  END GENERATE;

  v'DOT == a;
  s'DOT == v;
END ARCHITECTURE Simple;
```

Translate the bottom entity first, then the upper entity. Integrate and compile the whole model. See the distribution documentation for more. Run:

- vhdlams ball.vhd
- vhdlams model.vhd
- modelmex model first

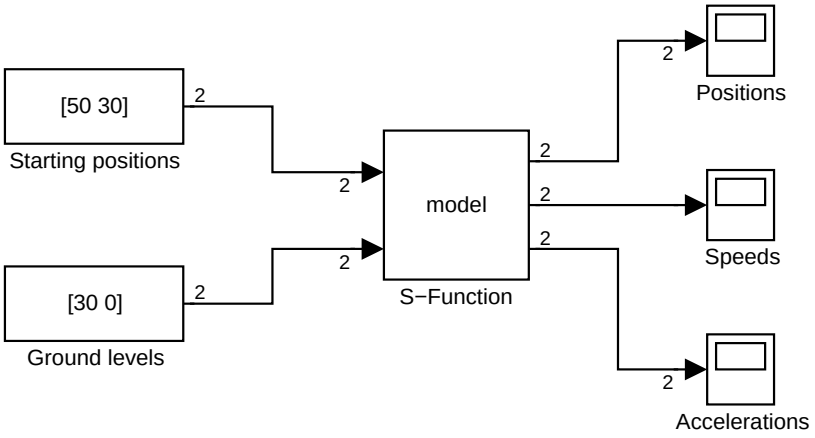

Figure A.1: Simulink model

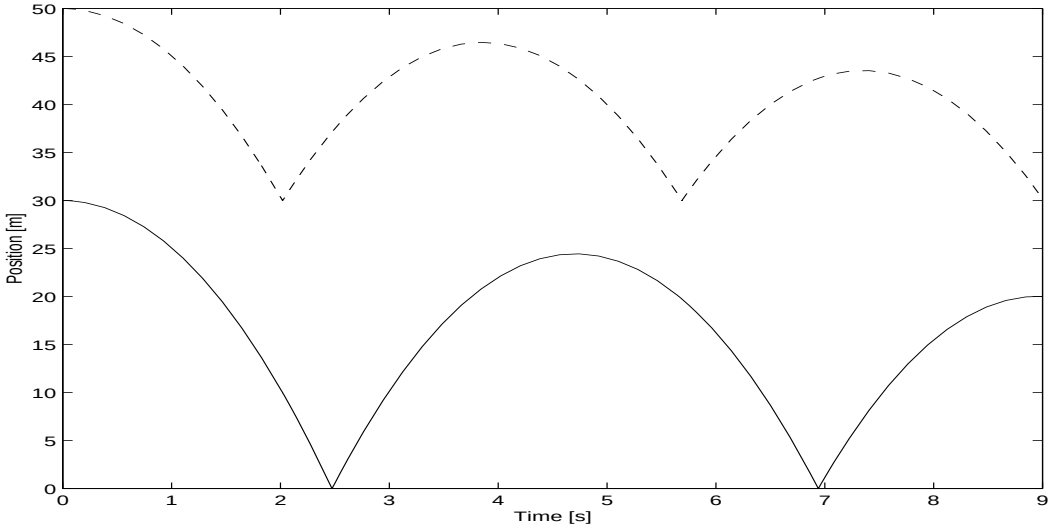

Figure A.2: Graph of ball positions

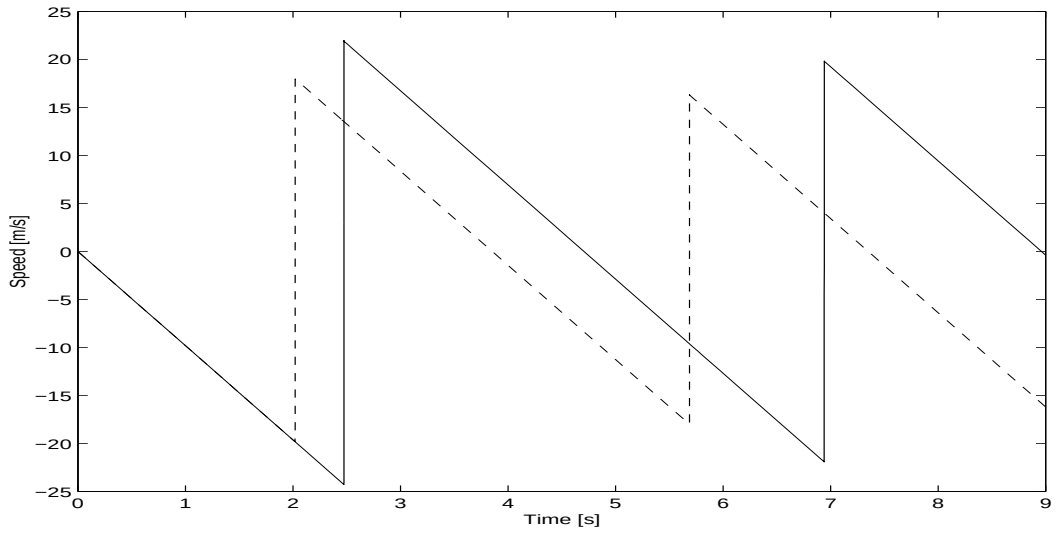

Figure A.3: Graph of ball speeds

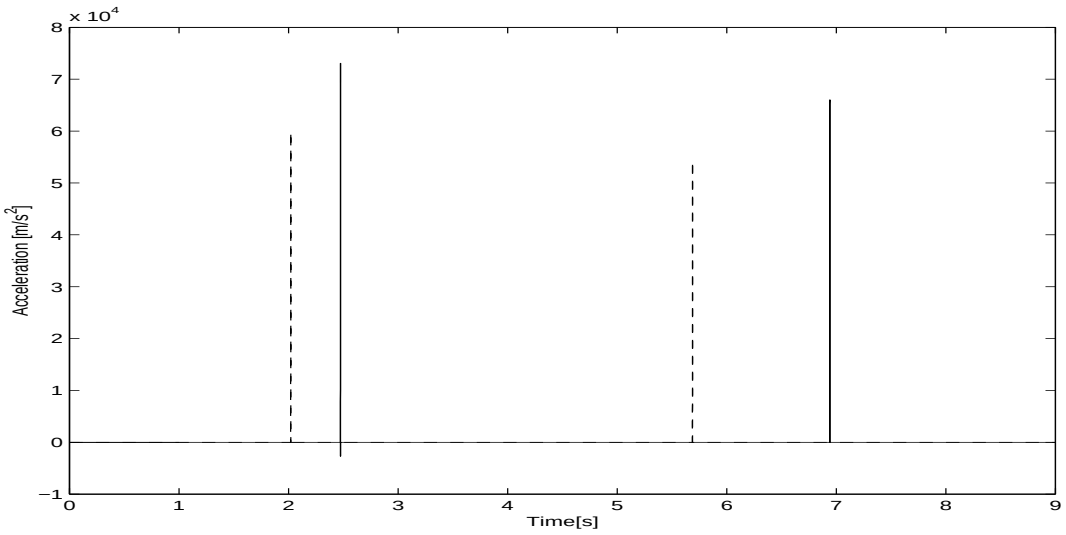

Figure A.4: Graph of ball accelerations

Appendix B

Developed framework

Container class Trans It has its own documentation in the commented source.

Persistent description classes Encapsulate and store parsed data: Architecture, Compo, Component, Entity, Extension, Port, SerialObject

Class ErrorHandler defined by [1], it performs a parser syntax error recovery

Class Token generated by JavaCC, then changed so that lowercase token representation is used because VHDL-AMS is case unsensitive

MatlabInterface uniform way to handle scalar and vector ports, integrals and derivations

Class VhdlParser defined in Vhdl.jj. It has report/debug functions and often used error message Trans objects. Several lists are used to carry an additional parsed information.

Class Vhdl processes VHDL source, calls parser. ENT, ARC description files and H file are generated. It prints a report about the translated parts, the syntax and some semantical errors.

Class MakeModel integrates the whole model

Structure Real defines operators for REAL type

Structure Vector operators, representation and behaviour of REAL_VECTOR and generally any VECTOR type.

model.cpp C/C++ interface between Simulink API and the main entity integration layer

Tested with Borland C++ 5.02, Matlab 5.3.0.10183 (R11) and Simulink 3.0 (R11) 01-Sep-1998. Developed with Sun JDK 1.3.0, JavaCC 2.1.

Appendix C

Implemented language features

- Entity, Architecture with generics and quantity ports
- Simple_simultaneous _statement [assumed as an assignment rather than equation]
- Simultaneous_if_statement
- Generate_statement—FOR version only, with nonegative integer generate parameter, without block declarations
- Initial and conditional BREAK
- Component instantiation
- REAL
- REAL_VECTOR(Size) indexed from 0. Unconstrained REAL_VECTORs, set before use, are allowed for the inner entity ports and the internal quantities.
- The first order attribute 'DOT for REAL and REAL_VECTOR
- Attributes 'LENGTH, 'HIGH, 'RIGHT, 'LOW, 'LEFT for REAL_VECTOR
- Attribute 'ABOVE for REAL
- Algebraic, relational and logical operators

Bibliography

[1] Christoph Grimm. *VHDL-AMSParser* [online]. 1997 [cit 2002-03-01]. <http://www.ti.informatik.uni-frankfurt.de/grimm/vhdlsrc.zip>. Free for evaluation purposes.

[2] Gabriel Vince, Richard Dudáš. *VHDL* [computer program] 2000 [cit 2000-09-01].

[3] IEEE, New York. *IEEE Standard VHDL Language Reference Manual (integrated with VHDL-AMS changes), Draft*, 1999

[4] Moser Eduard. *VHDL-AMS Subset for REAL-TIME.* Robert Bosch GmbH, 1999.

[5] Commerell Walter. *VHDL-AMS Realtime Simulator.* FH-Ulm University of Applied Sciencies, 2000.

[6] Analogy-Avanti, Inc. *Guide to Mixed-Signal Simulation*, 1999.

[7] National Instruments, Austin USA. *Lab-PC+ User Manual*, august 1993 edition, 1993.

[8] National Instruments, Austin USA. *NI-DAQ Users Manual for PC Compatibles*, september 1994 edition, 1994.

[9] *JavaCC* [computer program]. Ver 2.1. [USA], 1996 2002. [cit 2002-03-01]. <http://www.webgain.com/products/java_cc/>. Free registration required.