

Contents

1	Terms and Conventions	4
2	Introduction	7
3	VHDL-AMS RT	8
4	A brief Review of used Technologies	9
4.1	Matlab	9
4.2	Simulink	9
4.3	C and C++	10
4.4	Java	11
4.5	Parsers	11
4.6	JavaCC and Jjtree	12
5	State of the Art in AMS and RT Simulation	14
5.1	Analogy-Avanti, Inc.	14
5.2	Mentor Graphics, Corp	15
5.3	hAMSter	15
6	Research Problem Statement	16
6.1	Aim	16
6.2	Need	17
6.3	Value	17
7	Sources	19
7.1	VHDL-AMS Grammar from Grimm	19

7.2	VHDL-AMS translator from Vince, Dudas	19
8	Targets	21
8.1	Simulink and RT	21
8.2	C++ MEX	21
9	Strategy analyzes	23
9.1	Translation methodology	23
9.1.1	Traversing AST	23
9.1.2	Substitution production rules	24
9.1.3	Main grammar changes	25
9.2	Development process	26
9.3	Integration of instantiated entities and whole model with Simulink	28
9.3.1	Open box	28
9.3.2	Black box	29
9.4	General interface of architecture	31
9.4.1	Connection to Simulink	32
9.4.2	Hierarchical entities and generics	32
9.4.3	Ports	33
9.5	Vectors	33
9.5.1	Vector literals	34
9.5.2	Cached vectors	34
9.6	Collecting DOTs and ports	35
9.7	BREAK	37
9.8	Architecture	38
9.9	Trivial statements	39
10	Limits and possibilities	40
10.1	Hard tasks	40
10.2	Easy extensions	41
10.2.1	Processes, signals, delay mechanism	41
10.3	Relaxations to VHDL-AMS RT	42
11	Conclusions	43

A Example	45
B Developed framework	49
C Implemented language features	51

Chapter 1

Terms and Conventions

Most important general terms, shortcuts and those defined for 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 context

entity general specification of VHDL-AMS unit and its interface

architecture implementation of entity, its behaviour

instantiated entity component of upper entity

quantity analog value of specified type, scalar or vector

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

OOP Object Oriented Programming, high level abstraction that encapsulates data and 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, interface between Simulink and top entity layer

tools - programs that integrate 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 file.

serialized Java synonym for persistent

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

developer -analyst, designer and programmer of compiler and 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 optional 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 a VHDL-AMS RT simulation tool integrated with known industrial modeling framework, as Simulink. There are full-featured simulation environments ADVance MS from Mentor Graphics, Corp, TheHDL and VeriasHDL from Analogy-Avanti, Inc. They have a lot of usefull features. This makes overhead very long for RT simulation. Free simulator hAMSter has medial support for VHDL-AMS. However, its execution limitations don't fit for RT also.

Instead of large compiler and own simulation environment, translator of most used language subset to C++ is developed. It uses interface to Simulink. High speed of C++ and Simulink, stability, reliability and well structured API between them suit for fast and robust RT simulation. Existing Simulink modules and libraries are used and actuated from others. Existing support of A/D cards allows 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 [2]. This extension includes full VHDL. It enhances formal description of electrical, mechanical, hydraulical and other systems and processes.

VHDL-AMS is modular, structured and extensible. Separated formal interface and implementation supports team work. It's easy to use by non-programmers and no-IT specific engineers. Development process is fast, model is robust and safe. It defines what system does, instead of how it works (what you write is what you mean). Models are platform independent. Designers can choose from more simulation tools. They profit from 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. Behaviour of modeled systems is very detailed and thus suitable for REAL-TIME simulation. See [4].

VHDL-AMS-RT is VHDL-AMS Subset for REAL-TIME [3, Chapter 0]. It eliminates parts that don't fit requirements of Real Time simulation.

Chapter 4

A brief Review of used Technologies

4.1 Matlab

Matlab stands for **Matrix Laboratory**, a framework from Mathworks, Inc. It's widely used by engineers, scientists and mathematicians. It has own programming language with some structural possibilities, very efficient notation and implementation of vector and matrix operations. Reusable blocks of code can be defined in M file, that 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. User interface is easy to use and intuitive. Large set of toolboxes - standard. customized and 3rd party modules - is used to build models.

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 user library of specified functions that

are called from Simulink on each simulation step and interact together. It can be binary library compiled from C/C++, Ada and Fortran sources, or precompiled M file.

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

4.3 C and C++

C is general programming language providing both high level abstraction structures and low level efficient operators. It's ANSI and industrial standard used for interface specification of extensible software moduls - open systems. Source is easily portable to different platforms with minimal or none changes. 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 code. Object is instance - variable of class or struct. Member functions are used to access object's variables that can be protected from direct use by object's environment. Child class can inherit from parent class obtaining access to specified subset of its variables and functions. Child can implement or specialise virtual functions defined by parent.

C++ allows to write prototyped, easily readable, abstract and efficient program. Features are: classes, overloaded - redefined functions and operators according to argument types, templates - generic constructions (classes, functions and operators), exceptions. Types are more checked then in C, so code is safer. Syntax is simple, clean, short and easy to read, inherited by other successfull languages.

4.4 Java

Java is modern OOP programming language defined by Sun. It's used for standalone, Web/Internet and embedded device applications, being industrial standard for latter two areas. It is usefull for tool development, as it's 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. Generated platform independent byte code is interpreted and often 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 need for source code distribution, what suits for 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 development process, future integration possibilities, safety and robustness of designed applications.

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

4.5 Parsers

Grammar formally describes syntax of language. 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 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. Generally, left recursion must be transformed.

Parser reads input, analyzes it and generates AST (Abstract Syntax Tree) for it. Different parsing technologies exist. Several Compiler Compilers as

lex, yacc, bison, PCCTS/ANTLR, JavaCC are used. They generate parsers - stack machines that accept sentences of given language, translators - substitution engines, and compilers to binary code. See [PrecentHall].

4.6 JavaCC and Jjtree

JavaCC is state-of-art parser/translator tool. The source JJ file is a grammar specification consisting of token, 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, that 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 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 in separate for different lexical states that function as simple low-level parser state machine. Special tokens are those unimportant for parsing, usually comments.

productions BNF has symbolical and more structured rules then token definitions. BNF has structure *expansion-return-type name-of-production (optional-parameter-list) : {java-block} {expansion-rules}*. Java block is usually used for definition of local variables. 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 parsing process

JavaCC generates (but doesn't override) several Java classes. Object of parser class represents whole translator. It is attached to input stream, possibly from a file, that is parsed by chosen root production method. Its output (transformed input) and byside effects (generated files...) are the result of translation. It can be embedded in other systems and gains all 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 [8].

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 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 separated from JJT source and need to be synchronized with changes to JJT grammar structure. Because nodes are indexed and optional/repeated appearance must be checked 'manually', it doesn't 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 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 languages MAST, SPICE, VHDL-AMS and Verilog-AMS. Their products are used in Automotive, Communications, Military, Aircraft and Power electronics industries.

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

5.2 Mentor Graphics, Corp

Mentor is focused on 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 efficient ModelSim solver is used for digital part, and analog portions are simulated by one of several algorithms.

5.3 hAMSTER

Simulator HAMSTER is for MS Windows only. Own editor, simulation environment, limited ODE and unlinear solver. Only scalar quantities are allowed. Limited processes are supported. Integer and enumeration types, several predefined attributes, libraries and several packages can be used. It is focused on 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 implementation by this project
- Analyze strategies
 - integrating VHDL-AMS model with Simulink, interface between Simulink and main entity
 - connecting inner VHDL entities to each other and handling their analog signal flow to Simulink, that passes by 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 future
- Create the framework

- general interface between 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 top entity

6.2 Need

Software models, formal specification and prototypes increase productivity of development, design and manufacturing processes. Industry follows standards as VHDL-AMS 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 must today. Hardware-In-The-Loop (HIL) simulation shows hidden weaks, increases development speed, quality, robustness and safety of product. It's not sufficient to test analog and digital parts in separate, because they influence each other in unpredictable ways. This is critical in 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

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

DSP. Simulation is very fast and usable in RT. Designer uses all power from Matlab and Simulink.

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

Chapter 7

Sources

7.1 VHDL-AMS Grammar from Grimm

There are pure VHDL grammars for Yacc, that don't 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 and easy to use and compare with LRM for implementation notes. 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 parsing. Thus respective VHDL-AMS code parts are parsed by different, syntactically equivalent rules.

7.2 VHDL-AMS translator from Vince, Dudas

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

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

are checked recursively, required entities are processed first.

Translator is based on Grimm's JJT grammar. Transformation is performed by methods added to Jjtree generated AST classes. Statements are collected, 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 to instantiated entities are defined. MdlInitializeSizes sets number of ports, states and simulation options. Execution part (MdlOutputs) receives integrated values from Simulink, does equations and returns output ports to Simulink. MdlDerivatives evaluates derivations directly in assignment to vector that is sent to Simulink. Actual generics are set in latter two functions. Three other trivial or empty functions are generated. Equations - simple_simultaneous_statements with basic mathematical operators - are the only supported execution statements.

Chapter 8

Targets

8.1 Simulink and RT

The required calculation time for each time step must stay below some predictable maximum time [3, Chapter 0].

Simulink calls several user-implemented C functions, on initialization, each step and end of simulation. It offers several fixed-step ODE solution methods. Simulation is fast and fits for RT.

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

8.2 C++ MEX

Model code translated to C++ is fast, robust, has high structurality and portability. It can be used in other frameworks than Simulink with minimal changes. Such a model uses 3rd party and customer's modules accessing them through their interfaces defined in C/C++. Code is preprocessed to C and compiled for DSPs, because generally there are no C++ compilers for them.

Support for multifunction AD/DA and IO card Lab-PC+ from National Instruments [6] can be added. C++ drivers from National Instruments Data acquisition utilities NI-DAQ [7] for Borland C++ and Microsoft/Visual C++.

Chapter 9

Strategy analyzes

After general decisions before beginning of implementation, specific issues were analyzed during development when involved. Those two processes were tied and hard to separate. Analyzes depend on grammar and behaviour specification. Structure of grammar that makes skeleton and is a part of translator was changed and restricted during development. Also other behaviour constraints and relaxations were defined according to existing and future implementation needs. Thus descriptions of respective production rules are here instead of presenting them in separate.

9.1 Translation methodology

9.1.1 Traversing AST

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

Main problem is separation of JJT grammar and AST code. General access to any parsed node is by API functions and node indices rather by their names. They must be retyped and give possibility for runtime errors that are hard to find. Checking of optional nad repeated nodes must be done by Java conditional and loop commands. This is a lot of long trivial code, that only copies existing grammar structure and depends on it. Developer is counter and source code comparator. If grammar is reorganized, respective

AST code must be searched for and reflective changes made. This is much more complicated in presence of a lot (186) AST classes defined in their own files. Maintenance of such project becomes a nightmare.

Such approach makes illusion that it supports team work. Source is distributed to number of different files that are not updated automatically on 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 future, little 'active' part and big skeleton doesn't seem the best solution for large projects. A lot of trivial - waste code must be written.

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

Author added translation of `simultaneous_if_statement` to this framework using Trans container. Combination of AST and collecting - separating parsed statements solves only few language features. It is not general, can't grow lineary and is very limited to extend. None of this code is used in presented translator to C++. Approach was analyzed and results influenced decisions and solutions of this project. Idea of description files is enhanced by use of Java serialization.

9.1.2 Substitution production rules

Decision not to use Jjtree was chosen. JavaCC production rules do translation, substitution and integration tasks. This is what JavaCC productions suit for - to transform and pass up (return) 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 point where it is parsed. This code can be in nested grouped, optional and repeated statements and is executed each time rule accepts input. Results of 'called' productions are assigned to local Java variables in type-safe way. Symbolical use of parsed nodes and automated execution of specified code enhance

simplicity and quality assurance. Translator source is very expressive and resembles 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`. Content is printed with default or user-specified indentation. This by-side product is fully independent from this project and `JavaCC`, it can be used for format transformers, macro and template engines.

Important improvement is easier and safer translator development process. Translator code is compact, main part in one `JJ` file. Even most of code is centralized, it fits for team work [Trans doc]. No dependency checks between grammar rules and generation methods are needed when some of them was changed, as they are all at one place. Code is type safe and very easy to modify. Development control was more efficient and less painful. 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 additional information used for checks performed.

9.1.3 Main grammar changes

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. Original 334 productions are transformed to 234. Grammar is more compact for our purpose. Translator source code reduced from `JJT` source, 186 decentralized `AST` and 4 `JJT` additional classes (generated in unique files) to `JJ` file and 11 framework classes. Other 2 of original 4 classes defined by Grimm are used. This dramatically increases development speed. Startup and execution of translation process are faster.

Because `VHDL-AMS` is case insensitive, and `C/C++` case sensitive, all identifiers are transformed to lowercase. `C++` layer and integration library identifiers consist of uppercase, that prevents the conflict with user specified `VHDL-AMS` code. `C++` definitions used directly by `VHDL-AMS` code are

in lowercase. They're in different case for highlighting in this document, 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 a time, so long JavaCC processing must be repeated. See [Trans]. BNF HTML documentation allows fast orientation in grammar.

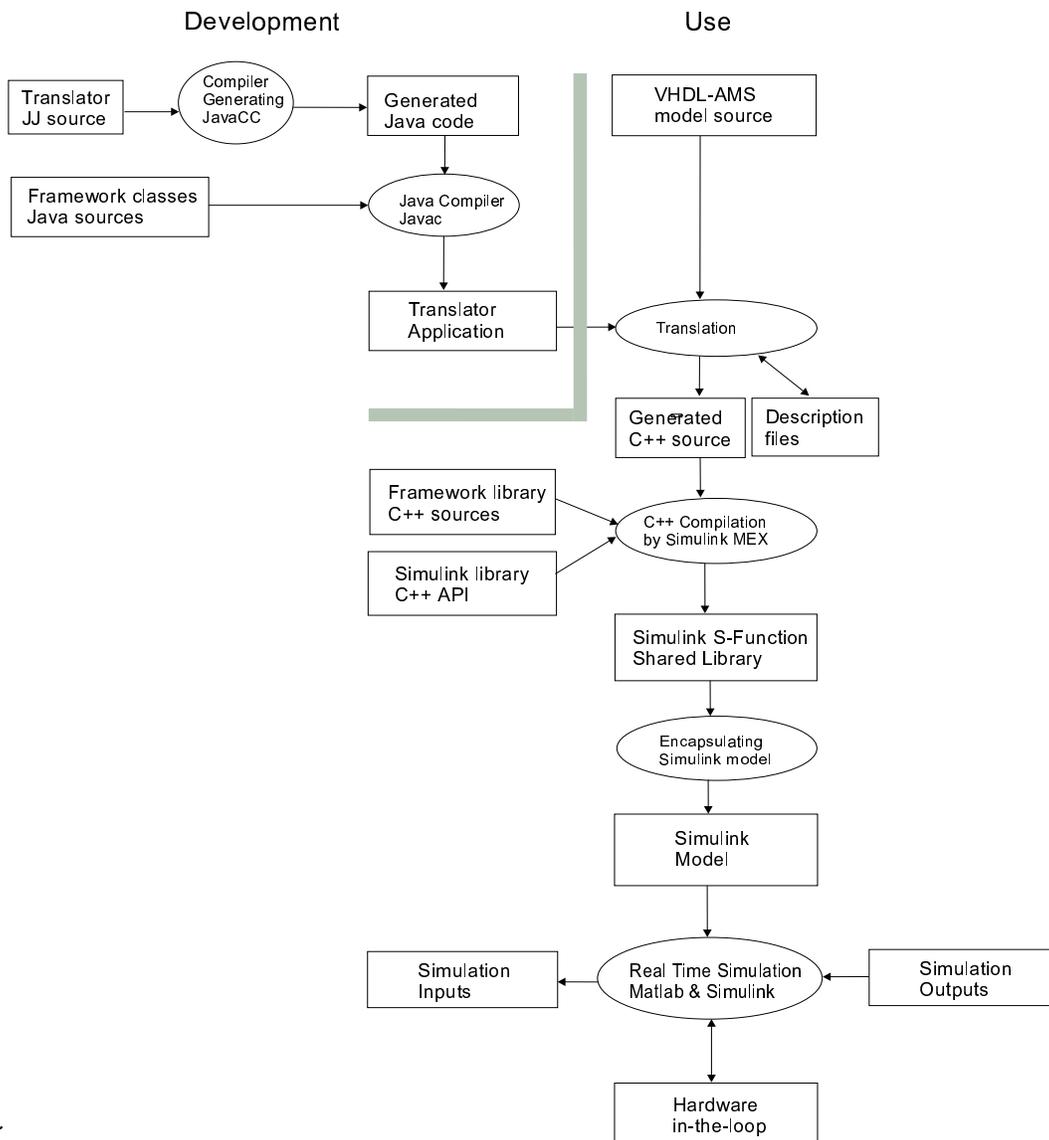
If translation code breaks parsing process, unusual runtime parser errors raise for correct VHD input. Those are all eliminated by use of simple rules. Generally, created container object must be returned from main block of production, or from its end [Trans].

Compilation of Java classes finds other errors uncaught by JavaCC. Report and generated compiler class resembles JJ code, so more errors can be fixed at the same time.

Most of BNF productions do trivial transformation from VHDL to C++. Complicated rules and special features have by-side effects, 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 translation and integration of upper entity. Semantical checking that can't be left for C++ is done.

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

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



Development process and use

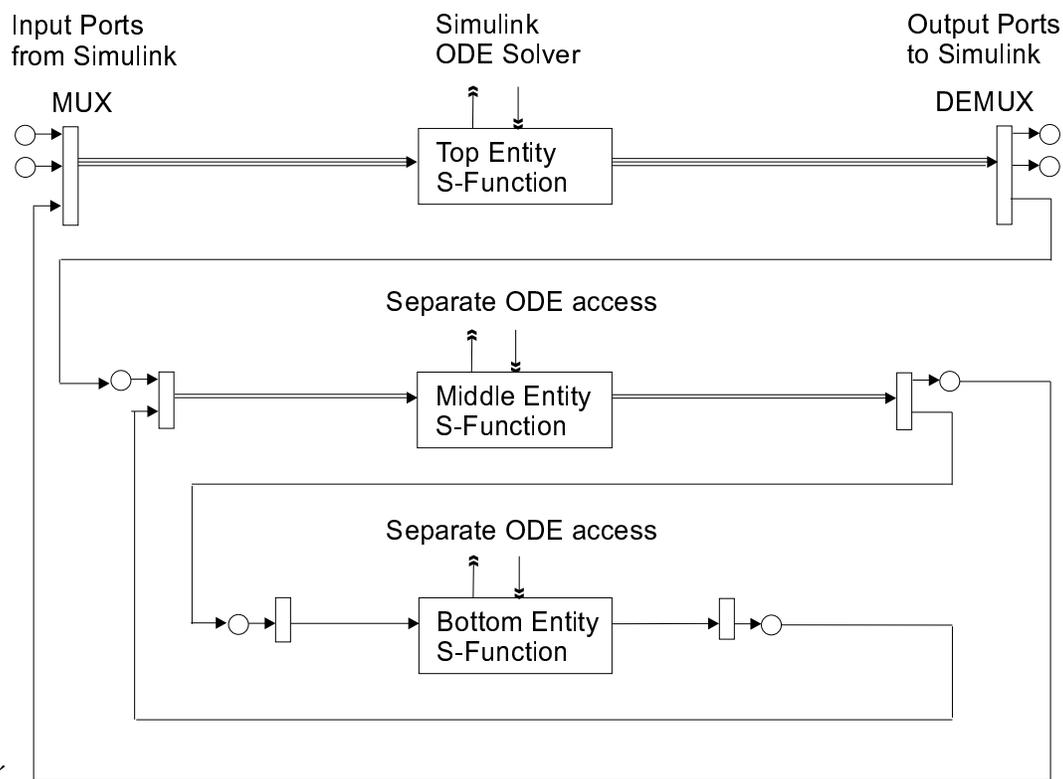
9.3 Integration of instantiated entities and whole model with Simulink

9.3.1 Open box

Framework of Vince-Dudas generates well structured MDL model that copies tree of instantiated entities with their mapped ports. This gives designer very usefull possibilities [chapt. 7.2]. Unique M S-Function generated for each instantiated entity has direct access to ODE solver with own list of derivations and integrals. That fits for VHDL-AMS feature of DOTed quantities specified 'directly on place', in architecture behaviour definition.

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

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



Several S-Functions in one hierarchical Simulink model

9.3.2 Black box

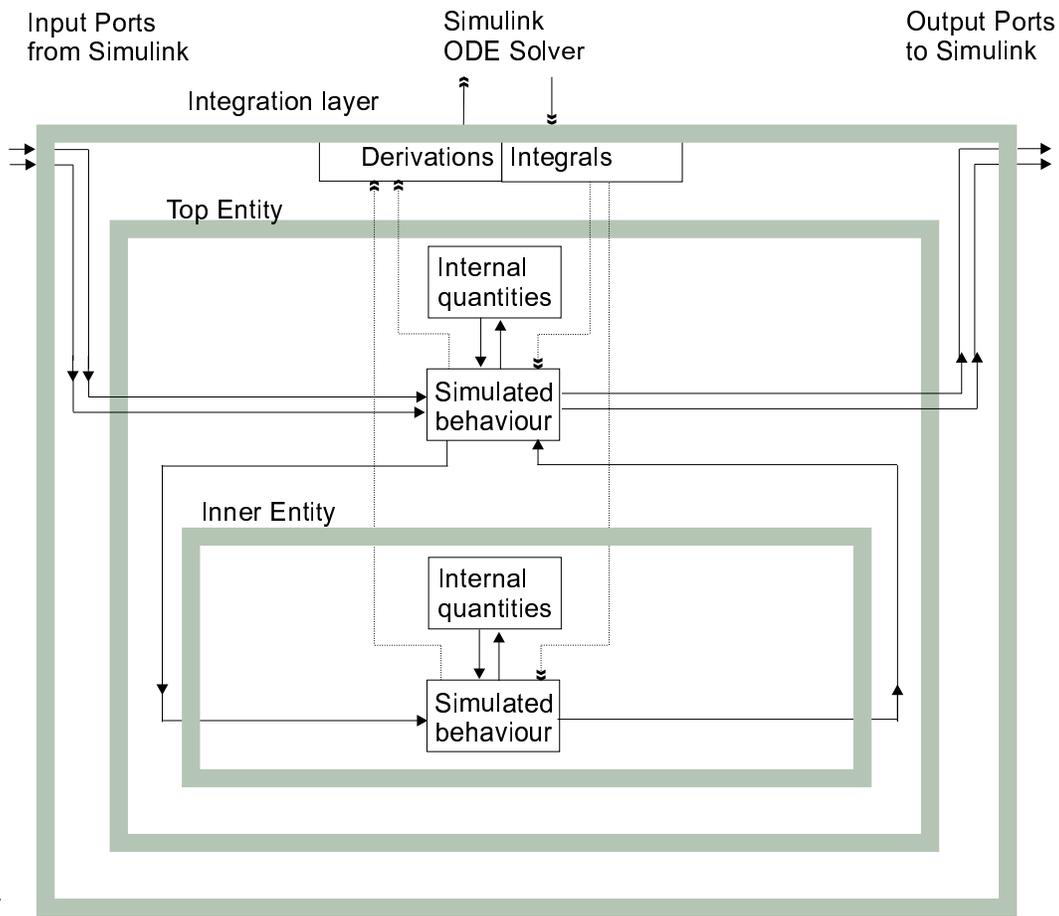
Translator generates one C++ S-function for whole model - main entity and all its subentities. C++ code created for each entity uses simple interface, is automatically indented and well structured. S-function handling code called by Simulink is separated in another, stable layer in file model.cpp. Thus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

Such a separation was impossible by M code, because it doesn't have comparable structural possibilities. Matlab can't pass parameters to M functions

by reference, global variables must be redeclared in functions that use them. OOP encapsulation of code by objects and operator overloading is not possible. This makes its extensible use in large project very hard.

Designer can debug an inner entity separately by making different model for it. 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 production use.

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



One S-Function for whole model

9.4 General interface of architecture

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

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

they are passed to and from Matlab on each simulation step.

Uniform MatlabInterface allows connection to other entities and librares, that communicate to Simulink. It defines 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

Connection to Simulink is used for ports and analog quantities. Nonintegrated analog and digital or discrete signals are implemented on C++ side. Number of analog quantities (states) and their default values are sent to Simulink during 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 Simulink. That's because of static variables and fixed generated S-functions name 'model'. Otherwise unique file model.cpp had to be generated. Designer can encapsulate several instantiated entities in simple higher entity that becomes new model.

S-Function is implemented as dynamic library. If already used by running Matlab and Simulink, those must exit before model is recompiled. Fixed-step ODE solver is required. Multi-step methods can make problems, see [Break].

9.4.2 Hierarchical entities and generics

Both actual mapped generics and ports of instantiated entity - only names or order and 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 entity is instantiated or integrated. Default generics are used if omitted. Named ports and generics at element_association() are processed correctly. Actual values of generics are set in call to its constructor when entity object is created in upper entity or integration layer.

To keep whole approach simple, there is no possibility to set generics from Simulink. Main entity can't have any generics, even with default values. Designer can instantiate it in trivial higher entity that specifies and passes generic values and connects the ports.

9.4.3 Ports

Input vector ports of main entity need width specification. Simulink allows dynamically sized ports, but restricts process of parameter exchange at initialization of S-function. It must know number of analog states, before it passes actual width of input ports. Because analog quantity vectors can be assigned from input ports, translated model needs their size first. So main entity must have all port widths specified. However, instantiated entities can have unconstrained input and output ports, which size is set in runtime from 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*). 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 scalar `REAL`. Underlying array is always allocated dynamically in vector constructor (if width is known) or on first assignment (if unconstrained).

`VeCtor` is subclass of `MatlabInterface` and implements its functions. Items of vector are sent to/from Simulink ports. `DOTed` vectors have their derivations sent to, and integrals received/sent from Simulink as whole. Derivations and values of `REAL_VECTOR` are implemented as separated vectors for efficiency. This restricts the use of integrated vectors. They can be accessed only in 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 flag `inSubtypeIndication`. It calls `name()` and `name_extension()` that handle parsed data in special way. Vector size is assigned to variable `typeParameters` and then carried in object of class `Port`, rather than passed to C++ as part of vector type. It becomes a default or first parameter to vector constructor. Vectors of unspecified widths have memory allocated on assignment. Sizes are always checked in runtime and shorter vectors can be assigned to longer ones.

Element access parenthesis operator (`index`) uses `name()` and `name_extension()`. 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 aggregate statement: `(expr0, expr1, ..., exprN)`. This is translated to C++ code `(VeCt(N,expr0),expr1,...,exprN)`. 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 don't check types of expressions and context of their use, possibly nested.

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

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

9.5.2 Cached vectors

Nested operations produce intermediary results, new vectors. They are not needed anymore after evaluation of whole expression. Repeated dynamic allocation and disposition is time-consuming and can't be used for RT.

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

Chosen way is to dynamically allocate intermediary arrays on 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 model destruction. This keeps whole approach simple and gains from C++ high level abstractions. Cached vectors are defined by template `tempVector<class T>`.

First pass that involves allocation is much longer. However, it is called at Simulink initialization before 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 simulation step. Possible, unimplemented solution is to duplicate list of cached vectors after initialization pass. This should make reserve high enough for demand peaks.

9.6 Collecting DOTs and ports

Derivation of quantity - in [3] for integration only - 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 function defined for types `REAL` and `REAL_VECTOR`.

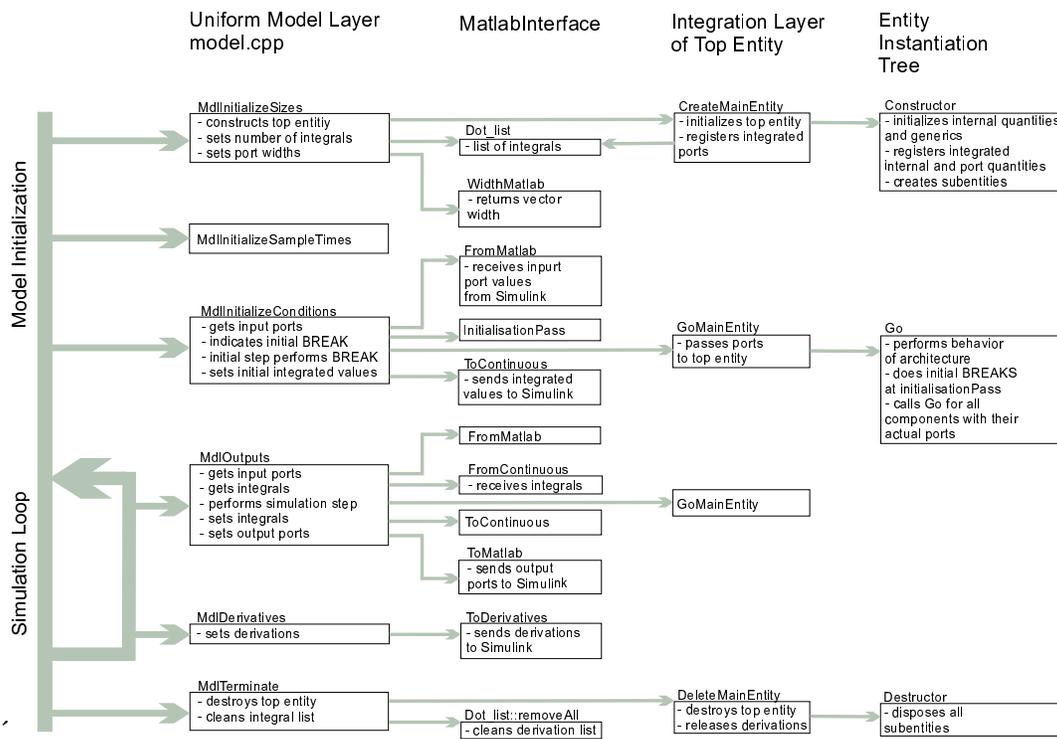
Entity defines analog and discrete quantities together with no distinction. Their behaviour is known from body of architecture. Scalar and vector internal quantities and output ports can be changed by DOT. Simulink gives limited control over analog values and their derivations. S-function is required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

quantities from all instantiated entities.

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

Only first order DOTs are allowed, according to [3]. DOTs of vector elements and elements of vector DOTs can't be used, see [9.5]. DOTs of record type elements, vector elements and translation of higher DOTs to form DOT(Quantity, Order) worked in experimental version. They are not implemented because of VHDL-AMS grammar ambiguity and complexity. Rather than those, vector literals and vector item access operator are implemented.

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



Simulation process

9.7 BREAK

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

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

Conditional breaks that have break list of `quantity=>value` pairs are translated to C++ `if()` statement. Assignments are performed when condition occurs.

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 entity. At `architecture_body()`, architecture name is checked (if it appears after END token). A header file `EntityArchitecture.H` is generated that includes H files of all components. `Library_unit()` clears lists of DOTs, generics, ports, quantities and components. Those are filled during parsing entity/architecture, checked and reported to user.

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

Structure has quantities, generics, instantiated entities and flag `InitializationPass` as its member variables. They are initialized by constructor's parameters or their default values outside of 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.

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

Constructor body sets `InitializationPass` to true and registers analog quantities and components' ports (that are not mapped to port of this entity) by function `integrated()`. Those were collected from `architecture_statement_part` and read from ARC descriptions of instantiated entities. 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 reference. IN ports are passed by 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). 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 architecture.

9.9 Trivial statements

- Algebraic, relational and logical operators are transformed to their C++ equivalents.
- Expression() checks whether NAND and NOR are in sequence, what 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 block declarative part can't be used. Iteration identifier is of type INTEGER and with nonnegative values only, that fits for 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 analyzes at beginning. Other constraints raised at development process and next ‘local‘ analyzes. Also some relaxations to VHDL-AMS-RT are available. For details, see [3], implementation description, translation output and Vhdl.jj source. Supported language features are in Appendix B.

10.1 Hard tasks

All those tasks are hard to implement by chosen approach. Use of symbol tables could make some of them possible.

Algebraic loop detection It requires symbol tables.

Matrices Even 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 analyzes.

10.2 Easy extensions

Libraries and Packages can be implemented by included C++ headers.

Standard STD package and its NOW function can return Simulink 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

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

Transformation to Simulink model and using its possibilities for discrete signals can be used. This involves 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 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 special process with restricted paralelism.

Wait statement introduces a list of remaining times or deadlines. More efficient C++ static local variables defined in `go(..)` function can't be used for it, because one entity can be instantiated several times. New structure for

each instantiated entity as in [Vince,Dudas] 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 substatements in each of its branches. See [3, 0.5.2]

Chapter 11

Conclusions

Presented solution is easily extensible within boundaries given by chosen approach. Most of usefull VHDL-AMS behavioral statements can be implemented by simple transformation to C++ code. Structural parts as [10.2] are easy to add. A lot of 'exotic' features can't be used, because of wide and ambiguous syntax.

Support for other simulation environment 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 initalisation process and allow unconstrained ports of main entity.

Language gives designer a large set of possibilities. Approach 'what you write is what you mean' allows short code that uses several structural and behavioral features. Feature of DOTed quantities specified 'directly on place of use' hides the necessary processing. This is hard to implement in efficient way. Context-dependency and ambiguous syntax are much bigger problems then paralel behaviour. User can't remember all of those constructions and probably uses just the preferred subset. However, correct implementation must accept and process all possibilities. The question is, whether it is worth. When reading the source, one must look at different parts to check, if symbol is a function, variable. . . Even simplier languages, based on Pascal and C, have more possibilities to distinguish different use. Unparameterized functions must be called with () pair, array access is by []. Thus source code

is easier to read, complain about errors and translate.

Incremental increasing of language features makes 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. Extension is 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(3);
          QUANTITY s, v, a       :OUT REAL_VECTOR(3) );
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.
-- Simplified source without discontinuity BREAKs.
-- Use variable step and 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;
```

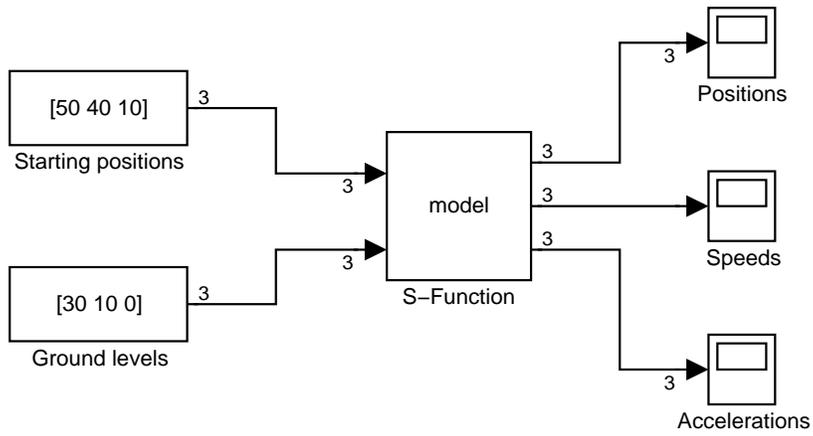
```
ARCHITECTURE Simple OF Ball IS
BEGIN
  BREAK s => start; --Initialization BREAK

  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;

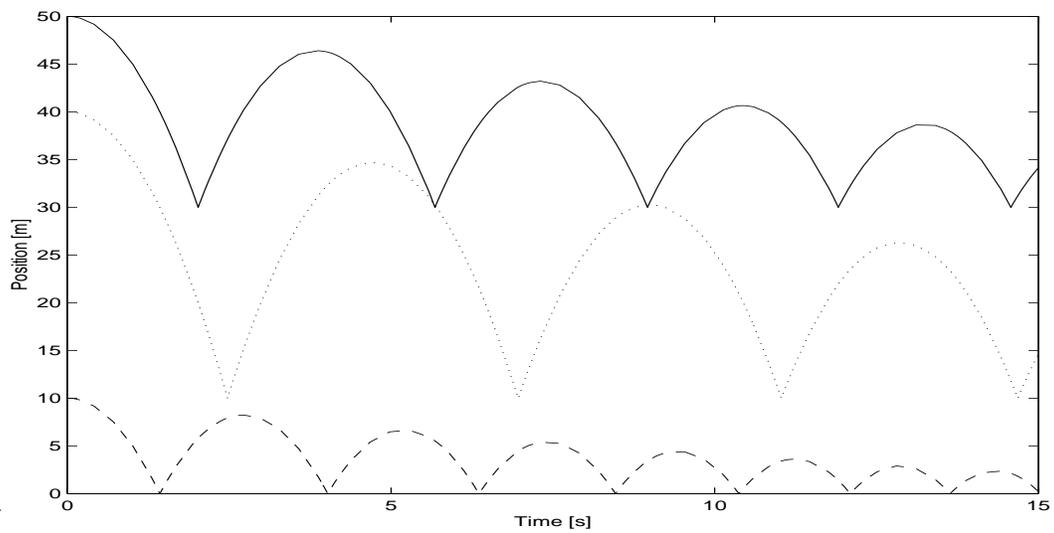
  v'DOT == a;
  s'DOT == v;
END ARCHITECTURE Simple;
```

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

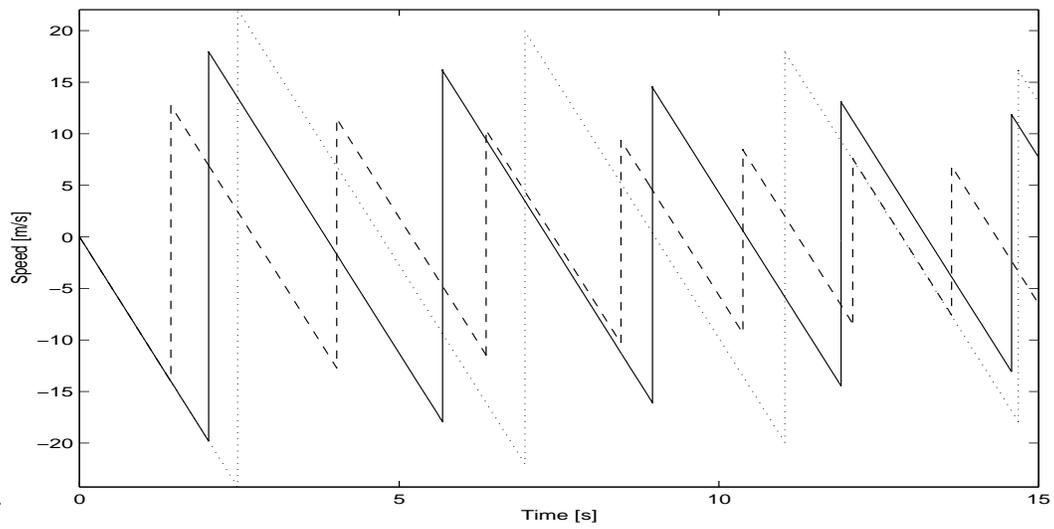
- vhdlams ball.vhd
- vhdlams model.vhd
- modelmex model first



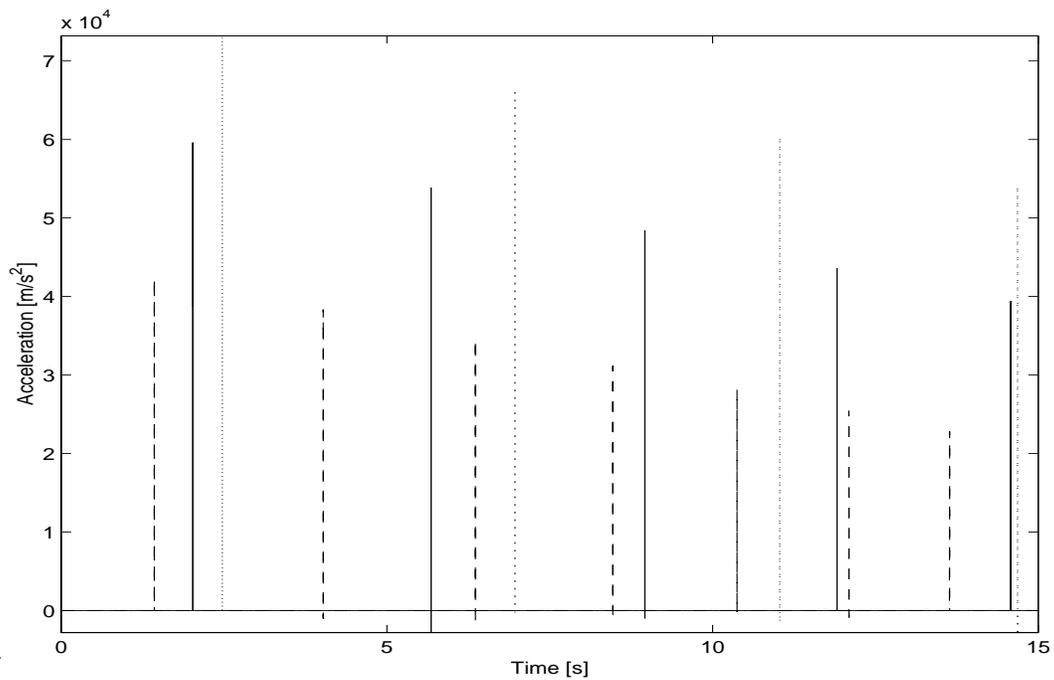
S-Function used in Simulink model



Graph of ball positions



Graph of ball speeds



Graph of ball accelerations

Appendix B

Developed framework

Container class Trans see [Trans.doc]

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

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

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

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 additional parsed information.

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

Class MakeModel integrates 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 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 allowed for inner entity ports and internal quantities set before use.
- First order attribute 'DOT for REAL and REAL_VECTOR
- Attribute 'ABOVE for REAL
- Algebraic, relational and logical operators

Bibliography

- [1] <http://www.ti.informatik.uni-frankfurt.de/grimm/hybrid.html>
- [2] IEEE, New York. IEEE Standard VHDL Language Reference Manual (integrated with VHDL-AMS changes), Draft, 1999
- [3] Moser Eduard, VHDL-AMS-RT
- [4] Commerell Walter. VHDL-AMS a normed Modeling Language for multi-domain Systems. Simulation News Europe, 1999.
- [5] Commerell Walter. VHDL-AMS Realtime Simulator. 2000
- [6] National Instruments, Austin USA. Lab-PC+ User Manual, august 1993 edition, 1993.
- [7] National Instruments, Austin USA. NI-DAQ Users Manual for PC Compatibles, september 1994 edition, 1994
- [8] Description of the JavaCC Grammar File, <http://>