

MSc THESIS

Translation of SystemC to Synthesizable VHDL

E.P.M. van Diggele

Abstract



CE-MS-2006-06

Efficient hardware/software co-design environments cater to the need to test entire systems early in the design cycle. These design environments allow hardware to be modeled at a higher level of abstraction than is possible with more traditional hardware design languages. SystemC is such a hardware/software co-design environment that allows hardware models to be written in C++. Consequently, the SystemC models need to be translated to a description (usually in traditional HDLs) to be synthesized to an actual hardware implementation. However, these translations can be quite complex and error-prone leading to various unwanted delays in the development process. In this thesis, we present a software tool that provides an automated translation from SystemC models to synthesizable VHDL models. In detail, the tool translates SystemC and C++ (with certain restrictions) to synthesizable VHDL. Besides performing obvious translation of many SystemC elements, the tool focuses on the automated translation of control structures and expressions. Finally, five real-world SystemC models (ranging from simple to complex) were translated using the tool. Simulation of both the SystemC models and the translated VHDL models yielded the same or expected results.

Translation of SystemC to Synthesizable VHDL

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

E.P.M. van Diggele
born in Etobicoke, Canada

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Translation of SystemC to Synthesizable VHDL

by E.P.M. van Diggele

Abstract

Efficient hardware/software co-design environments cater to the need to test entire systems early in the design cycle. These design environments allow hardware to be modeled at a higher level of abstraction than is possible with more traditional hardware design languages. SystemC is such a hardware/software co-design environment that allows hardware models to be written in C++. Consequently, the SystemC models need to be translated to a description (usually in traditional HDLs) to be synthesized to an actual hardware implementation. However, these translations can be quite complex and error-prone leading to various unwanted delays in the development process. In this thesis, we present a software tool that provides an automated translation from SystemC models to synthesizable VHDL models. In detail, the tool translates SystemC and C++ (with certain restrictions) to synthesizable VHDL. Besides performing obvious translation of many SystemC elements, the tool focuses on the automated translation of control structures and expressions. Finally, five real-world SystemC models (ranging from simple to complex) were translated using the tool. Simulation of both the SystemC models and the translated VHDL models yielded the same or expected results.

Laboratory : Computer Engineering
Codenummer : CE-MS-2006-06

Committee Members :

Advisor: Stephan Wong, CE, TU Delft

Chairperson: Stamatis Vassiliadis, CE, TU Delft

Member: Nick van der Meijs, CAS, TU Delft

Contents

List of Figures	ix
List of Tables	xi
List of Listings	xiii
Acknowledgements	xv
1 Introduction	1
1.1 SystemC	2
1.2 VHDL	2
1.3 Background and Related Work	3
1.4 Problem Statement	3
1.5 Thesis Overview	4
2 Synthesis and SystemC	5
2.1 Synthesizable SystemC	5
2.1.1 Module	6
2.1.2 Channels	7
2.1.3 Data Types	8
2.2 Synthesizable C++	9
2.2.1 Classes	10
2.2.2 Variables	10
2.2.3 Types	10
2.2.4 Functions and Methods	11
2.2.5 Statements and Expressions	11
2.2.6 Exceptions	12
2.2.7 Pointers	12
2.2.8 Specifiers and Classifiers	13
2.3 Conclusion	13
3 SystemC to VHDL Translation	15
3.1 Types	15
3.2 Declarations	20
3.3 Statements	22
3.3.1 If Statements	26
3.3.2 Switch Statements	27
3.3.3 Loop Statements	27
3.3.4 Constant Propagation	28
3.4 Expressions	29

3.4.1	Operation Expressions	29
3.4.2	Accessor Expressions	31
3.4.3	Call Expressions	32
3.5	Model Structure	33
3.5.1	Bindings	34
3.5.2	Components	35
3.5.3	Processes	36
3.6	Conclusion	37
4	The Tool	39
4.1	Parsing C++	39
4.2	GCC AST	40
4.3	VHDL AST	41
4.4	GCC AST to VHDL AST	41
4.4.1	Binding Constructor	42
4.4.2	Processes and Functions	45
4.4.3	Loops	46
4.4.4	If Statements	46
4.4.5	Switch Statements	46
4.4.6	Expressions	47
4.5	Code Generation	49
4.6	Conclusion	50
5	Testing	51
5.1	Test Models	51
5.1.1	Random Number Generator	52
5.1.2	MD5	52
5.1.3	DES	52
5.1.4	USB	52
5.1.5	CORDIC	53
5.2	Test Results	53
5.3	Conclusion	55
6	Conclusions	57
6.1	Summary	57
6.2	Main Contributions	58
6.3	Future Directions	59
	Bibliography	62
A	VHDL Abstract Syntax Tree	63
A.1	Common	63
A.2	Declarations	64
A.3	Expressions	67
A.4	Types	70
A.5	Statements	71

A.6	Strings and Identifiers	72
A.7	Integer Constants	73
A.8	Helper Nodes	74
B	Tool and Test Model Source	77

List of Figures

2.1	General Compiler Structure	5
2.2	Module Inheritance	6
2.3	Range and distribution of floating vs fixed representations	9
3.1	Finite state machine of C++ Loop	23
3.2	Control Flow Graph of Embedded Loop	27
5.1	VHDL Random Number Generator Model Timing	53
5.2	VHDL MD5 Model Timing	53
5.3	VHDL DES Model Timing	54
5.4	VHDL USB Model Timing	54
5.5	SystemC CORDIC Model Timing	55
5.6	VHDL CORDIC Model Timing	55

List of Tables

2.1	SystemC Data types	9
3.1	Mapping of Types	17
3.2	Mapping of Operators	30
4.1	VHDL AST Structures	42
4.2	Mapping of SystemC Methods	50
5.1	Test Models	51

List of Listings

3.1	Test Listing	16
3.2	SystemC Vector Type	16
3.3	VHDL Vector Type	17
3.4	SystemC Struct Declaration	17
3.5	VHDL Vector Type	18
3.6	C++ Structures	18
3.7	VHDL Mapping of records	18
3.8	VHDL Vector Type	19
3.9	SystemC Array	19
3.10	VHDL Array	20
3.11	SystemC Declarations	20
3.12	VHDL Declarations	21
3.13	SystemC Process with a single loop	22
3.14	VHDL State machine for single loop	23
3.15	Loop embedded in an if-statement	24
3.16	Naive mapping for an embedded loop	25
3.17	Correct VHDL State machine for an embedded loop	26
3.18	VHDL Loops	28
3.19	C++ Increment Expressions	31
3.20	Pre-Increment Operator Mapping	31
3.21	Post-Increment Operator Mapping	31
3.22	Index Operator in SystemC	32
3.23	Index Operator in VHDL	32
3.24	Call Expression	33
3.25	VHDL Call Expression	33
3.26	Conditional Expression	33
3.27	VHDL Conditional Expression	34
3.28	SystemC Binding	35
3.29	SystemC Component Binding	35
3.30	VHDL Component Instantiation	35
3.31	SystemC Process	36
3.32	VHDL Process	36
4.1	SystemC Structure	43
4.2	VHDL Structure	43
4.3	Algorithm to map loop statements	47
4.4	Algorithm to map if-statements	48
4.5	Increments as Conditions	48
4.6	Increments as Conditions: Mapping	49
4.7	Increments as Conditions: Tool Result	49

Acknowledgements

I would like to thank my advisor Stephan Wong for his efforts in guiding me during this thesis. His work kept me on track and allowed me to complete the work successfully. Sven Sammelius proof read my thesis for which I am deeply grateful. My use of language is, at times, convoluted and with out the efforts of Sven and Stephan this work would not have been legible.

My parents deserve gratitude and respect for supporting me during my studies and life. They gracefully dealt with my, at times, relaxed view on life supporting me throughout.

Finally, Tisha Stikvoort drove me to complete my thesis, basically telling me to "get a move on". With out her I might still be busy and I am grateful that I am not.

E.P.M. van Diggele
Delft, The Netherlands
June 21, 2006

In hardware design, the utilization of high-level hardware description languages (e.g., VHDL) has increased due to the existence of many automated tools, e.g., simulation and synthesis tools. Traditionally, the design of hardware is performed separately from the software design (when designing programmable hardware or when hardware is used in conjunction with software) and they are first combined in the integration stage to fully test their interoperability. This is rather late in the overall design process and as such new methods have been introduced to allow the hardware and software to be simulated in a shared environment. An example of such a hardware/software co-design environment is based around SystemC. SystemC is a class and template library for C++ that adds the necessary constructs to simulate hardware. More specifically, utilizing standard C++ compilers the hardware functionality can be simulated in conjunction with the application software itself. However, SystemC requires a huge manual effort to be rewritten in VHDL to allow existing synthesis tools to generate efficient hardware implementations.

An automated path from SystemC to a traditional hardware description language (HDL) allows SystemC to be used as a primary hardware development language by leveraging existing synthesis tools and methods. Moreover, research into automated translation of SystemC models to traditional HDLs would benefit from the availability of such a software translation tool to extend upon. Such a tool would provide a basic framework for the translation from SystemC to a traditional synthesizable HDL allowing further research to concentrate on specific areas of high-level synthesis of SystemC and traditional programming languages.

In this thesis, we present a software tool providing an automated translation from SystemC[23] to synthesizable VHDL[15][16]. The tool provides a translation for all of the directly translatable constructs. Furthermore, it provides a translation for all control structures, these are not all directly translatable since they either do not exist in the target language or are disallowed for synthesis. The most relevant of these control constructs is the loop construct. We further propose that, in the context of a SystemC model, static analysis of the model can provide a mechanism to further reduce the amount of control structures to simplify the translation. The software tool incorporates the static analysis method to reduce the amount of loop constructs in the SystemC model. To evaluate our tool, five SystemC models were translated to VHDL models. The comparison of the functional simulation results, from both the SystemC and VHDL models, yielded the same or expected results. This experimentally confirms the correctness of our proposed software tool and methodology.

1.1 SystemC

SystemC is not a formal language like VHDL is. It is a library of classes and templates for C++ that provides the needed constructs for hardware simulation. It provides a simulation kernel with which digital hardware can be simulated using an event based simulation approach. This method is similar to the simulation mechanisms used in most VHDL simulation environments. SystemC defines four main constructs that can be used in hardware modeling. These are listed below:

- Module: This represents a single hardware entity.
- Port: Hardware entities are interfaced through the use of ports.
- Channels: Channels provide an event driven communications mechanism.
- Data types: Various data types are provided that are useful in hardware design.

SystemC was designed to be highly extensible so that all of the classes that are defined can be extended and new ones can be defined. As an example, a new channel could be declared to model the behavior of a specific type of real-world communications channel. The same applies to data types. As long as a new data type derives from the data type interface it can be used as the type for all SystemC ports and channels. Allowing for widespread extension of the SystemC classes for use within a specific domain.

1.2 VHDL

VHDL is a hardware description language, the acronym stands for Very high speed integrated circuit Hardware Description Language. It was originally developed at the behest of the US department of Defense. They needed a more efficient manner to document the behavior of ASIC designs that supplier companies were presenting. It was developed as an alternative to the huge, highly complex manuals that were the norm.

The idea to simulate such a description was immediately an attractive prospect and logic simulators were developed that could take a VHDL description as input. From here synthesis to the hardware level became a focus of study and still is. Current state-of-the-art synthesis tools can create hardware for a large subset of the VHDL language and should at least support the synthesizable subset in IEEE1076.6-1999[16].

The syntax of VHDL was derived from Ada, with constructs added to handle the parallelism that appears in hardware designs. The language is strongly typed and case insensitive. The first version of the language that was presented in the IEEE standard 1076-1987 [14] was somewhat limited and a new version appeared in 1993 in IEEE standard 1079-1993 [15], which improved the consistency of the language. There have been various reviews and updates to the standard of the languages since, but the 1993 version is the most prominent and widely used. Both versions of the language ignored the need for a multi-valued logic type which is where the IEEE standard 1164 comes in. This standard defines a 9-valued logic type called *std_Logic*. This has become the standard type to use for multi-valued logic in VHDL descriptions. The language has

been further extended in a series of libraries. One such library is the *numeric_std* library which implements a signed and unsigned type. These types allow for arithmetic to be performed on arbitrarily sized logic vectors.

1.3 Background and Related Work

The synthesis of SystemC models has been provided by a series of commercial synthesis tools. Most notable is the CoCentric System Studio[25][26] that provides a direct synthesis path from SystemC. Tools capable of providing the translation to traditional HDLs also exist in the commercial field, such as Prosilog's SystemC to VHDL/Verilog compiler[4]. While such tools are available, they exist in the commercial domain and expanding on their capabilities for academic and research purposes is not possible. In the public domain, a tool exists that provides a capability similar to our proposal. This tool, named SC2V[6], provides a translation from SystemC to Verilog. The translation provided is between register transfer level (RTL) SystemC and RTL-level Verilog. However, SC2C is very restrictive on the input it accepts and makes no attempt to transform nontrivial constructs (such as loops) to a format suitable for a synthesizable HDL.

The translation of loops to a finite state machine(FSM) was proposed as a method to map loops to an RTL-level model in [9]. Mapping a loop to a 3-state state machine allows the loop to be represented directly in an RTL-level model. In essence, this is what is normally done manually in the design process when the model is lowered to RTL-level. While this mapping of loops has been presented in [9] and it mentions the possibility of generalizing the method for all control structures, this generalization was not elaborated. In this paper, we propose an extension to this method capable of handling all control structures and any combination of control structures.

Finally, synthesis of object-oriented paradigms has been performed with some success[12][11][19]. Using an extended version of SystemC, the object-oriented designs can be translated to standard SystemC without classes. Therefore, we do not focus on the synthesis of object-oriented paradigms as they can be translated to regular SystemC using already proposed methods. In turn, this can serve as input to our proposed software tool.

1.4 Problem Statement

Hardware/software co-modeling has become an important method in the rapid design of systems. The use of SystemC for such modeling is gaining support amongst users and providing a synthesis path from SystemC to the hardware implementation could remove a current bottleneck in the design flow. In this thesis we are therefore concerned with the following problem statement.

- Identify the subset of SystemC that can be described using synthesizable VHDL. Then design and implement a tool to translate this subset to synthesizable VHDL models.

In the context of this thesis, synthesizable VHDL will be taken to mean the subset of VHDL that is synthesizable according to the synthesizable VHDL subset of IEEE1076.6-1999[16]. The version of SystemC used throughout this thesis is SystemC 2.1.v1 as described in [23] and C++ is taken to be ISO C++ as formalized in [18].

1.5 Thesis Overview

In this section the framework for this thesis is presented. The thesis is laid out as follows:

- Chapter 2 analyzes SystemC as a language and identifies restrictions that need to be placed upon the language to make it synthesizable.
- Chapter 3 presents the details of the translation from SystemC to VHDL. All aspects of the translation are dealt with from an abstract level leaving the implementation details for Chapter 4.
- Chapter 4 provides the details of the implementation of the translation tool using the methods presented in Chapters 2 and 3. We provide an overview of the internal structures used in the tool and present the important algorithms used to implement the translation.
- Chapter 5 contains the results of translating some test cases using the tool. We use a range of test cases that represent real world use of modeling languages and perform a pre- and post-translation simulation to verify the correctness of the translations.
- Chapter 6 concludes by presenting an overview of the results of this thesis and highlights important points. We present areas of further work and list the primary contributions made.

Synthesis and SystemC

Translating SystemC to synthesizable VHDL amounts to synthesizing SystemC directly. Using VHDL as an intermediate representation does not change that the resulting tool to translate SystemC will, in essence, be a synthesis tool. The structure of a synthesis tool could be as is depicted in Figure 2.1. This structure is the classical compiler structure[10]. Synthesis tools are a specific type of compiler, also known as hardware compilers. In our tool VHDL would take its place in Figure 2.1 as the intermediate representation. We then reuse existing tools to perform the back-end processing and perform the actual synthesis. Just as is the case with VHDL not all of the features of SystemC can be synthesized with mainstream synthesis tools. In VHDL this has led to a synthesizable VHDL standard as defined in IEEE Std 1076.6-1999[16]. With this in mind, we can argue the need to restrict SystemC in a similar manner. To remove or restrict features so that the usability is not impaired to a great degree while improving the ability to perform correct and efficient synthesis of the model to hardware. In our case to improving the correct and efficient translation to synthesizable VHDL.

In the following chapter we will discuss restrictions that can be placed on SystemC to make it expressible in synthesizable VHDL. We will start with SystemC itself in Section 2.1 and then continue with C++ in Section 2.2. In Section 2.3 we finish with some concluding remarks about the discussed restrictions.

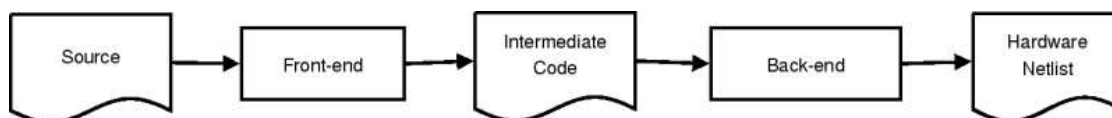


Figure 2.1: General Compiler Structure

2.1 Synthesizable SystemC

SystemC is not actually a modeling language in the strictest sense. It is only a set of classes that define a certain behavior. From a programming perspective this set of classes does not constitute a language in itself. Our goal, however, is to translate the behavior that is defined by using these classes. Since we do not want to compile or translate the SystemC library itself we need to attribute a specific semantic meaning to each of the classes or set of classes that SystemC defines. In other words, we will take the behavior that a certain SystemC class defines and use its semantic meaning in the translation while considering the class itself as an atomic entity. An example being the class *sc_signal*; this class being a SystemC class of which it is known that its semantics are the same as the VHDL signal type. This concept is used on the entire SystemC

library with all methods and operators that exist having a known semantic behavior which is expressed in VHDL in some manner.

In this section we will look at the various constructs that appear in the SystemC standard. We will look at four main constructs starting with the module in Section 2.1.1. We follow with channels and data types in Sections 2.1.2 and 2.1.3. For each of these constructs we will discuss the restrictions that are needed to make them compatible with synthesizable VHDL.

2.1.1 Module

A hardware component is described in SystemC by declaring a class that extends from module. This class functions as the container for the behavior of the component. Within this class ports, channels, submodules and processes are defined that describe the behavior of the component. In its generic form SystemC allows for hierarchical models where a module can be derived from another type of module, see Figure 2.2. It still derives from the SystemC default module and as such is a valid SystemC module declaration. However, the usefulness of this in a hardware model is disputable. For how we can reconcile such a concept with hardware is unclear at best.

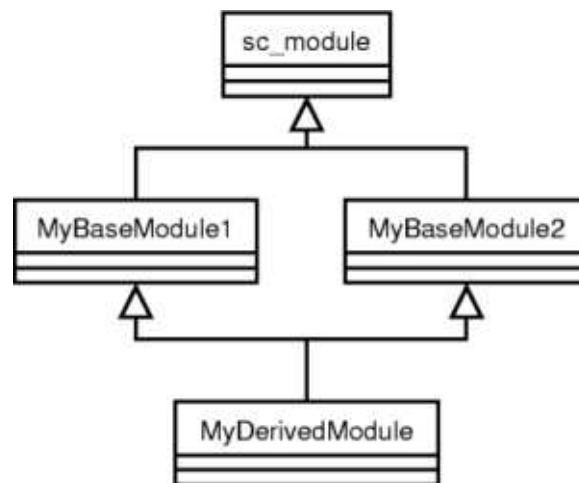


Figure 2.2: Module Inheritance

The SystemC module concept maps almost directly to VHDL in the form of an entity/architecture pair. Hierarchical models are, however, not directly mappable. A traversal of the inheritance lattice would be required and all of the modules in this lattice would need to be merged in some fashion. Since the usefulness of inheritance structures is disputable, and seems to be best avoided, we will restrict models to be uniquely and directly derived from the SystemC default module.

Ports that are declared within a module should be the only interface of the module. The SystemC language manuals[23][22] also state that communication between modules should be performed through ports. However, there is no built-in mechanism to enforce this. There is nothing stopping a module accessing an internal signal of a submodule as long as that signal was declared public. It greatly confuses the model to use such access

into a module and as such we will assume that only ports and the default constructor are declared public. The rest we will assume is declared private and as such only usable internally. Consequently conforming to the VHDL entity construct.

2.1.2 Channels

Channels are SystemC's generalized version of signals. In SystemC a new channel type can be created that has complex behavior that goes beyond that of ordinary signals. An example of this is the SystemC predefined channel *sc_fifo*. This channel, just as its name implies, acts as a FIFO buffer. Items written to it are placed in a queue and read out on a first in/first out basis. This extensibility of the channel type creates a problem with the semantic definition of the SystemC library. The channel type does not fully define its behavior, it acts more as an interface. Most of the behavior is only defined in the class that implements it. We can therefore not directly translate all channels but only channels for which the behavior is known and can be represented in VHDL. A class which does this is the *sc_signal* class. This class is an example of a channel type of which the behavior is fully defined. More than being fully defined, *sc_signal* is the only predefined channel type that can be directly represented in VHDL. The other channel types, as listed below, can only be represented by at least inserting a VHDL component or a process to represent the behavior of the channel.

- *sc_buffer*
- *sc_clock*
- *sc_fifo*
- *sc_mutex*
- *sc_semaphore*
- *sc_event_queue*

While the insertion of a component could represent the behavior of the channels it cannot represent the timing of the channel. SystemC channels do not time their internal logic in any way; it is assumed to take zero time. The insertion of a component to represent this channel will, however, introduce a considerable amount of logic that was previously not considered. This could easily cause a considerable amount of difference in the pre- and post-synthesis designs. The *sc_buffer* and *sc_fifo* channel types could be represented by the insertion of a component to handle the behavior. However, the implicit timing difference introduced outweighs their usefulness. It would be better if an explicit SystemC module is created to handle such behavior and to translate this together with the containing module. This way the timing is consistent across both models.

The *sc_mutex*, *sc_semaphore*, and *sc_event_queue* channels require more than just the insertion of a component. This is because they define locking semantics that cannot easily be represented in VHDL. The *sc_mutex* defines locking semantics that require knowledge of the process that is currently locking or unlocking the mutex. Knowledge of the writing or reading process is not available in VHDL. The channel *sc_semaphore*

presents yet a different problem, for this channel does not use the event model. It is a pseudo channel, since it acts in a fashion more like a variable. Its only purpose is to allow such a semaphore count to be available between modules. This is purely meant for simulation, because such a model is very risky to synthesize. Synthesizing such a model is asking for internal timing problems, seen as the timing of a signal was not taking into account in the design. While we might be able to simulate a zero delay signal such a signal can never exist in the actual implementation.

SystemC also supports a generalized mechanism for events of which the *sc_event_queue* is a part of. It supports the creation of events without using signals. We can see such an event in much the same way as a signal, except that an event carries no data. Such an event is, however, not much use in hardware. Signals are a much better mechanism to represent flags and events since they are not more complex to use but do represent the hardware better. There is no advantage to using events over signals and because of that we do not support them. A side effect is that the *sc_event_queue* is also not supported. The last predefined channel type is the *sc_clock*. This channel type has only one purpose, namely to create clock signals for test bench purposes. A SystemC model will generally only contain one such channel type and then only as a part of the test bench. As our goal is to generate VHDL for synthesis, translating test benches is not needed so *sc_clock* is not supported.

2.1.3 Data Types

Being a class library SystemC can use all of C++'s built-in types, and beyond that also declares data types of its own. These SystemC data types, as we will call them, are types that are useful in the modeling of hardware. The data types that SystemC declares are shown in Table 2.1. Most of these types are clear cut, and have a good representation in VHDL. Examples being *sc_logic* having a direct mapping to *std_logic* and *sc_lv* a mapping to *std_logic_vector*. However, SystemC also has a set of data types that do not have a mapping directly to a VHDL type. These are the fixed point data types *sc_fix* and *sc_ufix*. While a reasonable mapping can be made to a VHDL real, which is a floating point type, the mapping is not exact. Most importantly, the distribution of values in a floating point type is not even. The values that can be represented with a floating point type get sparser the further they get from 0. This is in contrast to a fixed point type, which has an even distribution along the number line. The difference is illustrated in Figure 2.3. Furthermore, VHDL reals are often not supported by synthesis tools, making them incompatible with our goal of generating synthesizable VHDL. For the above reasons the *sc_fix* and *sc_ufix* types are not supported. We do realize that a synthesizable implementation of fixed point data types can be made in VHDL and the SystemC fixed point types can be translated to this. We leave such an implementation open as an extension to the tool.

In the same fashion as is possible with channels, data types can also be defined by the user. These data types can define their own operators using C++ overloading and as such completely define their own semantics. As was true with channels this means we do not know the semantics of the data type beforehand and cannot create a mapping to VHDL. Consequently we will not support the definition of custom data types. Only the

SystemC Type	Description
<i>sc_int</i>	Width restricted integer
<i>sc_uint</i>	Width restricted unsigned integer
<i>sc_bigint</i>	Unrestricted integer
<i>sc_biguint</i>	Unrestricted unsigned integer
<i>sc_fix</i>	Unrestricted fixed point
<i>sc_ufix</i>	Unrestricted unsigned fixed point
<i>sc_bit</i>	Bit
<i>sc_logic</i>	4-value logic
<i>sc_lv</i>	<i>sc_logic</i> vector
<i>sc_bv</i>	<i>sc_bit</i> vector

Table 2.1: SystemC Data types

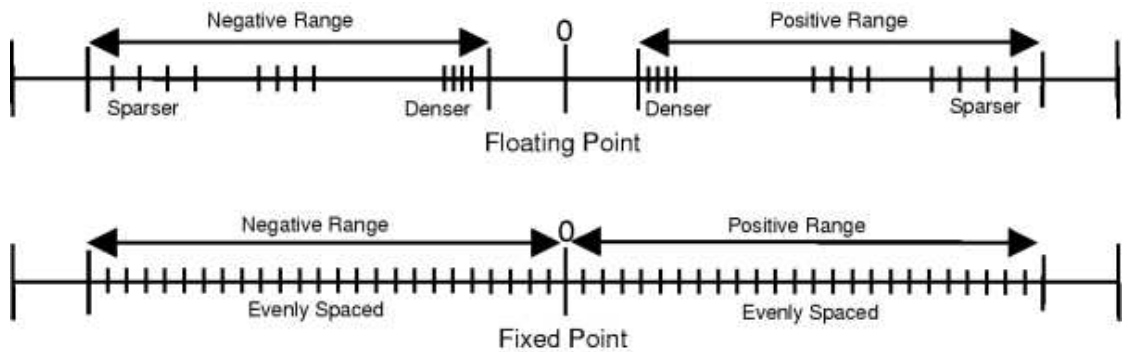


Figure 2.3: Range and distribution of floating vs fixed representations

predefined SystemC data types are supported. This does not mean that the definition of C++ data types is not supported. The support that is available for this is discussed in Section 2.2.3.

2.2 Synthesizable C++

As stated, since the SystemC library is built on top of C++ the translation also needs to translate general C++ constructs. C++ is a high-level object-oriented language containing many of the features expected in such a language. It supports an advanced group of object-oriented constructs from multiple inheritance to a general mechanism for operator overloading. Many of these kinds of features, while eminently useful in software engineering and also during hardware simulation, have little use in the creation of a synthesizable hardware design. The major language features of C++ will be examined in the following sections and the restrictions on their use will be explained. We start with classes, variables, types and functions as they are key elements of the language. We then continue with expressions and statements in Section 2.2.5. Finally we will discuss exceptions, pointers and classifiers/specifiers in Sections 2.2.6, 2.2.7 and 2.2.8

respectively.

2.2.1 Classes

Classes allow for code and data to be viewed in a conceptually appealing manner. They allow for the modularization of a program in a manner intuitive to software design. In hardware, however, such a modularization, as is allowed by classes, has no direct mapping. While a mapping of classes is possible as shown by Grimpe, et. al.[12] it falls outside the scope of this work. Restricting classes to be non-inheriting, makes a mapping to VHDL conceivable, but also has the effect of removing the primary benefits of the object-oriented paradigm. Considering that the behavior of such a class can easily be represented using a simple structure and a set of functions the addition of this is left as an extension. As mentioned in Section 2.1.1 in SystemC a module is declared by declaring a class that derives from *sc_module*. Therefore, we do need to parse class declarations, but we just do not deal with any class that does not uniquely derive from *sc_module*.

2.2.2 Variables

Looking from a high level there are only three types of variables. We have member variables that are a part of a class, in our case module. There are also global variables that are declared at the scope of the file or name space. Lastly, we have local variables, that are declared within the scope of a subprogram. While we must accept variables we will not accept variables that are declared at the global scope. This because such variables cannot be represented in VHDL and they break with our previous premise that the only communication between modules would occur through the ports. A side effect of not allowing global variables is that we also do not allow static member variables. Static member variables are semantically equivalent to global variables. Member variables can easily be represented in VHDL as variables declared in the scope of an entity. They then have the same semantics as a member variable in C++.

2.2.3 Types

C++ contains many standard types and as with the SystemC types most of these present no problems. They are all integers in effect, with the exception of the boolean and float. Floats and doubles are the only types that present a problem. There is no mapping to synthesizable VHDL for floats and doubles. Just as with the *sc_fix* and *sc_ufix* types we will not accept them, but leave the possibility of a VHDL implementation of a synthesizable floating point type open. Complex types such as enumerations, structures and unions all have a mapping to VHDL. Enumerations either in the form of a VHDL enumerator or in the form of a series of constant declarations. Structures and unions can be mapped to a VHDL record type with relative ease. Mapping a union to a record is semantically valid, since the union only differs from the structure in how it is laid out in memory. Mapping it to a record is not efficient but will preserve the semantic meaning.

Casting is an integral part to types. As we will discuss in Section 2.2.7 we will not support pointers in their general form. This entails that the only types of casts

that are possible are casts to and from integral C++ or SystemC types. All of these can be supported by including some conversion functions in the VHDL model. These functions will allow for these casts to be done, using the same conversion semantics as are defined by C++ and SystemC. This includes the named casts **reinterpret_cast** and **static_cast**. The named cast **dynamic_cast** will not be supported. The reason is that it has no use without a general form of classes and pointers. It is meant to be used to provide a safe runtime checked method of casting between a pointer to a base class and a pointer to a derived class. This is meaningless if we do not support classes and pointers.

2.2.4 Functions and Methods

Within a SystemC model we encounter three types of functions and methods, collectively called subprograms. The first of these is the constructor of the class defining the module. The SystemC standard advises that modules should use their constructor to perform any elaboration of the model. Examples of the actions that can take place are the binding of ports to signals or declaring a method to be a process. While it is possible to perform these initializations outside of the constructor in some sort of separate initialization method, this is not advised and also completely unnecessary. So we will assume that the initialization occurs in the constructor of the module.

The second type of subprogram is a process method. A process method is a method with no arguments, that has been declared as a process in the module's constructor. These types of subprograms need to be translated to processes in VHDL. The translation should amount to the translation of statements and expressions, to some equivalent in VHDL. Statements and expressions are dealt with in more detail in the next section.

The last type of subprogram is a non-process method or function. These are all the functions and methods that are not declared as processes, and are not a special method. Special methods are either constructors, destructors or operator overloads. Dealing with these subprograms is similar to dealing with processes, for C++ there is actually no difference between the two. The difference only occurs when the SystemC semantics are taken into account. The last type of subprogram also introduces parameters. We can map parameters to VHDL with relative ease, even reference parameters have a direct mapping using the **inout** keyword of VHDL. We encounter a problem with these reference parameters when we also consider the return value of subprograms. In VHDL only a procedure may contain an **inout** or **out** parameters. These procedures do not return any variables so we need to add an extra parameter to such procedures to represent the return value and declare that parameter as an **out** parameter. This will allow us to model the exact semantics of C++ subprograms while not restricting their use.

2.2.5 Statements and Expressions

The core of C++ as with many languages is in the statements and expressions. Logically this is where that actual behavior is defined. The expressiveness of C++ is in part due to the flexibility of its statements and expressions. An example of this is the **for** loop. The C/C++ loop construct is one of the most powerful and flexible loop constructs around. These flexible constructs in C++ present a problem for the translation because for some of the constructs there is no direct equivalent in VHDL. The **for** and **switch** statements

are good examples. While VHDL contains both a **for** loop and a **switch** statement, called **case** statement in VHDL, they are not nearly as flexible as the C++ versions. This means we need to transform these statements to some other form. Even more restricting is that synthesizable VHDL does not support the use of loops which cannot be equated at synthesis time. This entails that all loops cannot be mapped directly to synthesizable VHDL. A solution for this is to map the loop onto a finite state machine as proposed by Cote, et. al.[9]. This will be the approach we take in the tool to translate loops into synthesizable VHDL. Expressions in C++ are less of a problem to translate to synthesizable VHDL. Almost all the expressions that refer to some basic operator have a direct mapping to VHDL. The only real exceptions to this are the incrementation operators since VHDL does not have any such operator. However, this can be dealt with by replacing the expression with a temporary variable and assigning the correct value to this temporary variable. A similar construction is needed for function calls since we are returning the return value of a function as an **out** parameter. Replacing the call with a temporary variable and calling the function with this temporary variable as its return argument will also maintain the correct semantics.

2.2.6 Exceptions

The exception mechanism in C++ allows programs to generate and catch events. These events are passed down the call stack to the first subprogram that catches them. This mechanism allows programs to flag errors in an easy and intuitive manner. The mechanism is meant to catch errors such that they can be dealt with in a graceful manner. While some exceptions have their roots in hardware, for example a divide by zero exception, the exception mechanism has limited use in a SystemC model. Since any exception generated during the execution of a process would need to be dealt with in the process itself. If this is not done the exception would travel down the call stack to the SystemC simulation kernel and be caught there. This entails that all exceptions that could be generated are purely local to a process. This makes the C++ exception model rather limited in application in SystemC models, be that a simulation model or a synthesis model. Because the exception model has such limited use in SystemC, and a general method for dealing with it would be complex, we disallow the use of exceptions in the models that are to be translated.

2.2.7 Pointers

In the introduction we briefly mentioned pointers. While pointers have no intuitive meaning in hardware and have no real place in the definition of the module, there is one type of pointer we must support. It is not uncommon for a member to be accessed through the implicitly declared **this** pointer. Access is then performed using the dereferenced access operator $->$. We should accept such uses of this operator, so that the **this** pointer can be used. Semantically any access to a member from a method uses this operator, except that it is usually implicit and hidden because of the scoping rules. This is the only C++ element that relates directly to pointers that we will accept in the module to be translated. Since we do not accept pointers in a general form, we also do not accept any use of heap storage. Heap storage has no mapping to hardware since

an element of hardware either exists or it does not. We cannot dynamically create or destroy hardware elements. Realize that we are not considering whether the element is in use or not but whether it exists at all.

2.2.8 Specifiers and Classifiers

There are a couple more C++ elements that need to be mentioned. First we have the access specifiers in a class. In the module class we could encounter access specifiers of the form **public**, **protected** or **private**. These serve to define what members may be accessed by whom. They have no meaning in the function of a program. They only protect the user from possibly dangerous or unsupported uses of a class. While they have no use it might be useful to enforce all members to be 'private' except the constructor and ports. This would help to enforce our intra-module communication restrictions. Any declaration in C++ may be pre-pended by some storage classifiers. These classifiers serve to tell the compiler of the program how to store the declarations. For example the **register** classifier would tell the compiler to attempt to store that variable in a register. These classifiers have no effect on the behavior of the program. They only serve to aid the compiler in making efficient choices in the placement of variables, as such they can be safely ignored.

2.3 Conclusion

Mapping SystemC to VHDL can be accomplished with an amount of restrictions on the SystemC input. While the removal of many of the dynamic features of SystemC and the more abstract elements of C++ reduces the expressiveness of these languages, we feel that it does not greatly impair their use in the description of hardware models. Especially if one considers that altering an existing model so that it conforms to the restrictions will be easier than rewriting that model in VHDL manually. Since in that latter case the entire model needs to be rewritten when this might not be needed to create a SystemC model that conforms to our restrictions.

In this chapter the mapping between SystemC/C++ constructs and VHDL are defined in detail. For each class of construct we lay out the general approach and present the mapping patterns that will be used to perform the translation. We start with types in Section 3.1 because they are the simplest and lead us into declarations in Section 3.2. From there we move to statements in Section 3.3 and expressions in Section 3.4, where we will show the mapping of loops to a finite state machine. Finally we will deal with the extraction of the model structure from the module constructor in Section 3.5. We sum up our findings in Section 3.6.

3.1 Types

As we stated in Sections 2.1.3 and 2.2.3 types are in general directly mappable. The only complication is with the SystemC types. In SystemC the types can be declared with a fixed width. The width of the type can be defined in three different manners. The first is using a templated version of the type. With this method the parameters of the type are defined at compile time. The second is to declare it using the constructor of the type. This passes the parameters of the type during construction. The final method is to use the default width of a type and locally changing the default. This can be done using the SystemC context mechanism. With this mechanism the default for the types can be set locally. Every type created after this change will use the new defaults. Once the context goes out of scope the original defaults become active again. All three methods are shown in Listing 3.1.

The templated context method is very clear and concise. The width of the type is specified with the declaration and is constrained to be a literal value. In other words the type is fully specified by its declaration. Specifying the width during construction splits the specification of the type from the declaration. It allows the width of the type to be specified by parameters passed to the module in its constructor. This method is unsupported because we only support the default method of declaring the constructor with *SC_CTOR*, which does not support parameters. There is then no reason to support the constructor method, since it has no advantages and only splits the declaration up. The method of using the current context to declare the width of the type is also unsupported. The reason is that this method splits the specification of the type to the point where the actual width of a type can be declared almost anywhere. There is even the possibility for the type to be specified during the simulation. For the mapping to work the types must be fully specified during the translation and so we cannot support a mechanism that specifies the types during either elaboration or simulation. We will therefore only accept the types that use templates to specify their parameters. Even though SystemC does support the other mechanisms, the language manual implies that types should be

```

SC_MODULE(myModule)
{
  /* sc_int_base width specified in constructor */
  sc_int_base myConstructedInt;
  /* sc_int width specified by template parameter */
  sc_int<20> myTemplatedInt;

  SC_CTOR(myModule) : myConstructedInt(20)
  {
    /* sets the current length context to 20 */
    sc_length_context lengthContext(20);
    /* sc_int_base width set by length context */
    sc_int_base myDefaultInt;
  }
}

```

Listing 3.1: Test Listing

specified using templates by the naming conventions used. The templated types are the types which are specified with direct names such as *sc_int* and *sc_lv* while the types that support the constructor and context method have names like *sc_int_base* or *sc_lv_base*.

Since the types defined using templates are fully specified during the declaration that uses them, we just need to read the template parameters and adjust them to match with VHDL semantics. SystemC supports 3 types of integers in both a signed and unsigned form. These are the **int**, *sc_int* and *sc_bigint* types. The first is the integral integer type of C++ and can be mapped directly to a VHDL *integer* type. The second two types are the integer types from SystemC. These contain more features such as the ability to select single bits and ranges of bits. While SystemC has two integer types they can be considered the same. The only difference between the two is the method used to implement them. The *sc_int* type is limited in the size that can be used, but is fast as a consequence. The *sc_bigint* type is slower but supports an unlimited size. The best mapping for both of these types is the *SIGNED* and, for unsigned versions, the *UNSIGNED* types from the *numeric_std* library. This library implements an unlimited integer type as an *std_logic_vector* and thus supports bit selects and range selects. The bit width of the *SIGNED* and *UNSIGNED* types are specified directly just as it is with logic vectors. All vector types can be mapped with the mapping shown in Listings 3.2 and 3.3.

```

vec_type<width> ident;

```

Listing 3.2: SystemC Vector Type

Unlike SystemC, which only specifies the width of the vector, VHDL specifies the lower and upper bound. Converting between them is a trivial matter. All that needs to be done is to define the range to run from *width-1* **downto** 0. The minus one corrects

```
ident : vec_type(O to width - 1);
```

Listing 3.3: VHDL Vector Type

for the indexing differences between SystemC and VHDL. We use a reverse range because the SystemC library uses the same convention of numbering vectors from right to left. So that bit 0 of a *SIGNED* type is the least significant bit.

All the types and their mapping to VHDL are shown in Table 3.1. The functions *low* and *high* represent the calculation of the minimum of the range and maximum of the range respectively.

Source Type	Destination Type
<i>char</i>	<i>integer range(low(8) to high(8))</i>
<i>int</i>	<i>integer range(low(32) to high(32))</i>
<i>short int</i>	<i>integer range(low(32) to high(32))</i>
<i>long int</i>	<i>integer range(low(32) to high(32))</i>
<i>bool</i>	<i>boolean</i>
<i>sc_int</i> < <i>w</i> >	<i>signed(w-1 downto 0)</i>
<i>sc_uint</i> < <i>w</i> >	<i>unsigned(w-1 downto 0)</i>
<i>sc_bigint</i> < <i>w</i> >	<i>signed(w-1 downto 0)</i>
<i>sc_biguint</i> < <i>w</i> >	<i>unsigned(w-1 downto 0)</i>
<i>sc_bit</i>	<i>bit</i>
<i>sc_bv</i> < <i>w</i> >	<i>bit_vector (w-1 downto 0)</i>
<i>sc_logic</i>	<i>std_logic</i>
<i>sc_lv</i> < <i>w</i> >	<i>std_logic_vector (w-1 downto 0)</i>

Table 3.1: Mapping of Types

The mapping of compound types such as record types and structure types is a little bit more complex. The declaration of the new types in VHDL cannot occur in the same manner as it can in C++. New types in VHDL must be declared in a declarative region and they must be declared before they are used. In C++ we can introduce a new type in a declaration of a variable. So the C++ fragment in Listing 3.4 is valid while the VHDL fragment in Listing 3.5 is not.

```
struct { int a; int b; } myvar;
```

Listing 3.4: SystemC Struct Declaration

This distinction means that we need to extract all the compound types and declare them with a generated unique name. To accommodate this, every compound type that is found in a declaration is mapped to the equivalent in VHDL and all references to this type use a generated name to refer to it. The structures defined in Listing 3.6 are mapped to the structures defined in Listing 3.7. The VHDL translation is split into two sections. One is a generated package that declares all compound types called *scv_gen_types* and

```
myvar : record is { a : integer; b : integer };
```

Listing 3.5: VHDL Vector Type

the actual usage of these types.

```
struct MyStruct1
{
  int a;
  int b;
}

MyStruct1 myvar1;
struct { int a; int b; } myvar2;
```

Listing 3.6: C++ Structures

```
package scv_gen_types is
type recordtype1 is record
  a : integer range .. to ..;
  b : integer range .. to ..;
end record recordtype1;

type recordtype2 is record
  a : integer range .. to ..;
  b : integer range .. to ..;
end record recordtype2;
end package;

myvar1 : recordtype1;
myvar2 : recordtype2;
```

Listing 3.7: VHDL Mapping of records

While the two record types might be structurally equivalent, both C++ and VHDL maintain named equivalence of compound types. So the fact that the types are declared distinct is semantically correct. Since unions are just structures with a different memory mapping we can use the same method to map unions.

Enumerations in C++ are not really enumerations, they are pseudo enumerations. The enumeration is not truly a value from a predefined set. In C++ the use of enumerations is more consistent with how they could be and usually are implemented. So the following two C++ code fragments are equivalent and can be used interchangeably.

```
typedef enum { a, b } myenum;
```

```
typedef int myenum;
const int a = 0;
const int b = 1;
```

As the listing above showed C++ enumerations do nothing more than declare a series of constants and define the type to be an integer. The size of this integer is the only difference, since the compiler will scale the range of the integer to fit the amount of enumerators. VHDL enumerators are true enumerators, in the sense that the enumerators are the literal values that can be assigned and not as in C++ just constants that equal some integer. Even though VHDL enumerators might be implemented with a method like what is used in C++ the VHDL language user does not see it this way. This contrasts with C++ where we can make use of the knowledge that enumerators will be integers. We can even specify the integers the enumerators should represent. While this difference may seem just theoretical it has an important implication, namely that casting to and from integers is not possible in VHDL. Casting enumerators to integers has no semantic meaning in VHDL. Because the possible values of an enumerator in VHDL are not numbers but elements from the set of enumerators. To map the C++ enumerator to VHDL we need to make sure that all the semantics of the C++ enumerator are maintained. A method of achieving this is to not map them to enumerators but to map them to a set of constant declarations and have all uses of the type be declared as an integer with a suitable range. We use this approach to map enumerators to VHDL as is shown Listing 3.8, this fragment shows the VHDL mapping enumerator declaration in the previous code fragment.

```
constant a : integer = 0;
constant b : integer = 1;

myvar1 : integer range(0 to 1);
```

Listing 3.8: VHDL Vector Type

The final type that has thus far not been mentioned is the array. With our restriction on pointers the array type in C++ is required to be fully specified in the declaration. We do not have unsized arrays or index operations on pointers as we otherwise might have. This means that the C++ array is equivalent to the VHDL array. Just as with the vector types, which are actually just array types in VHDL, we can get the sizes of an array from its declaration. This applies to both single and multi-dimensional arrays. We can then declare an equivalent array in the VHDL translation. Listings 3.9 and 3.10 illustrate this mapping. Note that in the case of arrays we do map the indexes in ascending order since this is the expected ordering of a C++ array.

```
int myarray[10][20];
```

Listing 3.9: SystemC Array

```
myarray : array(0 to 9, 0 to 19) of integer;
```

Listing 3.10: VHDL Array

3.2 Declarations

Declarations concern any part in the code where a new identifier is introduced into the scope. For the purposes of the translation this only refers to variable, member, and subprogram declarations. The reason for this is because the translation will take place by scanning the module constructor and filling in all the needed information from that point on. This will be dealt with in detail in Section 3.5. It will not translate all declarations found in the source. This means that the only form of declaration we will need to deal with are subprogram declarations, variable declarations within a subprogram and member declarations in the module.

In C++ and in VHDL all variable or member declarations take on the same form so that they can be translated in the same fashion. Member declarations in the module are mapped to declarations in the declarative region of the architecture in VHDL. Variable declarations in a process or subprogram are less trivial. This is because C++ is a language where local variable declarations may occur anywhere where a statement may occur. This entails that variables can be declared throughout a subprogram. In contrast, VHDL requires all declarations to occur within a declarative region at the start of a process or subprogram. This is reminiscent of ISO C where all declarations must occur before any other code in a block. To deal with this we need to move all declarations found within the body of a subprogram to the declarative region at the start of the respective process or sub-program. Listings 3.11 and 3.12 illustrate this. These also illustrate the problem with a naive approach to moving the variable declarations.

```
SCMODULE(mymodule)
{
int a = 0;
int b;

void myprocess()
{
  if(true)
  {
    int a = 1;
  }
  b = a;
}
}
```

Listing 3.11: SystemC Declarations

As we can see just moving all declarations to the start of the process does not maintain

```
architecture SystemC of mymodule is
  shared variable variable a: integer = 0;
  shared variable variable b: integer;
begin

  myprocess : process is
    variable a : integer = 1;
  begin
    if(true)
      —Empty
    end if;
    b := a;
  end process myprocess;
```

Listing 3.12: VHDL Declarations

the scoping rules of C++. In the C++ fragment the second declaration of *a* goes out of scope before *b* is assigned the value of *a*, thus *b* will be assigned 0. In the VHDL fragment we moved the declaration of *a* to the declarative region of the process. Now the second declaration of *a* is not out of scope during the assignment of *b* and so *b* is assigned to 1. Clearly this is not the intent and we will need to modify the names of declarations to reflect that they are declared in different scopes. The idea is that whenever we encounter a declaration that is not at the scope of a declarative region, for example it is inside a block statement such as the if statement, then we append the variable with a number that is incremented upon use. Consequently, all variables declared at a scope where we can also declare in VHDL maintain their original identifier. Since this is the most common scope to declare in the majority of the translation will be direct. However, when we need to declare something that cannot be declared at the current scope we make the name unique by appending a unique number to it.

There is another concern with declarations. More specifically with the identifier introduced by the declarations. C++ is a case sensitive language such that *a* is not the same as *A*. In contrast VHDL is case insensitive so the two identifiers are equivalent. In order to ensure that names that differ only in case remain distinct in VHDL we use a name map during the translation. If we encounter a declaration we check whether the name has been declared before, using a case insensitive search. If so, then if the names are not equivalent in case then we append a number to the identifier. The number that is appended is the current amount of identifiers that have matched the original identifier. So if the identifiers *A*, *a*, *B* and *b* have been declared in C++ they will be *A*, *a2*, *B* and *b2* in VHDL. Scope is not taken into account so that even if a name clash would not occur in VHDL due to the scope of the declaration a number is appended anyway. This does not affect the correctness of the translation but greatly simplifies the mapping. Since users generally do not use two names that only differ in case it should not occur often.

VHDL supports two types of subprograms; functions which cannot accept reference parameters but return a value and procedures which can accept reference parameters but

do not return a value. In contrast in C++ all general subprograms in C++ can both accept reference parameters and return a value. As we already mentioned in Section 2.2.4 we can create a generic mapping for C++ subprograms but adding an extra **out** parameter to the VHDL procedure to return the function's return value. Using this method we can map all combinations of reference parameters and return values that can be made within the other restrictions we have placed on the input.

3.3 Statements

The most involved parts of translating SystemC to VHDL is the translation of statements. VHDL is much less general in what is possible with control statements than C++ is. Furthermore, synthesizable VHDL is even more restrictive by disallowing the use of most loop statements. More specifically it disallows the use of any loop that cannot be fully unrolled and compile time. While many loops can be unrolled at compile time there are also plenty of cases where it is not possible. We need some way of dealing with these loops, because we cannot set the restriction that such loops are not allowed. We do not wish such a restriction because it can greatly reduce the expressiveness of C++ and using such loops to represent certain hardware constructs can provide a much more compact description.

As already mentioned in Section 2.2.5 we will translate loops that cannot be unrolled to a finite state machine. Such a translation will allow the general behavior of these loops to be expressed. However, there is a concern with this translation in that it will not maintain the timing of the model. The timing changes introduced can also not be predicted, if they could be we would be able to unroll the loop. This means that such loops should be used with care and knowledge of the possible impact. We will be looking more closely at the impact that this FSM translation has in Chapter 5. Given a process that takes on the form shown in Listing 3.13 we can map that to a state machine described in Listing 3.14 and depicted in Figure 3.1.

```
myprocess ()
{
  <pre loop statements>
  while(<condition>)
  {
    <body>
  }
  <post loop statements>
}
```

Listing 3.13: SystemC Process with a single loop

There is more to the generation of a state machine in VHDL, like the process to handle clocked state changes in the case of a synchronous state machine, but we will ignore those for now. As is shown in the example, the mapping of a loop to a finite state machine is not very complicated and the timing change that is involved is proportional

```

myprocess : process is
    state : stateenum;
begin
    case(state)
    when state0: <pre loop statements>
        if(<condition>)
            state <= state1;
        else
            state <= state2;
    when state1: <body statements>
        if(<condition>)
            state <= state1;
        else
            state <= state2;
    when state2: <post loop statements>
        state <= state0;
    when others: state <= state0;
end process myprocess;

```

Listing 3.14: VHDL State machine for single loop

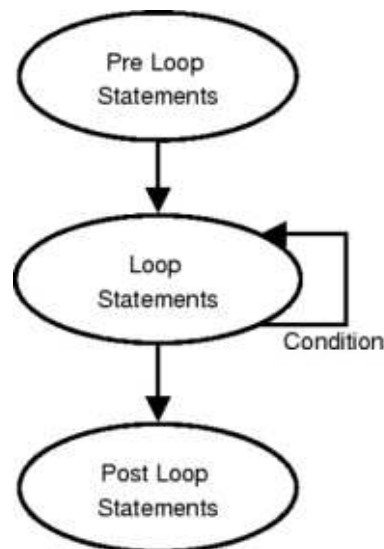


Figure 3.1: Finite state machine of C++ Loop

to the amount of iterations the loop makes at runtime. In the case of a synchronous model we could drive the state machine on the global clock of the model. However we could also very well introduce an extra port into the model that will be used to deliver a clock specific to the state machine. This might allow for faster implementations since each run of the state machine processes is faster than the full run of the original loop.

For example, a SystemC process that contains a loop with 10 iterations should in

principle take around 10 times longer to complete than one iteration. Then each run of the FSM process will take only the time of a single iteration so that the clock driving the state machine can be clocked faster. Since we cannot derive the timing of a single iteration, we have no hardware knowledge during translation, we cannot know the characteristics of this new clock signal. This must be determined by the user in some manner. In Chapter 5 we show an example of a model that has been translated using a finite state approach.

The complexity of the state machine increases when we take into account other statements that might be a part of the process. This is highlighted by the example in Listing 3.15 and 3.16. In this example the for-loop is embedded in an if-statement.

```

myprocess ()
{
  <statements1>
  if(<condition1>)
  {
    <statements2>
    while(<condition2>)
    {
      <statements3>
    }
    <statements4>
  }
  <statements5>
}

```

Listing 3.15: Loop embedded in an if-statement

As we can see in the mapping of such an embedded loop as shown in Listing 3.16, we cannot translate this with only the mapping that we have for loops. The mapping that was given for loops assumes that all the statements following the loop could be considered as a statement list. In the process of Listing 3.15 this is not true, since it is not given that *<statements4>* and *<statements5>* will be executed in sequence. This entails that the generation of the state machine cannot be done only at the level of the current block of statements. We also need to consider what happens to all the statements that follow the block and all the statements that lead up to the block containing the loop. A correct mapping of the above example is given in Listing 3.17. In the mapping we introduce an extra state to handle the code that follows the if-statement. Now we can move to the correct state immediately in the initial state and the states handling the loop can also correctly move to this state when they are complete. In general this means that if during the translation of a control statement body a new state is introduced then we need to add a new state for all the statements following that control statement and switch to this state in the appropriate places. The correctness of this approach can be shown if we create a control flow graph of the process. Such a graph for the process in Listing 3.15 is given in Figure 3.2. The control flow graph is equivalent to the finite

```
myprocess : process is
    state : stateenum;
begin
    <statements1>
    if(<condition1>)
        case(state)
            when state0: <statements1>
                if(<condition1>)
                    <statements2>
                    if(<condition2>)
                        state <= state1;
                    else
                        state <= state2;
                    end if
                end if
            <statements5>
            when state1: <statements3>
                if(<condition2>)
                    state <= state1;
                else
                    state <= state2;
                end if
            when state2: <statements4>
                state <= state0;
            when others: state <= state0;
        end case
    end if
    <statements5>
end process myprocess;
```

Listing 3.16: Naive mapping for an embedded loop

state machine produced by our method. In fact, a method to generate the finite state machine could be to generate such a control flow graph, since it makes little difference if the control flow is implemented in jump instructions for a processor or in a dedicated state machine. We can thus emulate jump instructions within the state machine. While this method would allow us to support the use of goto-statements we still feel that such a statement has no place in SystemC.

For each of the control statements (if, switch and for) we describe the method used to map them. We use the same mechanism for all the loop statements(for,while and do) so we only consider one of them. The only difference between them is where the conditional state changes occur and in the case of the for-loop we need to append the iteration expression.

```

myprocess : process is
    state : stateenum;
begin
    case(state)
        when state0: <statements1>
            if(<condition1>)
                <statements2>
                if(<condition2>)
                    state <= state1;
                else
                    state <= state2;
                end if
            else
                state <= state3;
            end if
        when state1: <statements3>
            if(<condition2>)
                state <= state1;
            else
                state <= state2;
            end if
        when state2: <statements4>
            state <= state3;
        when state3: <statements5>
            state <= state0;
        when others: state <= state0;
    end case
end process myprocess;

```

Listing 3.17: Correct VHDL State machine for an embedded loop

3.3.1 If Statements

The method for the if-statement performs no state translation unless either the then-body or the else-body introduce a new state. This new state can only be introduced if those bodies contain loops. If a state is introduced then we create a state for the statements following the if-statement called the "tail state". This state will collect all the statements translated after the if-statement. We also add state changes to this state at appropriate places: at the end of the state introduced by the mapping of the then-body or, in the case of the else-body, we might need to add a state change at the end of the else-body when it does not introduce a new state but the mapping of the then-body did. This will faithfully produce a normal if-statement if no states are introduced in the then- or else-bodies, but will append the state changes if they are needed.

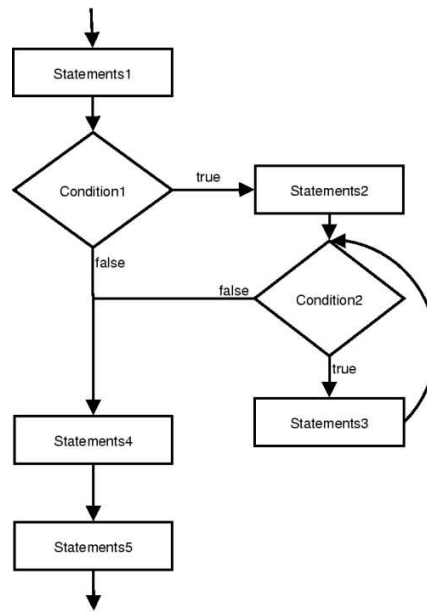


Figure 3.2: Control Flow Graph of Embedded Loop

3.3.2 Switch Statements

The mapping of switch-statements works in a manner similar to if-statements. We map the body of each case in turn and add state changes when a new state is introduced or was introduced by earlier cases. This ensures that the statements trailing the switch-statement are executed appropriately when a case completes. We still need to address the difference between C++ switch-statements and VHDL case-statements. The main difference is that in C++ the case does not end until it reaches a break-statement. In VHDL there is no such break-statement possible inside a case-statement. Furthermore the case ends at the subsequent case. So when mapping the case we need to map all statements in the case until the first top level break-statement and we need to introduce new states when a break-statement is found that is embedded in an if-statement. We can then move to the state at the end of the switch-statement when the break occurs and move to a state containing the remaining statements of the case if it does not.

3.3.3 Loop Statements

Loops are translated just like the first mapping we introduced, except that we need to track whether the body introduces a new state or not. If it does then we add the conditional state change to the introduced state instead of adding it to the local body state. We deal with the break- and continue-statements in a manner similar to the break-statements in switch-statements. For break-statements we introduce a new state for the statements following the break-statement and switch to either that new state or the tail state of the loop. Continue-statements are dealt with in the same fashion with the differences; that we move to the body state of the loop if it occurs, that we include the iteration statement and that we check the condition of the loop before we move the

start of the body state.

We will support the ability to override the finite state machine generation and perform another type of mapping. This mapping will cause all loops to be mapped to a general loop construct in VHDL. This secondary mapping of loops is shown in Listing 3.18. This mapping will not produce synthesizable VHDL but the result can be simulated. The method could be used to verify the translated model for any timing problems. It allows the a translation to be generated that should maintain the timing of the model and using this can be used to verify other parts of the model.

```

— Mapping of a while Loop
while <condition> loop
  <statements>
end loop;
—Mapping of a for loop
<for_init>
while <condition> loop
  <statements>
  <for_expr>
end loop;
—Mapping of do loop
loop
  <statements>
  exit when not<condition>
end loop;

```

Listing 3.18: VHDL Loops

3.3.4 Constant Propagation

As mentioned, the mapping of loops to a finite state machine introduces a timing problem in the translation. In an effort to reduce this problem we looked into maximizing the translator's ability to unroll loops instead of mapping them to a state machine. The conditions to unroll are as follows:

- Loop condition must be equatable at compile time.
- Conditions on break-statements, continue-statements and loops in the body must be equatable at compile time.

The need to be able to equate the loop condition and the conditions that might be involved in break- and continue-statements leads us to look at constant propagation and constant folding. The concept is already widely used in compilers to optimize the executed code and reduce operations to the ones that can only be equated at runtime. However, for the SystemC models that we are translating the extra restrictions we have placed on the input reduce the places where values can be changed. Actually, since we

only allow the module to be interfaced through its ports, anything that is not influenced by the ports in some manner is in principle calculable at the time of translation.

The principle we use for the propagation of constants works by maintaining a list of declarations made and tracking the values that they contain. As long as any assignment made to these declarations can be equated, the value can be known and subsequent uses of this declaration substitute its value if needed. This means that in many cases we can propagate literals to the conditions used in loops. When we find a loop that we can equate we can then unroll the loop and avoid the need to create a state machine to represent it. If all loops are equatable no timing problem is introduced and in this situation the mapping from SystemC to VHDL will maintain the same timing characteristics.

We do not track all declarations within the model. This would not be feasible because some declarations like signal declarations do not have the assignment semantics our approach assumes. They do not change their value upon assignment but rather do so after a delta delay. This is, however, not a problem since we will rarely see the situation where a signal is used internally while having no dependencies on the ports of a module. Since a dependency on one of the ports of the module makes the value unknown it will rarely matter that we cannot track signals.

3.4 Expressions

Expressions in C++ as in other languages define the operations to perform on data. This can be in the very literal operation of adding two values together or it can be the more abstract operation of referencing an element in an array or vector. We can classify expressions into three categories, namely operation expressions, accessor expressions and call expressions. We will discuss the mapping of each of the classes of operations in the following sections. Operation expressions are handled in Section 3.4.1, accessor expressions are handled in Section 3.4.2 and finally call expressions in Section 3.4.3.

3.4.1 Operation Expressions

Operation expressions are all expression that take operands and return some calculated result. These operations are for the most part trivial to map to VHDL since there are direct mappings of these operations for all the types. Even if the operation does not exist for the types we need we can add an overloaded operation to the VHDL translation to deal with this case. In essence this extends the standard VHDL with the operation semantics of SystemC. This allows us to keep the translation of the expression as close to the SystemC model as possible, seem as it is one of the goals of the translation we will use this approach to map most operations. All of the arithmetic, logical and comparison operators listed in Table 3.2 can be mapped in this fashion. Using the *numeric_std* for our translation defines almost all the operators we need with all the permutations of the types on these operators. We extend these standard VHDL operations with overloads to accomodate the extra operations supported in SystemC. These extra operations are comparison operations and logic operations between all permutations of the *integer*, *SIGNED* and *UNSIGNED* types. These operations are not all defined by default in *numeric_std*. Furthermore, comparison operators are added for the boolean and integer

compares allowed by C++. With these extra operators defined we have a complete set of binary operators that we can map too as well.

C++ Operator	VHDL Operator
<i>unary</i> + -	<i>unary</i> + -
+ -	+ -
*	*
\	\
%	mod
&	and
&&	and
	or
	or
^	xor
~	not
!	not
<	<
>	>
<=	<=
>=	>=
=	=
/=	/=

Table 3.2: Mapping of Operators

The alert reader will have noticed that there are some operations missing from Table 3.2. The operators in question are the incrementation operations. These have no equivalent in VHDL at all. Furthermore, their semantic behavior is such that we cannot replace them by simple additions or some form of function call. We need to introduce extra expressions before and after the expression being translated to accurately translate them. In C++ we have an incrementation and decrementation operation, both of which have a prefix and postfix form. It is in the distinction of prefix or postfix that the semantic incompatibilities occur. In C++ the prefix incrementation returns the incremented operand. Furthermore, it should return an l-value. In contrast, the postfix incrementation operator returns a copy of the operand as an r-value and then increments the operand. This entails that the value of the operation is the same as the operand. To illustrate, the code in Listing 3.19 shows the distinction. While the semantic difference is not always relevant and users generally only use the postfix form of the operations we do need to ensure semantic equivalence. To do this we introduce expressions before and after the primary expression in which the operations occur. The mapping shown in Listing 3.20 shows we can map prefix expressions so that it both increments the value and returns an l-value. The mapping in Listing 3.21 illustrates the mapping of postfix expressions. We do not ensure that the value returned from the postfix expression is an r-value and we do not need to since, assuming the C++ input is correct, we can replace an r-value for an l-value.

Splitting the expression in this manner raises the question of whether the order of evaluation is important. Since an expression could contain both a prefix incrementation and a separate use of the operand of this incrementation, for example $(++x) + x$. The ISO C++ standard states that the order of evaluation of operands and the order in which side effects take place is unspecified. In other words, the user cannot assume that in the previous example x will equal the incremented result of $++x$ or not. The result of the example is formally unspecified. The same would apply to the potential side effect of a function call. This entails that we do not need to enforce any particular ordering of the expression. However, GCC does maintain an informal evaluation order, namely that the expressions are ordered as if they were evaluated according to an in order traversal of the expression tree. So listing the expression in reverse Polish notation gives a list of the evaluation order. Using the same mechanism to perform the splitting of the tree will get us close to the same ordering that GCC uses. It does not, however, guarantee the same ordering, since we can form expressions with function calls which will not follow the GCC order of expression evaluation. An example is given in Section 3.4.3. We will leave this since a solution to would require significant extra analysis and it is not needed to comply with our goal to accept and translate ISO C++.

```

int i = 0;
int pre_inc;
int post_inc;

pre_inc = ++i; /* pre_inc = 1 & i = 1 */
pre_inc = ++i++; /* pre_inc = 2 & i = 3 */
post_inc = i++; /* post_inc = 3 & i = 4 */
post_inc = ++(i++); /* Invalid, (i++) not an l-value */

```

Listing 3.19: C++ Increment Expressions

```

i = i + 1;
pre_inc = i;

```

Listing 3.20: Pre-Increment Operator Mapping

```

post_inc = i;
i = i + 1;

```

Listing 3.21: Post-Increment Operator Mapping

3.4.2 Accessor Expressions

Expressions which are used to access parts of a type, such as an index on an array or record field accesses, can be expressed directly in VHDL for C++ types. If we look only

at C++ types then the only place we can use index expressions is on array types, since we do not support pointers. This maps directly to the index operator on a VHDL array type. The same is true for record field access as this maps to the same expression in VHDL. The SystemC data types support bit and range selects of appropriate types. We have chosen the mapping of the types so that these operations are also available in the VHDL mapping. Mapping the bit and range select operations of SystemC requires us to identify calls to specific functions in the SystemC library and mapping their arguments to a VHDL index or slice operation. We can then map the code in Listing 3.22 to its VHDL equivalent in Listing 3.23.

```
mysc_int.range(32,0) = mysc_int2;
mysc_int(32,0) = mysc_int2;
```

Listing 3.22: Index Operator in SystemC

```
mysc_int(32,0) := to_sc_signed(mysc_int2,32);
mysc_int(32,0) := to_sc_signed(mysc_int2,32);
```

Listing 3.23: Index Operator in VHDL

In the examples we add an extra function call to the translation. This call to *to_signed* is used to describe the semantics of an assignment in SystemC. In SystemC we are allowed to assign a vector type to a range that is not of the same size. The r-value of the assignment is then truncated or extended according to the type involved. The function *to_sc_signed* is one of the overloaded cast operators we introduced and will perform this truncating or extending as needed.

3.4.3 Call Expressions

The mapping of subprograms as is proposed at the end of Section 3.2 creates the need for special handling of call expressions. Since VHDL does not support subprograms that both contain reference parameters and return values we need to emulate the behavior. The proposed mapping works around this by returning the return value as an **out** parameter in the subprogram. This entails that expressions cannot use a call expression as an r-value, since all user defined subprograms do not have return values. To map call expressions we need to use a method similar to increment expressions. We perform the actual call expressions before the full expression which uses its result, storing the result in a temporary. Then we replace the call expression with the temporary so its value can be used. This maps the semantics of a return value to VHDL faithfully. Listing 3.24 shows an example expression with a call in it that is mapped to the code in Listing 3.25. Just as was the case with the prefix incrementation operator we perform this splitting during an in order traversal of the expression tree. As we mentioned in Section 3.4.1 this does not guarantee the same semantics as GCC. If in Listing 3.24 the argument *x* that is passed to the function *fun* is a reference parameter that is changed in the function then

the VHDL mapping would use this value for x during the evaluation of the expression while the C++ expression compiled under GCC would use the previous value of x .

A final expression that requires special handling is the conditional expression as in Listing 3.26. There is no equivalent conditional expression in VHDL and we map these expressions using the same method as call expressions and incrementation expressions as listed in Listing 3.27.

```
x = x + fun(x);
```

Listing 3.24: Call Expression

```
fun(temp1, x);  
x := x + temp1;
```

Listing 3.25: VHDL Call Expression

```
x = ( i==0? a: b );
```

Listing 3.26: Conditional Expression

3.5 Model Structure

A SystemC model contains both code to be executed in processes as well as structure information that binds signals to each other, instantiates components and defines sensitivity information for the processes. All this information about the structure of the model also needs to be included in the VHDL translation of the model. The structure information in a SystemC model is specified during a specific phase of execution of the simulation. This phase, called the elaboration phase, also exists in VHDL simulators and is used to set up the internal structure for the simulation. In SystemC information about the structure is specified by calling specific methods defined by SystemC. Using these methods the user specifies the binding information for signals and ports, instantiates components binding the components ports, and specifies which methods in the model class are processes and what signals they are sensitive to.

There are two main methods to extract the information created during the elaboration phase of the model. The first is to statically analyze the program recording the information as needed. The second method is to actually execute the elaboration phase and then read the information about structure directly from the SystemC kernel. The first method is appealing because it uses the published information about SystemC to extract the information. It does not attempt to read internal information out of the SystemC kernel, which is undocumented. The second is appealing because it allows the model to be as complex as needed during the elaboration phase, for example settings could be read from a file. The dynamic approach is taken by the creators of Pinapa[21].

```
if ( i==0) then
    temp1 := a;
else
    temp1 := b;
end if;
x := temp1;
```

Listing 3.27: VHDL Conditional Expression

In that system they modify the source of an implementation of SystemC so that it can be used to read the structure information out of the SystemC kernel. We feel that the dynamic approach while more powerful is restrictive because it confines the user to using a particular implementation of the SystemC standard. While in principle all implementations of a standard should respond the same way this is rarely true in practice. Forcing the user to use the same version that the translator tool uses. Furthermore, it is dangerous to extract information out of the kernel since it is completely undocumented and using the information correctly becomes a guessing game. We prefer the static analysis approach and will use that for the tool. The writers of the Karlsruhe SystemC Parser Suite [7] used a similar approach with good results. While it might restrict the user to specific constructs it also enforces good coding and that is not a bad side effect. In the following sections we will discuss the extraction of the structure data from the constructor of a module. As stated in Section 2.2.4 we make the assumption that the structure of a module is fully specified in its default SystemC constructor and this is where we will look for it.

3.5.1 Bindings

Bindings refer to the association of signals and ports with each other. SystemC allows for signals and ports to be explicitly bound to each other so that operations on the ports are passed on to the signal. Binding in SystemC is asymmetrical such that a port bound to a signal does not imply the reverse, namely that the signal is bound to the port. In other words the binding is one way. The bindings are generally used to connect different parts of a hierarchical model together into a coherent whole, with the ports of components being bound to signals. This can be directly represented in VHDL using the component port map. The binding mechanism can also be used to bind a local port to a local signal and then to use the local port instead of the signal. While this is valid according to the SystemC language reference manual it is a rather strange construction to use. It seems to serve no purpose at all since why would the local signal need to exist at all in this case.

While it is possible to bind a port to multiple signals this is unsynthesizable, since it models a 'pin' being connected to multiple distinct nets without any form of resolution logic. Since connecting two nets to each other makes them a single net this is simply a contradicting model. We therefore do not support this and will output an error when a call is made to one of the methods that facilitate it, namely the *operator*[] overload of *sc_port* class. The same behavior can be modeled using multiple explicit ports and this

can be readily modeled and synthesized.

```
port1.bind(sig1); /* Explicit Function Call */
port2(sig1); /* Overloaded operator Binding */
```

Listing 3.28: SystemC Binding

3.5.2 Components

In hardware models we often distinguish between behavioral models and structural models. The difference between the two is that the first describes the behavior of the model using processes while the second describes the model at a lower level using components and the structure of the model. While models can be purely of one type or the other they are generally a mixture of the two. The declaration of components, submodules, is a key part of a structural model. It allows us to define the behavior of a module based on the behavior of submodules. We can define the interconnections of these modules and define processes to work on the signals generated by the submodules. In SystemC components are declared by instantiating a class that derives from *sc_module*. In the constructor we then bind the ports of the component to internal signals which we can then work with. This component instantiation and binding can be mapped to the VHDL component structure using a component instantiation in VHDL and the bindings can be mapped to a port map in this instantiation. So the call to bind the ports of a component in Listing 3.29 can be mapped to the component instantiation in Listing 3.30. There are three methods to specify a binding in a component, the first is to use the explicit function *bind* to perform positional binding. The second is to use the overloaded **operator** () to perform a positional binding. The last is to reference the component's ports directly and bind them. We can collect all the bindings for a component during the processing of the constructor and then output a single component instantiation in VHDL, so that all the methods may be used.

```
module1 comp1;

SC_CTOR(MyModule)
{
    comp1.bind(sig1, sig2);
}
```

Listing 3.29: SystemC Component Binding

```
comp1 : module1 port map(sig1, sig2);
```

Listing 3.30: VHDL Component Instantiation

3.5.3 Processes

In SystemC a process is just a method of the module class and the user needs to inform the SystemC kernel which methods are processes. This is also done in the constructor of the module. We specify which methods are processes and we also specify the sensitivity that the processes have towards signals. We can use the calls that register processes to find out what methods are processes and output the respective VHDL for the processes. The sensitivity is likewise discoverable in the constructor. We need to check for specific function calls and analyze the arguments. A problem does occur in that SystemC supports three types of sensitivity. It supports sensitivity that occurs when a signal changes, but also supports more restricted sensitivity that only occur when a signal changes on a positive edge or on a negative edge, basically leading or trailing edge sensitivity. This leading or trailing edge sensitivity can be expressed in two manners in SystemC. The first makes use of the special members of *sc_module* called *sensitive_pos* and *sensitive_neg*. The second uses a member in *sc_signal* to return an event that only triggers on a positive or negative edge. While we cannot directly support this sensitivity in the VHDL sensitivity list, we can embed the entire body of the process in an if-statement checking for the positive and negative edges of the signals as needed. This leads to a mapping as shown in Listings 3.31 and 3.32. The detection of positive and negative edges is performed using runtime methods to make the generated code more concise.

```

SC_CTOR(MyModule)
{
    SC_METHOD(MyProcess);
    sensitive << sig1;
    sensitive_pos << sig2;
    sensitive << sig3.neg_event();
}

MyProcess()
{
    <Statements>
}

```

Listing 3.31: SystemC Process

```

MyProcess : process (sig1, sig2) is
begin
    if(sig1.event || pos_event(sig1) || neg_event(sig3)) then
        <Statements>
    end if;
end MyProcess;

```

Listing 3.32: VHDL Process

3.6 Conclusion

A SystemC model written within the restrictions we have described in Chapter 2 can be mapped to VHDL with our mappings. For some specific constructs extra expressions need to be introduced to simulate C++ semantics and a considerable amount of transformation needs to occur to map runtime bounded loops to a finite state machine. The implicit conversions in assignments defined by SystemC require us to pass the r-value of an assignment through a conversion function to make these conversions explicit. While this adds a lot of code to the VHDL mapping we must realize that all of these cast functions only serve to instruct the VHDL compiler what is intended with the data on a bus and will not add any logic during synthesis of the VHDL model.

The behavior is maintained throughout the mappings and the resulting translation will in principle always return a behaviorally valid translation even if the timing of the model is suspect. The suspect timing occurs when a runtime bounded loop is translated into a finite state machine. Therefore, the use of a runtime bounded loop must always be done with the final translation of the loop in mind. While this reduces the use of such loops, the loops remain a powerful tool to describe complex constructs very compactly.

This chapter deals with the actual implementation of the translation discussed in the previous chapters. In particular we will give a brief background on parsing techniques in Section 4.1 and explain the reasons for using GCC as a front-end for the tool. In Section 4.2 we give a short description of the output generated by the GCC front-end, this being the point at which the translation to VHDL starts. The tool uses an intermediate abstract syntax tree (AST) to represent the VHDL translation which is presented in Section 4.3. The method we used to generate the VHDL abstract syntax tree from the GCC abstract syntax tree is discussed in Section 4.4. In that section we will also lay out the algorithms used to generate a finite state machine from a subprogram body. Section 4.5 describes the code generation stage of the tool and we conclude in Section 4.6.

4.1 Parsing C++

C++ was originally developed by Bjarne Stroustrup at Bell Labs. It was designed as a version of C that supported new programming paradigms like object-oriented programming. Many more features were added over time and in 1998 it was formalized into a standard language in the ISO/IEC 14882:1998 standard[17] and again in ISO/IEC 14882:2003[18]. The roots of the language are relevant for parsing because many of the difficulties in parsing C++ are rooted in how it developed. The language was not created as a consistent whole but rather a series of extensions to C. Because the original syntax of C was used as the starting point for the syntax of C++, many of the grammatical difficulties of C were transferred into C++. Thus, due to the complex extensions of classes and templates, these grammatical difficulties became more challenging to deal with. A classic example of C++ parsing difficulties is the declaration expression ambiguity.

```
f (x);
```

Intuitively, the above line appears to be a function call to the function *f* with the parameter *x*. However, if the following *typedef* is in the current scope then it is a declaration of a variable *x* of type *int* with redundant parentheses.

```
typedef int f;
```

The token stream can only be resolved to an expression or a declaration by performing name lookup, which is not the job of syntax analysis. This ambiguity is directly inherited from C but there are many more in C++, as the Roskind Grammar[24] illustrated.

It is common practice to create parsers for languages using a parser generating tool. Such a tool generates the parser based on a formal grammar of the language to be parsed. One of the defining characteristics for a grammar is the amount of tokens that are required to identify constructs in the source. For example, the popular parser

generator Bison[1] is based upon an LALR(1) grammar. This grammar uses one token of lookahead, as it is called, to decide what action to take. So that different language constructs must be discernible with only one token of lookahead. Many languages can be expressed in such a grammar and tools such as Bison are extremely useful for generating parsers for such languages. C++ is however not directly expressible in such a grammar because of all the ambiguities in the syntax. While we can create a grammar that will parse C++, we need to perform significant amount of post processing to identify some of the more problematic constructs. The post processing required can become so complex that it may rival the effort required to create the parser manually[27][13][8].

There are other grammar forms that would allow C++ to be expressed fully. One such grammar is a GLR grammar. These grammars are a generalized form of LR grammar which allow ambiguities to exist and be resolved at a later point. This style of grammar can analyze the source and generate an initial abstract syntax tree for the source. We are however still left with a great deal of work in semantically analyzing the input. For example, we need to correctly link identifiers to their use. While we might consider performing all this analysis in our tool, it seems a waste of effort because so much of this has already been done in all C++ compilers that exist.

A highly popular and accepted compiler is the GNU GCC compiler. It is the defacto standard compiler used on Linux and is based on the Unix C compiler, even though that historical connection is rather faint now. We can, with some modifications to the GCC compiler, use it as a front-end for our tool as was done in Pinapa[21]. Using GCC as the front-end has various advantages the greatest of which is that we can let the existing front-end of GCC handle all of the input processing. By letting it deal with all the syntactic and semantic quirks of C++, we can concentrate on the translation of the abstract syntax tree. However, there is a downside to this approach, since with GCC as a front-end we cannot maintain the comments in the source throughout the translation. At least we cannot do so easily, as we would need to perform this separately re-parsing the input manually looking for comments and mapping them to source lines. Despite the fact that we cannot maintain the source comments in the translation we will use GCC as a front-end for our tool. If only for the reason that it will bring us closest to a front end that correctly parses ISO C++.

4.2 GCC AST

Once we let GCC parse the input we are left with an abstract syntax tree representing the input. This tree is created by GCC and would normally be passed to the optimization and code generation modules of GCC to output the final code. We instead use the tree to generate a new abstract syntax tree that represents the input in a form closer to VHDL. We take this step so that we can make a clear distinction between parsing, translation and code generation. We want this clear distinction for possible future extension of the tool and because it is conceptually appealing to design in this manner.

The syntax tree is described in the GCC Internals Manual[2]. The basic concept is that each structure starts with a member that contains some basic information about a node in the tree, this initial member is again a structure called *tree_common*. It contains a identifier that allows users to identify the particular type of node that it is and then use

it as such. All the information is accessible using macros that are defined, but it is up to the user to ensure that these macros are only used on nodes that are of the correct type. This mechanism allows users to treat each node as the same type when they are not. It is a primitive form of sub classing and polymorphism implemented using C structures. It allows GCC to create very complex trees while not having to continually deal with many different types of nodes. Every node is just a pointer to a *tree_common* structure with the node type set to the type that the node should represent.

4.3 VHDL AST

The VHDL translation is represented internally as a second abstract syntax tree. This approach has the advantage of logically splitting the three main steps of the translation from each other. Namely the parsing, which is handled by GCC, the translation proper and finally the code generation. It means that the translation routines do not need to concern themselves with the syntax of the output and we can easily re-target the tool for other languages or possibly for future syntax changes in VHDL as proposed in the VHDL-200X proposal[5]. For example, creating a back-end to output a SystemC model should be trivial and would be useful for testing purposes.

Using the GCC internal tree representation as a starting point we decided to use a similar mechanism to represent the VHDL tree. We use the same pseudo subclassing method by having all tree structures start with the same common structure so that we can know at runtime what type of tree node we are dealing with. This has various advantages over a static structure tree because it allows runtime checks on the nodes to determine if a node is of the type it should be and not an expression node where only a declaration node is valid. With static structures we would never be able to determine the type of structure a pointer points to. This is why C++ introduced runtime type information; we merely emulate this. The VHDL abstract syntax tree contains six node structures as listed in Table 4.1. There are two extra node structures that are used during translation but they are never part of the resulting tree named *vh_expr_retrun* and *vh_process_list*. While there are only six actual types of structures there are many more nodes but they all share the same types. For example, all expressions use the expression structure, the actual code of the node defines whether it is an addition or subtraction. Allocation of the nodes in memory is always performed using one function called *vh_build*. This function tracks all nodes that are built so that the entire tree can be deallocated in one go. No part of the tree can be deallocated separately, because the tree is complex and we would need to do a full tree search to invalidate all the pointers to a node before deallocating it. A complete listing of all the tree codes and the macros used to access the tree data is available in Appendix A.

4.4 GCC AST to VHDL AST

In this section we will describe the path the tool takes through the GCC AST in order to generate the VHDL AST. We will assume that we are only concerned with a single module. Since adding support for multiple modules is a simple matter of running the

Structures
<i>vh_type</i>
<i>vh_expression</i>
<i>vh_statement</i>
<i>vh_declaration</i>
<i>vh_identifier</i>
<i>vh_integer_cst</i>

Table 4.1: VHDL AST Structures

same algorithm once for each module. The translation starts by finding the abstract syntax tree node that represents the class of the module we are interested in. We perform a name look-up to find this tree for now, but we could also look for all modules declared in the input and translate each in turn. Once we have the top node for the class we can start to translate the module.

We start the mapping by first dealing with the member fields of the module. We deal with each in turn and map them to a field in VHDL, mapping the types as needed. During the mapping of the fields we create a table of the connection between a GCC tree node for a field and the new VHDL tree node for that field. This will allow us to quickly find the correct node to map references to this field too. The mapping of types also maintains a list of the types so that user defined types can be tracked and a single declaration for them can be generated. Type references are resolved in the same manner as field references, we look them up in a pointer map that maps a GCC node to a VHDL node.

The central portions of the translation are explained in the following sections. We will describe the translation of the structural data from the constructor of the module in Section 4.4.1. In Section 4.4.2 we describe the translation of a process body, the mapping of the body to a finite state machine and the translation of expressions to VHDL expressions.

4.4.1 Binding Constructor

Once we have dealt with the fields of a module we can truly get started mapping the behavior and structure of the module. This all starts from the binding constructor as we mentioned in Chapter 3. We look at all the constructors of a module trying to find one that contains only one argument namely one that is of type *sc_module_name*. The first of these that we find is assumed to be the binding constructor where we can find all the structure information of the module. We then analyze each statement in the function and attempt to identify those statements that define structure. Listings 4.1 and 4.2 show a sample mapping of the structure defined in SystemC and the resulting entity/architecture pair in VHDL.

For process identification we need to find syntax tree nodes that represent initialization expressions. If the initialization expression is of type *sc_method_handle* then we have a potential method registration. Next, we confirm that the initialization expression is a function call to the *sc_method* registration method. In the arguments of this call we

```

SCMODULE(MyMod)
{
  sc_in<sc_logic> port1;
  sc_signal<sc_logic> sig1;
  MyComponent mycomp;
  SC_CTOR(MyMod) : mycomp("mycomp")
  {
    SC_METHOD(MyProcess);
    sensitive << sig1;

    mycomp.bind(sig1);
  }
  MyProcess()
  {
    port1 = sig1;
  }
}

```

Listing 4.1: SystemC Structure

```

entity MyMod
  ports(port1 : in std_logic);
end entity MyMod;

architecture SystemC of MyMod
signal sig1 : std_logic;
component MyComponent
  ports(port1 out : std_logic);
end component MyComponent;
begin
  MyProcess : process(sig1)
begin
    port1 <= sig1;
end process MyProcess;
end architecture SystemC;

```

Listing 4.2: VHDL Structure

can identify the method that is referred to and then translate this method as a process. We do not do this immediately. We create a VHDL AST node to represent the process but we delay translating the body until we have finished with the binding function. This is necessary since we need to collect the sensitivity information before translating the body of the process.

Identifying the sensitivity of a process is done by identifying call expressions that call

the overloaded *operator <<* on the members *sensitive*, *sensitive_pos* and *sensitive_neg*. Identifying these calls is a matter of finding the abstract syntax tree nodes that represent the overloaded operators and checking whether the call expression calls one of these. We can then analyze the arguments of the call expression to find the signals that the process is sensitive to. We add this information to the process the was last translated since this is consistent with the semantics of SystemC where the process sensitivity refers too is the last process to be defined.

All the information we need to gather from the binding constructor is defined using function calls in SystemC. These function calls range from constructors that set initial values of SystemC members to functions that bind ports to signals. We introduce a general mechanism to identify function calls that can be easily extended to support extra calls in the future. This call identification mechanism is also extensively used while translating expressions. Using this call mechanism we no longer need to worry about the myriad of functions that are defined throughout the SystemC language. This means that the binding function declared on *sc_in* and the one defined on *sc_port* can be dealt with in the same manner, because both have the exact same semantic meaning. The semantics being: bind the argument to the port the function is called on, so that *port1.bind(sig2)* binds *port1* to *sig2*. All bind functions, be they based on a member function or based on an operator overload, can be grouped into two groups. The first are the single argument binds, like the bind on signals, or multiple argument binds, such as the positional binding for modules. GCC has already translated member function calls into global calls with a *this* pointer so we can easily identify the target of the call and the arguments can be translated in the same manner.

Using the identified objects that need to be bound we can then add their VHDL representations to the VHDL AST. A binding between an internal signal and a port on a component is mapped to a binding in the component's binding clause. The component will already have been translated since it will have been found in the fields of the module. Bindings between the ports of two components need to introduce a new signal with that signal being bound to the two components, since we cannot bind two component ports together in VHDL. The last binding that may occur in SystemC is a binding between a local port and local signal. This binding can also not be represented in VHDL but it is also a useless binding within our restrictions of SystemC. It only serves to introduce an alias for the port. We can therefore replace references to the signal with the port to perform this binding. We make use of the pointer map to create such an alias.

Since we use GCC to parse the source model the abstract syntax tree of the binding function will contain many parts that are of no interest to us. For example, it will contain a call to the super constructor in *sc_module* and will contain a constructor call for all of the SystemC fields and components in the module. We need to identify these only to be able to ignore them, so that we can still signal errors correctly. We therefore only warn if we find a constructor call in the binding function that we do not recognize and we output an error if we find a function call that we do not recognize. We do not support functions or methods being called in the binding function other than the methods that directly relate to the elaboration as described above.

4.4.2 Processes and Functions

A process in SystemC must be a method in the module. It has no parameters and returns *void*. We therefore only need to concern ourselves with the body of the process. A method body is represented in the GCC AST as a list of statements and expressions. We translate the body by dealing with each statement or expression in turn while maintaining a current translation state. This translation state contains information such as the level of the tree the statement is found in or the current FSM state that the statement must be placed in.

Ignoring expressions for now we can lay out the algorithms used to translate statements into either a single list of VHDL statements or a synthesized state machine to represent the body. While generating the state machine it is important to realize that only the translation of loops can introduce the need to generate a state machine. If there is no loop in the body of the process then there is also no need to generate the state machine. Using a post-order traversal of the GCC AST as a starting point we would translate the statements by a series of functions, each dealing with a single type of statement. Each of these functions then returns the VHDL statement corresponding to the argument. Both a process and a function are translated using the same algorithms. However, if a function is being translated then the tool will not allow any statements that require a finite state machine to be created.

The bottom-up approach works as long as each statement can be represented in a VHDL statement. This, as we know, is not the case, since loops need to be synthesized into finite state machines. We can use the bottom-up approach as a starting place for the algorithms, however, and add the logic needed to deal with state to each function. For if the translation of a statement introduces FSM states then it still needs to return some kind of statement to represent the switch to this FSM state. This is the approach used in the algorithms for mapping process bodies to VHDL. The main types of statements that we need to translate are if-, for-, while-, do- and switch-statements.

The entire translation of statements is tied together in a general function that will accept any statement and map it. This function decides the appropriate action given the statement type. When a list of statements is found then these are translated in sequence. Whenever the translation of a statement creates a state then we remember that state and all statements following will be placed in that state instead of being returned as part of the translated list. Because only the next statement enters the list we are ensured that any non-empty C++ statement list will translate to a VHDL list with at least one statement. This ensures that we never have empty bodies if they were not empty to start with. This is important because empty bodies are not allowed in VHDL, they must be explicitly empty with the null-statement. For bodies that are empty in the original model we introduce null-statements so that the translation is valid, a warning is generated since it just complicates the output for no need.

The following sections describe the particulars of the translation of the control statements. Section 4.4.3 presents an algorithm to translate all the loop statements. The translation of if-statements is described in Section 4.4.4 and switch-statements are translated as described in Section 4.4.5. The translation of expressions is described in Section 4.4.6 along with the translation of built-in SystemC methods.

4.4.3 Loops

We will start with the loop statements and show a general algorithm for translating them. All the loops are dealt with in almost the same manner, the differences are only in where we add conditional state changes in the body for iteration and that we need to add extra expressions to the FSM when dealing with for-loops. The general algorithm for translating a loop is given in Listing 4.3. The algorithm assumes that we already know that this loop must be translated to an FSM and that if that were not true, then the translation would be done using a different algorithm. We start by immediately creating two FSM states, namely the state for the body of the loop and the state for the statements trailing the loop, called body state and tail state. We translate the body of the loop using the function `map` and append the result to the body state. We need to add a conditional state change to the end of the body. This state change decides whether to iterate or exit the loop. In the simple case the body of the loop does not contain an embedded loop so that when `map` returns the FSM has no new states, we can then add the state change to the end of the body state. However, the loop body might contain an embedded loop, in that case the FSM does have new states and part of the loop body will have be placed in these new states, see Section 3.3. This entails that we need to add the state change to this new state and not to the body state. Finally, we return a conditional state change to the body state or tail state. The caller of this function will place this state change in place of the original loop.

4.4.4 If Statements

Translating if-statements, when the bodies of the then- and else-clauses do not contain loops is trivial. It is just a matter of translating the condition, the then- and the else-clause and returning a VHDL AST node representing an if-statement. When either the then-clause or the else-clause contains a loop it becomes more complicated. We need to check for the introduction of states when a body is translated and if a state was introduced then we append a state change to a new tail state. This is illustrated in Listing 4.4. If either the else-body or the then-body introduces a state this state change needs to be introduced at the end of both bodies, since all the trailing statements will now be collected in the introduced tail state.

4.4.5 Switch Statements

The last control statement that we need to translate is the switch-statement. A switch-statement is actually just a general form of if-statement and we follow the same method while translating it. If during the translation of one of the cases a new state is introduced then we add a new tail state and all cases switch to this at the end of their bodies. It is clear that if the state is introduced on any but the first case we will need to back-track and modify all the previously translated cases to add the state change. As we already mentioned in Section 3.3 special handling of the case-statements is required to deal with the differences between C++ switch-statements and VHDL case-statements.

```

map_loop_stmt()
{
    bodystate = new_state();
    tailstate = new_state();

    append(<bodystate>, map(<loopbody>));

    if(<map introduced state>)
    {
        append(<map introduced state>, <loop iterator>);
        append(<map introduced state>,
            <conditional state change to body or tail>);
    }
    else
    {
        append(<bodystate>, <loop iterator>);
        append(<bodystate>,
            <conditional state change to body or tail>);
    }

    return <conditional state change to body or tail>;
}

```

Listing 4.3: Algorithm to map loop statements

4.4.6 Expressions

Expressions are mapped directly as listed in Section 3.4 with only the call expressions, incrementation expressions and conditional expressions getting special consideration. The special processing required for these expressions is also the reason for the translation context needing to remember where an expression occurs, since we cannot expand an expression that is used as a condition to multiple expressions. For example, an if-statement with as its condition $i++$ cannot be expanded to an if-statement with two expressions as its condition. We need to handle these cases in a special fashion. We do this by performing the expressions that must occur before the main expression before the statement that uses them and performing expressions after the main expression in all the bodies of the statement. So that Listing 4.5 maps to 4.6. Call expressions are dealt with in the same fashion with the actual call being moved to the surrounding statement and its original place being substituted with a temporary variable that hold the return value.

The example that is described in these listings also illustrates another point. Since we use GCC as the front-end we can sometimes end up with code that we did not expect. These listings are an example of this, since the code generated by the final tool will not be what is listed in Listing 4.6 but rather what is in Listing 4.7. The reason for this is that GCC has already recognized that with a simple substitution the code can be made more

```

map_if_stmt()
{
  append(<if stmt>, map(condition));
  append(<if stmt>, map(thenbody));
  <then state> = <map introduced state>
  append(<if stmt>, map(elsebody));
  <else state> = <map introduced state>

  if(<then state> || <else state>)
  {
    tailstate = new_state();
    if(<then state>)
      append(<then state>, <state change to tailstate>);
    else
      append(<thenbody>, <state change to tailstate>);

    if(<else state>)
      append(<else state>, <state change to tailstate>);
    else
      append(<elsebody>, <state change to tailstate>);
  }

  return <if stmt>
}

```

Listing 4.4: Algorithm to map if-statements

efficient, namely the post-incrementation is replaced with a pre-incrementation and the comparison is performed on 1 instead of 0. This optimization actually improves the code that is output by the tool since we do not need to duplicate the expression to increment i .

```

if( i++ && ++j)
  <statements1>
else
  <statements2>

```

Listing 4.5: Increments as Conditions

So far when discussing expressions we have not considered what to do with the methods of the SystemC classes. These methods pertain to various actions that can be performed on the classes. For example, there are writing and reading operations of signals, ports and types. We will use the general function call identification mechanism to identify such function calls, grouping them logically as needed. We can then replace the function calls with semantically equivalent operators or, if no such operator exists,

```

j := j + 1;
if (i /= 0 and j /= 0)
  i := i + 1;
  <statements1>
else
  i := i + 1;
  <statements2>
end if;

```

Listing 4.6: Increments as Conditions: Mapping

```

j := (j + 1);
i := (i + 1);
if (((i /= 1) and (j /= 0))) then
  <statements1>
else
  <statements2>
end if;

```

Listing 4.7: Increments as Conditions: Tool Result

we can create a call to a runtime function. The functions that we need to define in a runtime library are shown in Table 4.2. These will be defined in a separate library that is used by the translated model. These functions will be implemented to mimic their SystemC counterpart. Next to these functions we also define runtime functions that can cast the types according to the cast semantics defined by SystemC. While VHDL implements some casts by default, not all are available so we extend the casting with extra functions. We add a cast for each type that can be assigned to another type. For example, a cast from a logic vector to an integer since assignment from the one to the other is available in SystemC.

4.5 Code Generation

Code generation in the tool is trivial. The abstract syntax tree for the VHDL translation is tailored to hardware modeling languages and in particular to VHDL. It is therefore a simple matter of representing the syntax tree with text. We essentially only need to serialize the tree structure. We traverse the tree starting at the list of top-level declarations and output each declaration in turn. In order to ensure that we generate readable VHDL code we increase or decrease an indent level when needed. The source printing routines handle the need to insert indent characters after each newline. During the translation between C++ and VHDL the VHDL syntax tree records the C++ source lines that are the source of the new syntax tree. This information can be output to the translation in the form of comments. The translation can also introduce comments into the syntax tree to explain where certain constructs are come from. These can also be

SystemC Method	VHDL Method
<i>length</i>	<i>length</i>
<i>lrotate</i>	<i>lrotate</i>
<i>rrotate</i>	<i>rrotate</i>
<i>reverse</i>	<i>reverse</i>
<i>and_reduce</i>	<i>and_reduce</i>
<i>nand_reduce</i>	<i>nand_reduce</i>
<i>or_reduce</i>	<i>or_reduce</i>
<i>nor_reduce</i>	<i>nor_reduce</i>
<i>xor_reduce</i>	<i>xor_reduce</i>
<i>xnor_reduce</i>	<i>xnor_reduce</i>
<i>is_01</i>	<i>is_01</i>

Table 4.2: Mapping of SystemC Methods

output into the VHDL source. As we noted in Section 4.1, comments that were in the original SystemC source are not available in the VHDL output.

4.6 Conclusion

The tool translates the model starting at the binding function. This allows the tool to only translate that which is directly relevant for the module in question. After all the fields of the module are translated we start collecting the structural information from the binding constructor. As we find processes we record them for later processing. When all structure information has been translated we translate each process in turn. Since the tool only translates that which is used directly by the model in question, we cannot use functions that are only declared. We require that everything be fully defined in the source so that it can all be translated. There is no ability to use linking semantics in the context of the tool.

As GCC is used as the front-end the tool accepts any model that is also accepted by GCC. We can give clear errors when translating the module if any invalid constructs are found because we have parsed the entire model and have not limited ourselves to a parser for only the synthesizable subset. This also improves the ability to extend the tool since the parser already accepts anything that can be expressed in SystemC.

The tool described in the previous chapter has been implemented and in order to provide some measure of validation for the tool we tested it with a series of test models. In this chapter we will describe these models and present simulation results allowing a comparison of the pre- and post-translation models. In Section 5.1 we will introduce the models used to test the tool and then we follow in Section 5.2 by presenting the simulation results. Finally, comparing the simulation results for both the original and the translated models give us a place to draw some conclusions about the validity of the translation in Section 5.3.

5.1 Test Models

To verify the created tool we translated five SystemC models as tests. The original SystemC models and the resulting VHDL models were simulated and the results of the simulation compared. The first four of these are models from the Open Cores initiative[3]. The models were written as synthesizable SystemC models meaning that they already conform to the described input restrictions. The models used are a model of a random number generator (RNG), an MD5 hash implementation, a DES encryption implementation and a USB implementation. All of these are useful hardware elements and represent the real world use of modeling languages. The models represent a range of modeling methods and complexity, with the random number generator being the simplest model with only a single module of 176 lines and the USB model being the most complex with 24 modules spanning 5,792 lines. The modeling styles range from the simple RTL model of the random number generator to complex structural and finite state machine models of the USB and DES encryption. The last model we used is an implementation of the CORDIC algorithm. The model was specifically written to be a very compact and simple model of a single CORDIC mode and is meant as a proof of concept for the finite state machine mechanisms. Some characteristics of each model are listed in Table 5.1. In the following sections a short description is given of each model, focusing on the rationale for using them as tests for the tool.

Modle	Type	Modules	Size (Lines of Code)
Random Number Generator	RTL(Basic)	1	176
MD5 Hashing	RTL(FSM)	1	478
DES Encryption	RTL(Structural/FSM)	11	2,393
USB	RTL(Structural/FSM)	24	5,702
CORDIC	Behavioral	1	169

Table 5.1: Test Models

5.1.1 Random Number Generator

The first model we used is a random number generator. It is based on a combination of a linear feedback shift register (LFSR) and a cellular automata shift register (CASR) as presented in [20]. It provides good statistical results with a small footprint. The model contains three processes: one for the LFSR, one for the CASR and finally a process that combines the results.

The random number generator is our first test as it is a very simple model. It makes no use of the more intricate features of the tool and has a straightforward mapping to VHDL.

5.1.2 MD5

The next model we used to test is a more complex model of the MD5 hashing algorithm. While now considered out of date, the MD5 hash was and still is a widely used hashing algorithm. Since the algorithm has proven to be vulnerable to collisions and such collisions can be easily generated, its use in security environments is inadvisable. It is, however, still a very viable algorithm to check for random data corruption.

The MD5 model is a more complicated model that is still contained within a single module. It serves as a test for a model that implements a state machine.

5.1.3 DES

The Data Encryption Standard (DES) was for a long time the standard encryption used in the security industry. It was the encryption algorithm approved by the National Security Agency of the United States for use in government applications. While it has since been shown that DES is not secure and the NSA approval has been withdrawn, a variant of DES called triple DES (3DES) is still widely used. The only difference with triple DES is that triple DES performs the DES algorithm three times with different keys. Since performing DES three times with different keys is a simple extension to the model we use, our model still has reasonable significance in current security environments.

The model of the DES algorithm is one of the more complicated models we tested. We selected it primarily because it is composed of various modules that are instantiated. This serves to test the tool's ability to derive VHDL structural information from the SystemC model.

5.1.4 USB

The Universal Serial Bus (USB) is a very widely used interface for peripheral hardware in general computing systems. The model of the bus conforms to the USB 1.1 standard. We selected this model because it represents such an obvious real world hardware element and further because of the complexity of the model. The model consists of various modules and makes use of some of the more complex parts of C++ such as conditional expressions and implicit object instantiation. These portions of C++ are not covered by the other tests.

5.1.5 CORDIC

Function evaluation in hardware has always been a significant area of study since there are large differences in area and speed between various possible implementations. The CORDIC algorithm is a generalized function evaluator for trigonometric functions. It can provide the evaluation of a large amount of trigonometric functions in a single implementation with a relatively small area. The implementation of the CORDIC algorithm we use was written for demonstration purposes and as such is not a complete implementation. It only supports a limited amount of CORDIC modes, but it serves as a proof of concept for the translation of a model using the finite state machine mechanism of the tool.

In order to test the finite state machine generation we forced the loop contained in this model to be unanalyzable. This was done by hiding the loop bound in a function call. Forcing the generation of a finite state machine is an efficient way to ensure that a model does not get expanded fully. Having the loop fully expanded would result in an explosive growth of the area required for the model whereas we wanted to create an area efficient function evaluator.

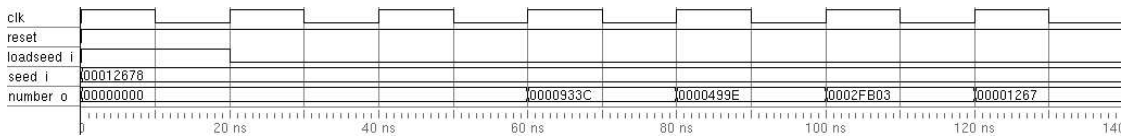


Figure 5.1: VHDL Random Number Generator Model Timing

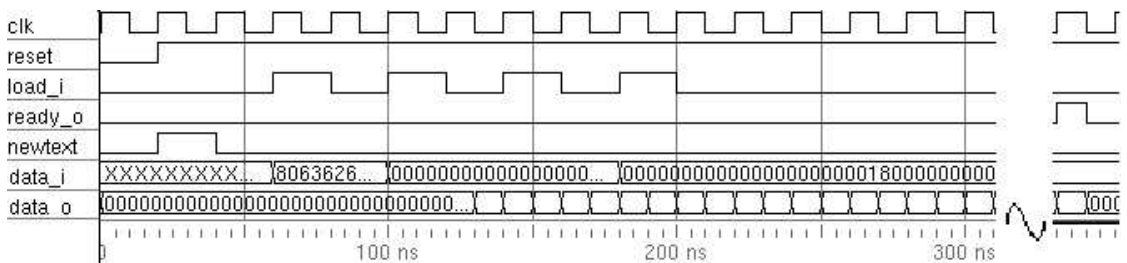


Figure 5.2: VHDL MD5 Model Timing

5.2 Test Results

All of the test models were translated using the tool and the resulting VHDL models were simulated. The waveforms that result from these tests are then compared with waveforms generated by simulating the original models. In the comparison we compare the final results as well as the intermediate signal changes confirming that they occur at the correct times.

The random number generator, the MD5 and DES algorithms generate identical waveforms from the original and translated models given the same stimulus. The produced waveforms can be seen in Figures 5.1, 5.2, 5.3 and 5.4. Only one for each of the

tests is shown since both the pre- and post-translation tests produce an identical wave. The waveforms are representative of the complete tests for the models, since depicting the complete wave for DES and USB is impractical. These results maintain our assertion that when translating a model that does not contain any runtime bounded loops the translation made is exact. The only difference that could be found in the models is that in SystemC a type has an implicit initial value, whereas in VHDL this is not the case. This is especially true of the SystemC integer types that have an initial value of '0' while in VHDL they are a vector of 'X' values. In the test models this difference did not cause any difference in behavior and relying on these implicit initial values is bad practice. However, we might consider introducing initial values for all declarations when none is given in the original model.

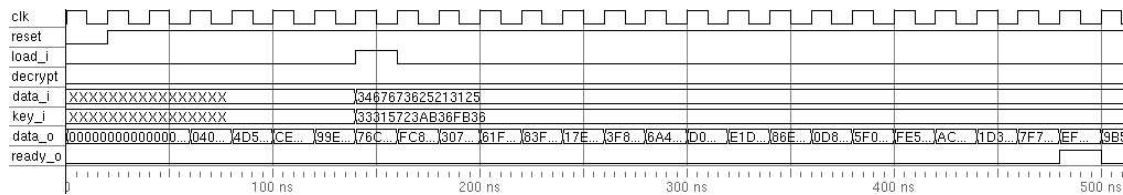


Figure 5.3: VHDL DES Model Timing

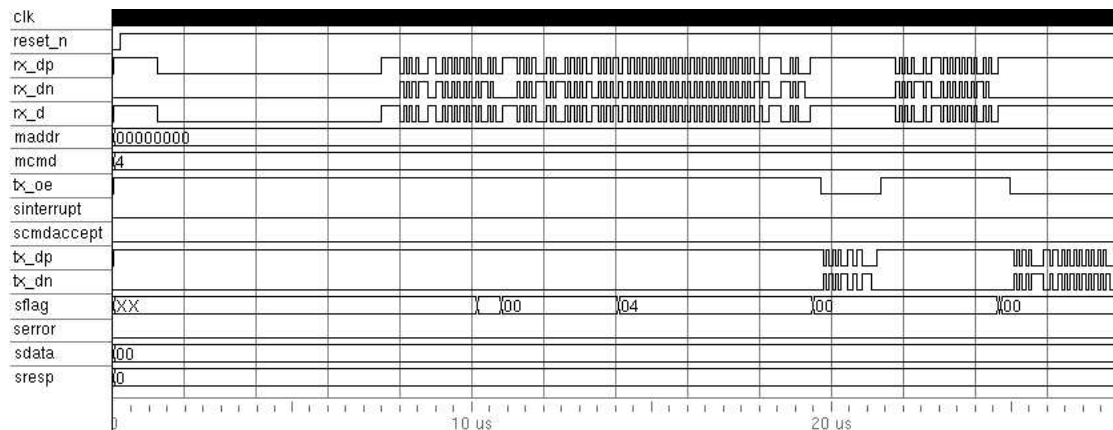


Figure 5.4: VHDL USB Model Timing

The model of the CORDIC machine was designed so that it would require a finite state machine to express in synthesizable VHDL. In Figures 5.5 and 5.6 we can see the waveform generated by the pre- and post- simulation of the model. It is immediately apparent that they differ but this is expected since the translated model has a finite state machine implementing a loop. We can also see that the final results are accurate and that the translation faithfully implements the behavior of the original model. As we mentioned in Section 3.3 the state machine generated during the translation can be run on its own clock signal. During the testing of this model we used a clock signal for the finite state machine that was twice as fast as the global clock for the model. This was to highlight the fact the the actual extra delay introduced by the translation to a finite

state machine need not be linked to the global clock. It is likely that we can tighten the clock for the FSM much more than the global clock so that depending on the design the final timing difference could be negligible. This is however heavily dependent on the technology used to implement the design and, as such, is part of a further design phase.

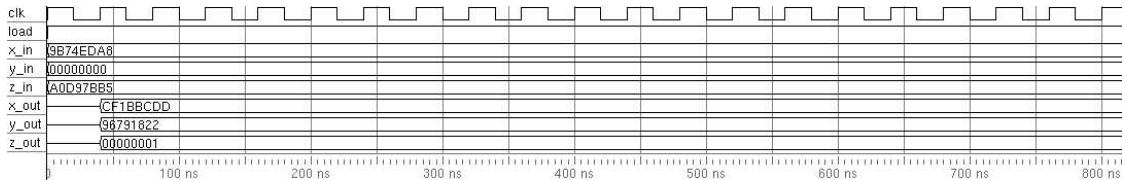


Figure 5.5: SystemC CORDIC Model Timing

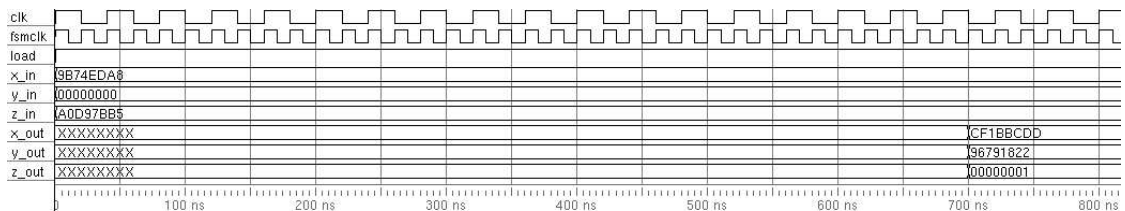


Figure 5.6: VHDL CORDIC Model Timing

For this particular model, the delay introduced by the translation to a finite state machine can be predicted. This is because the loop in the CORDIC model is not a true runtime bounded loop. It was made runtime bounded artificially by hiding the loop bound inside a function. The loop runs for exactly 32 iterations and given that it is not embedded in another control statement the generated finite state machine will take $34 * fsmclk$ where $fsmclk$ is the width of the FSM clock. The two additional cycles are to execute the statements before and after the loop. In the CORDIC model this amounts to a delay of $680ns$ and this is verified in the waveform. A true runtime bounded loop, where the amount of iterations is dependent on the input, cannot be analyzed in this manner. In such cases only a lower and upper bound can be given, even if these bounds are only the range limits on the loop condition.

5.3 Conclusion

To test the tool we translated a series of models. The models we translated represent some real world hardware models. We found that, for models that do not contain any runtime bounded loops, we can produce an accurate translation of the models. Both in algorithmic behavior and in the timing of signals the pre- and post-simulation results were identical. The only difference we found in the models revolves around the initial values of signals and variables, where the default initial values differed between SystemC and VHDL. Adapting the tool to compensate for this seems a logical course of action.

The model of the CORDIC machine correctly generated the finite state machine to represent the runtime bounded loop. Both the pre- and post-simulations produced

the correct algorithmic results. While we can derive the exact delay introduced by the translation for this specific model, this is not possible in general. A model requiring a mapping to a finite state machine will require the intervention of the user to confirm the correct behavior of the translation and might require adaptation of the model so that it still functions correctly.

The translations given by the tool for the test models were what was expected. The tool faithfully produced valid translations for test models given the methodology we used for the translation.

Finally, we have to note that the after-”place-and-route” simulations were not performed, because their results mostly differ from behavioral simulation results due to synthesis inference rules and targeted technology. Therefore, manual effort is needed to match them and this was not the focus of our work or of this thesis.

Conclusions

Translation of SystemC to VHDL allows for synthesis of SystemC without the need to develop separate tools for SystemC synthesis. It provides a means to leverage existing technology for synthesis while using a modeling paradigm tailored for hardware/software co-design. The error-prone manual translation from a SystemC model to a synthesizable VHDL model can be reduced to a manual reduction of the SystemC model to conform with our tools restrictions. Even this manual reduction can be avoided by modeling the SystemC model within our restrictions to begin with.

In this chapter we give a summary of this thesis in Section 6.1. We give an overview of the main contributions made in Section 6.2 and present areas of future work in Section 6.3.

6.1 Summary

Mapping SystemC to VHDL can be accomplished with an amount of restrictions on the SystemC input. While the removal of many of the dynamic features of SystemC and the more abstract elements of C++ reduces the expressiveness of these languages, we feel that it does not greatly impair their use in the description of hardware models. Especially if one considers that altering an existing model so that it conforms to the restrictions will be easier than rewriting that model in VHDL manually. Since in that latter case the entire model needs to be rewritten when this might not be needed to create a SystemC model that conforms to our restrictions.

A SystemC model written within the restrictions we have described in Chapter 2 can be mapped to VHDL with our mappings. For some specific constructs extra expressions need to be introduced to simulate C++ semantics and a considerable amount of transformation needs to occur to map runtime bounded loops to a finite state machine. The implicit conversions in assignments defined by SystemC require us to pass the r-value of an assignment through a conversion function to make these conversions explicit. While this adds a lot of code to the VHDL mapping we must realize that all of these cast functions only serve to instruct the VHDL compiler what is intended with the data on a bus and will not add any logic during synthesis of the VHDL model.

The behavior is maintained throughout the mappings and the resulting translation will in principle always return a behaviorally valid translation even if the timing of the model is suspect. The suspect timing occurs when a runtime bounded loop is translated into a finite state machine. Therefore, the use of a runtime bounded loop must always be done with the final translation of the loop in mind. While this reduces the use of such loops, the loops remain a powerful tool to describe complex constructs very compactly.

The tool translates the model starting at the binding function. This allows the tool to only translate that which is directly relevant for the module in question. After all the

fields of the module are translated we start collecting the structural information from the binding constructor. As we find processes we record them for later processing. When all structure information has been translated we translate each process in turn. Since the tool only translates that which is used directly by the model in question, we cannot use functions that are only declared. We require that everything be fully defined in the source so that it can all be translated. There is no ability to use linking semantics in the context of the tool.

As GCC is used as the front-end the tool accepts any model that is also accepted by GCC. We can give clear errors when translating the module if any invalid constructs are found because we have parsed the entire model and have not limited ourselves to a parser for only the synthesizable subset. This also improves the ability to extend the tool since the parser already accepts anything that can be expressed in SystemC.

To test the tool we translated a series of models. The models we translated represent some real world hardware models. We found that, for models that do not contain any runtime bounded loops, we can produce an accurate translation of the models. Both in algorithmic behavior and in the timing of signals the pre- and post-simulation results were identical. The only difference we found in the models revolves around the initial values of signals and variables, where the default initial values differed between SystemC and VHDL. Adapting the tool to compensate for this seems a logical course of action.

The model of the CORDIC machine correctly generated the finite state machine to represent the runtime bounded loop. Both the pre- and post-simulations produced the correct algorithmic results. While we can derive the exact delay introduced by the translation for this specific model, this is not possible in general. A model requiring a mapping to a finite state machine will require the intervention of the user to confirm the correct behavior of the translation and might require adaptation of the model so that it still functions correctly.

The translations given by the tool for the test models were what was expected. The tool faithfully produced valid translations for test models given the methodology we used for the translation.

Finally, we have to note that the after-”place-and-route” simulations were not performed, because their results mostly differ from behavioral simulation results due to synthesis inference rules and targeted technology. Therefore, manual effort is needed to match them and this was not the focus of our work or of this thesis.

6.2 Main Contributions

- We studied the SystemC library and C++. We found that, within some restrictions, we can provide a translation to synthesizable VHDL making the SystemC model synthesizable. A mapping between SystemC and VHDL was developed and implemented in a tool.
- The semantics of SystemC and C++ can be expressed in synthesizable VHDL within our restrictions. Only runtime bounded loops cannot be fully expressed because of the VHDL restriction on runtime bounds in loops.

- We found that we can provide a feasible mapping of complex control statements, such as runtime bounded loops, to synthesizable VHDL. While the process requires user insight it provides a powerful mechanism to describe complex behavior.
- Within the restrictions we place on the SystemC input, the tool provides full constant propagation and folding. It performs these more aggressively than a traditional compiler because the locations of influence are reduced within our restrictions. This propagation can attempt to analyze the conditions on loops such that they are bounded only by constants and a direct mapping to VHDL can be made.

6.3 Future Directions

Future work on this topic can be directed in two areas.

- Relaxation of the restrictions on the input:
 - Since no standard implementation of a synthesizable fixed and floating point data type exists for VHDL we do not allow these types. By providing synthesizable VHDL implementations of fixed point and floating point data types we can remove the restrictions on their use.
 - The translation of classes was avoided due to the needed restrictions on them, hereby negating their primary use. A mechanism to flatten class hierarchies into single noninheriting classes can reduce the restrictions we have placed on them. This can allow for a useful implementation of classes within the tool.
 - Since classes were not considered in the tool, we also did not allow templates. If a useful implementation of classes can be made then we can consider an implementation of templates as well.
- Improvement of the tool:
 - The current constant propagation and constant folding mechanism is based on single values. Providing the ability to propagate ranges can allow a reduction in the amount of cycles introduced by the finite state machine mapping of runtime bounded loops.
 - Constant propagation can also be used to provide a mechanism to parameterize modules. Allowing the constructor of a module to contain more arguments and propagating these as constants can provide the ability to parameterize modules. This can be mapped to generics in VHDL.
 - The tool currently uses a generic approach to mapping loops to a finite state machine. Further analysis of the mapping of loops, and how they are commonly used in models, could find that there are constructions that are often used. These may have a more efficient mapping than the generic one currently employed.
 - The tool currently only supports output in VHDL. This was, after all, the primary target. Output back into SystemC could prove a valuable feature for

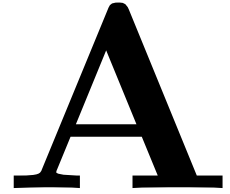
validation of the model. It would allow the mapped translation to be tested in its original testing environment and any needed changes to the model could be implemented directly in this environment.

Bibliography

- [1] *Bison - Parser Generator*, www.gnu.org/software/bison/.
- [2] *GCC Internals Manual 4.0.1*.
- [3] *OPENCORES.ORG*, www.opencores.org.
- [4] *Prosilog SystemC to VHDL Translator*, www.prosilog.com.
- [5] *VHDL-200x*, <http://www.eda.org/vhdl-200x/>.
- [6] *SystemC to Verilog Synthesizable Subset Translator*, www.opencores.org/projects.cgi/web/sc2v/overview, 2004.
- [7] *Karlsruhe SystemC Parser Suite*, 2005.
- [8] A. Johnstone, E. Scott, G. Economopoulos, *The Grammar Tool Box: a case study comparing GLR parsing algorithms*, The 4th Workshop on Language Descriptions, Tools and Applications, 2004.
- [9] C. Cote, Z. Zilic, *Automated SystemC to VHDL Translation in Hardware/Software Codesign*, The 9th IEEE International Conference on Electronics, Circuits and Systems, 2002.
- [10] D. Grune, H. Bal, C. Jacobs, K. Langendoen, *Modern Compiler Design*, John Wiley and Sons, Ltd, 2002.
- [11] E. Grimpe, B. Timmermann, T. Fandrey, R. Biniash, F. Oppenheimer, *SystemC Object-Oriented Extensions and Synthesis Features*, Forum on Specification and Design Languages, 2002.
- [12] E. Grimpe, F. Oppenheimer, *Object-Oriented High Level Synthesis Based on SystemC*, The 8th IEEE International Conference on Electronics, Circuits and Systems, 2001.
- [13] F. Yang, H. Mei, W. Yuan, Q. Wu, Y. Guo, *Experiences in building C++ front end*, SIGPLAN Not. **33** (1998), no. 9, 95–102.
- [14] Institute of Electrical and Electronics Engineers, *IEEE 1076-1987: IEEE Standard VHDL Language Reference Manual*, 1987.
- [15] Institute of Electrical and Electronics Engineers, *IEEE 1076-1993: IEEE Standard VHDL Language Reference Manual*, 1993.
- [16] Institute of Electrical and Electronics Engineers, *IEEE Std 1076.6-1999: IEEE standard for VHDL Register Transfer Level (RTL) synthesis*, 1999.
- [17] International Standards Organization, *ISO/IEC 14882-1998: C++ Standard Manual*, 1998.

-
- [18] International Standards Organization, *ISO/IEC 14882-2003: C++ Standard Manual*, 2003.
 - [19] M. Radetzki, *Synthesis of Digital Circuits from Object-Oriented Specifications*, Ph.D. thesis, Carl von Ossietzky Universität Oldenburg, 2000.
 - [20] Motorola, *A Hardware Random Number Generator*, (2002).
 - [21] M. Moy, F. Maraninchi, and L. Maillet-Contoz, *PINAPA: An Extraction Tool for SystemC Descriptions of Systems-on-a-Chip*, (2005).
 - [22] The Open SystemC Initiative, *SystemC Version 2.0 User Guide*, 2002.
 - [23] The Open SystemC Initiative, *Draft Standard SystemC Language Reference Manual*, 2005.
 - [24] J. Roskind, *A YACC-able C++ 2.0 grammar, and the resulting ambiguities*, (1990).
 - [25] Synopsys, *CoCentric SystemC Compiler: Behavioral Modeling Guide*, 2001.
 - [26] Synopsys, *CoCentric SystemC Compiler: RTL User and Modeling Guide*, 2001.
 - [27] N. Churcher W. Irwin, *A Generated Parser of C++*.

VHDL Abstract Syntax Tree



In this appendix the tree structure used to represent the VHDL translation is presented. There are 6 main tree node classes: Declarations, Expressions, Types, Statements, Identifiers and Integer Constants. Each of these classes can represent a variety of actual tree codes. For each of the classes we will lay out the tree codes that are available and the macros used to access the data in the nodes. It is the user's responsibility to ensure that a macro is only used on a tree node of the correct type.

A.1 Common

There are some common utility node types that consist only of the common tree structure:

- *VH_TREE_LIST*

Represents a list of tree nodes. Use *VH_TREE_CHAIN* to get the next node in the list.

- *VH_ERROR_MARK*

A node with this type represents an error in the abstract syntax tree. There is one special node with this type. If it is encountered in the tree then some error occurred while creating the tree. If an error mark is placed in the tree then an error message must be output and the tree should not be handed to the code generation routines.

There are some common macros that can be used on all tree nodes:

- *VH_TREE_CODE(node)*

The tree code associated with this node. This is one of the tree codes listed below.

- *VH_TREE_TYPE(node)*

The tree node representing the type of this node. This is not set for all tree nodes. Mainly declarations have this set to represent their type, but it could also be set on certain expressions.

- *VH_TREE_HASH(node)*

Returns a hash value for this tree node. This can be used to create a hash table of tree nodes.

- *VH_TREE_FLAG*(*node*)
A generic flag on a node. Can be used for any purpose. Used for *VH_PROCEDURE_DECL* to mark a returning procedure.
- *VH_TREE_COMMENT*(*node*)
A *VH_STRING_CST* node with a comment to be output in the final source.
- *VH_TREE_LINE*(*node*)
A *VH_INTEGER_CST* node with the line in the original source this node was made for.
- *VH_TREE_FILE*(*node*)
A *VH_STRING_CST* node with the file name of the original source this node was made for.
- *VH_TREE_VALUE*(*node*)
This is a special value for *VH_TREE_LIST* nodes, gets the node for the position in the list represented by node. Equivalent to *VH_TREE_TYPE*.
- *VH_TREE_CHAIN*(*node*)
Gets the next element in a list of nodes. All nodes can contain a successor node, not only *VH_TREE_LIST*. The *VH_TREE_LIST* nodes are only used when the list is a secondary list of nodes and not the primary node position. For example, a list of arguments for a *VH_CALL_EXPR* node.
- *VH_TREE_CHAIN_APPEND*(*c,n*)
Append the node *n* to node *c* in the tree chain. The macro also equals the final list so that
newnode = VH_TREE_CHAIN_APPEND(head, tail);
is valid.

A.2 Declarations

- *VH_SIGNAL_DECL*
Represents the declaration of a signal
- *VH_IN_PORT_DECL*
Represents the declaration of an in port
- *VH_OUT_PORT_DECL*
Represents the declaration of an out port
- *VH_INOUT_PORT_DECL*
Represents the declaration of an inout port

- *VH_CONST_DECL*
Represents the declaration of a constant
- *VH_VAR_DECL*
Represents the declaration of a variable
- *VH_COMPONENT_DECL*
Represents the instantiation of a component
- *VH_IN_PARAM_DECL*
Represents the declaration of an in parameter
- *VH_OUT_PARAM_DECL*
Represents the declaration of an out parameter
- *VH_INOUT_PARAM_DECL*
Represents the declaration of an inout parameter
- *VH_ENTITY_DECL*
Represents the declaration of an entity
- *VH_PROCESS_DECL*
Represents the declaration of a process
- *VH_PROCEDURE_DECL*
Represents the declaration of a procedure
- *VH_TYPE_DECL*
Represents the declaration of a type
- *VH_PACKAGE_DECL*
Represents the declaration of a package
- *VH_BUILT_IN_DECL*
Represents the declaration of a built-in function.
This serves as a placeholder for *VH_PROCEDURE_DECL* nodes in *VH_CALL_EXPR* nodes.

The following macros are defined to access the data in declaration nodes:

- *VH_DECL_CHECK(node)*
Casts a *vh_common** to a *vh_declaration** node. Used by all other declaration macros.

- *VH_DECL_IDENTIFIER*(*node*)
Returns the identifier node for this declaration. All declarations have some kind of identifier associated with them.
- *VH_DECL_NAME*(*node*)
Returns the string representing the identifier for this declaration. It returns a *char** for the identifier node.
- *VH_DECL_SPEC*(*node*)
Returns the special tree argument. The meaning of this differs for each tree code. See the node specific macros further down.
- *VH_DECL_BODY*(*node*)
Returns the body tree of this declaration. Only *VH_ENTITY_DECL*, *VH_PROCESS_DECL* and *VH_PROCEDURE_DECL* have this set. For all other node types this is always NULL.
- *VH_DECL_DECLS*(*node*)
Returns the declaration list for this declaration. This list contains all the declarations for the declarative region of the declaration. As with *VH_DECL_BODY*(*node*) this is only set for nodes that contain a declarative region.
- *VH_PROCEDURE_RETURNS*(*node*)
This macro is only used on *VH_PROCEDURE_DECL* nodes. It is flag marking whether the procedure's first argument is a synthesized return variable. If this is so, then calls to this procedure need to call it with a temporary variable as the first argument.

The following macros are aliases for *VH_DECL_SPEC*. They serve to identify the purpose of *VH_DECL_SPEC* when it is used.

- *VH_ENTITY_DECL_PORTS*(*node*)
Returns the ports declared for an entity declaration. This is one of the port declaration nodes.
- *VH_COMPONENT_DECL_BINDS*(*node*)
Returns the bindings for a component instantiation. This is a *VH_BIND_EXPR* node.
- *VH_PROCESS_SENS*(*node*)
Returns the sensitivity list of a process. This is a *VH_TREE_LIST* node.
- *VH_FIELD_INIT*(*node*)
Returns the expression to use to initialize this field declaration with. This is a node valid in the context of an expression.

- *VH_PROCEDURE_ARGS*(*node*)

The arguments of a procedure declarations. If *VH_PROCEDURE_RETURNS*(*node*) is set, then the first argument is the argument to for returning the return value. Returns one of the parameter declaration nodes.

A.3 Expressions

The following tree codes are valid for expressions. All expressions contain two operands:

- *VH_RETURN_EXPR*

A return expression.

- *VH_VAR_EXPR*

A reference to a variable declaration. Operand 0 is the declaration referenced.

- *VH_SASSIGN_EXPR*

A signal assignment. Operand 0 is the lhs and operand 1 is the rhs.

- *VH_ASSIGN_EXPR*

A variable assignment. Operand 0 is the lhs and operand 1 is the rhs.

- *VH_SENSITIVE_EXPR*

An expression that represents sensitivity of processes. Operand 0 is the signal to be sensitive to. Operand 0 can be a *VH_VAR_EXPR* node containing the declaration to be sensitive to.

- *VH_CALL_EXPR*

A function call expression. Operand 0 is the function declaration to call and operand 1 is a *VH_TREE_LIST* node containing the parameters.

- *VH_BIND_EXPR*

An expression that represents bindings in component declarations. Operand 0 is the port in the component and operand 1 is the signal to bind. Both operand 0 and 1 might be *VH_VAR_EXPR* nodes containing the actual declaration.

- *VH_INDEX_EXPR*

An array index expression. Operand 0 is the array and operand 1 is the index.

- *VH_SLICE_EXPR*

A vector slice expression. Operand 0 is the vector and operand 1 is a *VH_RANGE_EXPR* representing the range of the slice.

- *VH_RANGE_EXPR*

A range expression represents a range in both *VH_SLICE_EXPR* and in vector type declarations.

- *VH_FIELD_EXPR*
A field accessor. Operand 0 is the record type and operand 1 is the field.
- *VH_UNARY_PLUS_EXPR*
An unary plus. Opernad 0 is the expression the plus is expressed on.
- *VH_UNARY_MINUS_EXPR*
An unary minus, same as *VH_UNARY_PLUS_EXPR*.
- *VH_CONCAT_EXPR*
Vector concatenation expression.
Operands 0 and 1 are the two operands.
- *VH_PLUS_EXPR*
An addition expression. Operands 0 and 1 are the two operands.
- *VH_MINUS_EXPR*
A subtraction expression. Operands 0 and 1 are the two operands.
- *VH_MULT_EXPR*
A multiplication expression. Operands 0 and 1 are the two operands.
- *VH_DIV_EXPR*
A division expression. Operands 0 and 1 are the two operands.
- *VH_MOD_EXPR*
A remainder expression. Operands 0 and 1 are the two operands.
- *VH_BIT_IOR_EXPR*
A bitwise inclusive or-expression. Operands 0 and 1 are the two operands.
- *VH_BIT_XOR_EXPR*
A bitwise exclusive or-expression. Operands 0 and 1 are the two operands.
- *VH_BIT_AND_EXPR*
A bitwise and-expression. Operands 0 and 1 are the two operands.
- *VH_BIT_NOT_EXPR*
A bitwise not-expression. Operand 0 is the operand.
- *VH_TRUTH_ANDIF_EXPR*
A logic and-expression. Operands 0 and 1 are the two operands.
- *VH_TRUTH_ORIF_EXPR*
A logic or-expression. Operands 0 and 1 are the two operands.

- *VH_TRUTH_NOT_EXPR*
A logic not-expression. Operand 0 is the operand.
- *VH_LT_EXPR*
A less-than-expression. Operands 0 and 1 are the two operands.
- *VH_LE_EXPR*
A less-than-or-equal-expression.
Operands 0 and 1 are the two operands.
- *VH_GT_EXPR*
A greater-than-expression. Operands 0 and 1 are the two operands.
- *VH_GE_EXPR*
A greater-than-or-equals expression. Operands 0 and 1 are the two operands.
- *VH_EQ_EXPR*
An equals-expression. Operands 0 and 1 are the two operands.
- *VH_NE_EXPR*
A not-equals-expression. Operands 0 and 1 are the two operands.
- *VH_LSHIFT_EXPR*
A left-shift-expression. Operand 0 is the item to be shifted and operand 1 is the shift amount.
- *VH_RSHIFT_EXPR*
A right-shift-expression. Operand 0 is the item to be shifted and operand 1 is the shift amount.

The following three macros can be used on expression nodes. The meaning of the operands depends on the tree code. See the tree code explanations for their use.

- *VH_EXPR_CHECK(*node*)*
Check that *node* is an expression node and cast it to the expression structure.
- *VH_EXPR_OPERAND_0(*node*)*
Returns the tree representing operand 0.
- *VH_EXPR_OPERAND_1(*node*)*
Returns the tree representing operand 1.

A.4 Types

The following tree codes are available for types:

- *VH_INTEGER_TYPE*
An integer type. Conforms to the VHDL *integer* type.
- *VH_NATURAL_TYPE*
An unsigned integer type. Conforms to the VHDL *natural* type.
- *VH_SIGNED_TYPE*
An integer type. Conforms to the VHDL *SIGNED* type.
- *VH_UNSIGNED_TYPE*
An unsigned integer type. Conforms to the VHDL *UNSIGNED* type.
- *VH_BOOLEAN_TYPE*
A boolean type. Conforms to the VHDL *boolean* type.
- *VH_LOGIC_TYPE*
A multi-level logic type. Conforms to the VHDL *std_logic* type.
- *VH_BIT_TYPE*
A bit type. Conforms to the VHDL *bit* type.
- *VH_BITVEC_TYPE*
A bit vector type. Conforms to the VHDL *bit_vector* type.
- *VH_LOGICVEC_TYPE*
A multi-value logic type. Conforms to the VHDL *std_logic_vector* type.
- *VH_ARRAY_TYPE*
An array type. The element type is given by *VH_TREE_TYPE(node)*.
- *VH_ENUM_TYPE*
An enumerator type.
Enumerators are given by *VH_ENUM_FIELDS(node)*.
- *VH_ENTITY_TYPE*
The type of an entity. This is the type used for component declarations. Ports are given by *VH_TYPE_PORTS(node)*
- *VH_RECORD_TYPE*
A record type. Fields are given by *VH_RECORD_FIELDS(node)*.

The following macros can be used on type nodes:

- *VH_TYPE_CHECK*(*node*)
Checks that *node* is a type node and casts it to the type structure.
- *VH_TYPE_IDENTIFIER*(*node*)
Returns a *VH_IDENTIFIER* node for the name of the type.
- *VH_TYPE_NAME*(*node*)
Returns a *char** for the name of the type.
- *VH_TYPE_RANGE*(*node*)
Returns the range of the type. This is valid for all types except *VH_BIT_TYPE*, *VH_LOGIC_TYPE*, *VH_ENUM_TYPE*, *VH_RECORD_TYPE* and *VH_ENTITY_TYPE*.
- *VH_TYPE_PORTS*(*node*)
Returns the ports of the type. This is valid for *VH_ENTITY_TYPE*
- *VH_RECORD_FIELDS*(*node*)
Returns the fields of a *VH_RECORD_TYPE* node.
- *VH_ENUM_FIELDS*(*node*)
Returns the enumerators of a *VH_ENUM_TYPE* node.

A.5 Statements

The following codes are available for statement nodes. Statements can have up to four operands:

- *VH_IF_STMT*
An if-statement. Operand 0 is the condition, operand 1 the then-clause and operand 2 the else-clause.
- *VH_FOR_STMT*
A for-statement. Operand 0 is the initialization expression, operand 1 is the loop condition, operand 3 the incrementation expression and operand 4 the loop body. This conforms closer to a C++ loop than a VHDL loop, but the for-loops are output as while-loops anyway.
- *VH_WHILE_STMT*
A while-statement. Operand 0 is the loop condition and operand 1 the loop body.
- *VH_DO_STMT*
A do-statement. Operand 0 is the loop condition and operand 1 the loop body.

- *VH_EXPR_STMT*
An expression statement. Operand 0 is the expression in this statement.
- *VH_CASE_STMT*
A case-statement. Operand 0 is the case-expression and operand 1 a list of *VH_WHEN_STMT*s.
- *VH_WHEN_STMT*
A when-statement. Occurs as an operand of a *VH_CASE_STMT*. Operand 0 is the value and operand 1 is the statement for this statement.
- *VH_NULL_STMT*
A null-statement. Represents the VHDL null-statement. Must be placed when a statement position is empty.

The following macros are defined for statement nodes:

- *VH_STMT_CHECK*(*node*)
Check that *node* is a statement node and cast to the statement structure.
- *VH_STMT_OPERAND_0*(*node*)
Return operand 0.
- *VH_STMT_OPERAND_1*(*node*)
Return operand 1.
- *VH_STMT_OPERAND_2*(*node*)
Return operand 2.
- *VH_STMT_OPERAND_3*(*node*)
Return operand 3.

A.6 Strings and Identifiers

There are various types of string-based nodes. There are identifiers and string literals of various types:

- *VH_IDENTIFIER*
Represents an identifier.
- *VH_STR_LIT*
A string literal. Output directly as a string.
- *VH_HEX_LIT*
A hexadecimal string literal. Output as a VHDL hex literal.

- *VH_OCT_LIT*
An octal string literal. Output as a VHDL oct literal.
- *VH_BIN_LIT*
A binary string literal. Output as a VHDL bin literal.
- *VH_CHAR_CST*
A character literal. Output directly as a character literal.

The following macros are used on nodes that represent a string of some kind:

- *VH_IDENTIFIER_CHECK(node)*
Check that *node* is an identifier node.
- *VH_IDENTIFIER_POINTER(node)*
Return the *char** for the string represented by this node.
- *VH_STRING_CHECK(node)*
Same as *VH_IDENTIFIER_CHECK(node)*. Used to differentiate the use of identifiers and string literals.
- *VH_STRING_POINTER(node)*
Same as *VH_IDENTIFIER_POINTER(node)*. Used to differentiate the use of identifiers and string literals.

A.7 Integer Constants

There is only one tree code for integer constants. The node contains a flag to mark it signed or unsigned.

- *VH_INTEGER_CST*
Represents any integer constant in the AST.

A series of macros are used to access the value of integer constants:

- *VH_INTEGER_CST_CHECK(node)*
Check that *node* is an integer constant node and cast to the integer constant structure.
- *VH_INT_CST_SIGNED(node)*
Returns true if *node* is a signed integer constant.
- *VH_INT_CST_VAL_RAW(node)*
Returns the value of the integer constant node, without regard to the whether it is signed or not. Always returns an unsigned value.

- *VH_INT_CST_VAL*(*node*)
Returns the value of the integer constant node, case to a signed integer is needed.
- *VH_BUILD_INT_CST*(*val*)
Returns a VHDL tree node representing the signed integer *val*.
- *VH_BUILD_UINT_CST*(*val*)
Returns a VHDL tree node representing the unsigned integer *val*.

A.8 Helper Nodes

There are two helper nodes defined that aid in the translation but are not a part of the final abstract syntax tree:

- *VH_EXPR_RETURN*
Represents the return value of certain functions that translate expressions. It is used to return the pre-, post- and main expressions when an expression needed to be split into multiple expressions.
- *VH_PROC_LIST*
Represents a list of process nodes. It contains both the GCC tree node for the method of the process and the VHDL tree node for the process. It is used to delay the translation of process nodes until all the structure information has been translated.

The following macros are used on these two special nodes:

- *VH_EXPR_RET_CHECK*(*node*)
Check that *node* is a *VH_EXPR_RETURN* node and cast it to the correct structure.
- *VH_EXPR_RET_PRE*(*node*)
Returns the expressions that must be evaluated before the primary expression.
- *VH_EXPR_RET_MAIN*(*node*)
Returns the primary expression of this *VH_EXPR_RETURN* node.
- *VH_EXPR_RET_POST*(*node*)
Returns the expressions that must be evaluated after the primary expression.
- *VH_PROC_LIST_CHECK*(*node*)
Check that *node* is a *VH_PROC_LIST* node and cast to the correct type.
- *VH_PROC_LIST_SYSC*(*node*)
Returns the GCC tree node for the process. This is a function declaration node in the GCC tree.

- *VH_PROC_LIST_VH*(*node*)

Returns the VHDL tree node for the process. This node has no body as of yet, but already contains all the sensitivity information.

Tool and Test Model Source

B

This appendix contains the tool and the test model sources. The contents of the appendix are contained in the attached CD-ROM. A source distribution of GCC 4.0.1 and SystemC 2.1.v1 is also included.

Curriculum Vitae



E.P.M. van Diggele was born in Etobicoke, Canada on the 9th of April 1981. He graduated in 1999 at the secondary school Rijnlands Lyceum in Oegstgeest obtaining the International Baccalaureate degree. The same year he started his study in Computer Science at the Delft University of Technology, the Netherlands. After completing the Bachelor of Science program in 2003 he joined the Computer Engineering laboratory, led by professor Stamatis Vassiliadis. Under the supervision of assistant professor Stephan Wong he worked on his MSc graduation project. His research interests include compiler design, reconfigurable computing, embedded systems and computer architecture.