Circuits and Systems
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ens.ewi.tudelft.nl/

CAS-2010-13

M.Sc. Thesis

Extracting Behavior and Dynamically Generated Hierarchy from SystemC Models

ing. J.Z.M. (Harry) Broeders

Abstract

SystemC is a popular modeling language which can be used to specify systems at a high abstraction level. Currently, SystemC tools can not cope with SystemC models for which the module hierarchy depends on dynamic parameters.

We present a novel approach to extract the dynamically generated module hierarchy and its behavior from a SystemC model. In this approach the hierarchical information is retrieved by executing the model under control of a debugger. Thereafter, the behavioral information is retrieved by using a C++ compiler extension. Finally, the behavioral information is linked with the hierarchical information. Our approach is completely non-intrusive. The SystemC model and the SystemC reference implementation can be used without any modification.

To identify the information which must be extracted by a SystemC front-end a SystemC metamodel is defined. Currently, no other detailed SystemC metamodel has been published.

We have implemented our approach in an open-source SystemC front-end called Systemc Hierarchy and Behavior Extractor (SHaBE). SHaBE is developed using a test-first approach during which more than 250 test cases were successfully implemented. The extraction of the module hierarchy of the model has a time complexity of $O(n \cdot \log n)$, where n is the number of SystemC objects used in the model.

This front-end facilitates the development of SystemC visualization, debugging, static verification, and synthesis tools.

15 December 2010

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

Extracting Behavior and Dynamically Generated Hierarchy from SystemC Models

Thesis

submitted in partial fulfillment of the
requirements for the degree of

Master of Science

in

Embedded Systems

by

ing. J.Z.M. (Harry) Broeders
born in Dirksland, The Netherlands

This work was performed in:

Circuits and Systems Group
Department of Microelectronics & Computer Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Delft University of Technology

Copyright © 2010 Circuits and Systems Group
All rights reserved.

Delft University of Technology
Department of
Microelectronics & Computer Engineering

The undersigned hereby certify that they have read and recommend to the Faculty of Electrical Engineering, Mathematics and Computer Science for acceptance a thesis entitled **"Extracting Behavior and Dynamically Generated Hierarchy from SystemC Models"** by **ing. J.Z.M. (Harry) Broeders** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dated: 15 December 2010

Chairman: ________________________

Prof. dr. ir. Alle-Jan van der Veen

Advisor: ________________________

dr. ir. René van Leuken

Committee Members: ________________________

dr. ir. Nick van der Meijs

____

Georgi Kuzmanov, Ph.D. M.Sc.

Abstract

Modern embedded systems are far to complex to describe their hardware and software at a low-level of abstraction. SystemC is a popular modeling language which can be used to specify systems at a higher abstraction level. The primary way to deal with complexity in SystemC is to apply modularization. The module hierarchy of a SystemC model is dynamically constructed during the execution of the elaboration phase of the model. This means that a system designer can build regular structures using loops and conditional statements. Currently, SystemC tools can not cope with SystemC models for which the module hierarchy depends on dynamic parameters. We present a novel approach to extract the dynamically generated module hierarchy and its behavior from a SystemC model.

In our approach the hierarchical information of a SystemC model is retrieved by executing the elaboration phase of the model under control of a debugger. Thereafter, the behavioral information of the model is retrieved by using a C++ compiler extension. Finally, the behavioral information is linked with the hierarchical information. Our approach is completely non-intrusive. The SystemC model and the SystemC reference implementation can both be used without any modification. The only precondition is that they both are compiled to include debug information.

To identify the information which must be extracted by a SystemC front-end a SystemC metamodel is defined. This metamodel, models the module hierarchy of a SystemC model at the end of the elaboration phase. Currently, no other detailed SystemC metamodel has been published.

We have implemented our approach in an open-source SystemC front-end called Systemc Hierarchy and Behavior Extractor (SHaBE). SHaBE can extract all relevant hierarchical information and a well defined subset of all behavioral information from a model. The implementation is based on open-source tools and is developed using a test-first approach during which more than 250 test cases were successfully implemented. To extract the module hierarchy from the model the source code of the SystemC reference implementation has been carefully analyzed to determine the function calls which need to be monitored. Breakpoints are placed on these function calls and crucial information is extracted by using debug commands to inspect the stack, the function arguments, the members of an object, etc. The extraction of the module hierarchy of the model has a time complexity of $O(n \cdot \log n)$, where n is the number of SystemC objects used in the model. The behavior of the model is extracted by a compiler plug-in which extracts the abstract syntax tree in static single assignment form from the source code of the functions which define the behavior of the SystemC processes.

The output of SHaBE is saved in an XML based format which describes the module hierarchy and the behavior of a SystemC Model. Presently, there is no other XML format available which can be used to describe the module hierarchy of a SystemC model as well as its behavior.

This front-end facilitates the development of SystemC visualization, debugging, static verification, and synthesis tools.

Acknowledgments

It is a pleasure to thank those who made this Master's thesis possible: my thesis advisor, my employer, a specific student, and my family.

First of all, I would like to thank my advisor dr. ir. René van Leuken for his assistance during the writing of this thesis. Thank you René for elevating me from the implementation details of which I am so fond, for showing me what scientific research is all about, for the help in structuring my thoughts by asking the right questions, and for sharpening my writing skills. I sincerely hope we will be given the opportunity to present our paper at the Design Automation Conference 2011.

I wish to express my gratitude to my employer, The Hague University, for generously funding my Master's degree and giving me the flexibility to take classes while working. I also want to thank all colleagues and students from the Academy for Technology, Innovation & Society Delft (TISD) for the interest they have shown and for the encouragements I have received.

I am indebted to Bas van den Aardweg a Bachelor student form TISD who worked three months with me on this thesis project as an internship. Bas, thank you very much for implementing the first version of the GCC plug-in. I truly enjoyed working with you.

Finally I want to thank my spouse, Marie-Louise, and my children: Anne, Theo, Koen and Linda for their love and support. Marie-Louise, I am sorry for all the times I neglected you because I wanted to study, I thank you for your loving encouragements and for exempting me of all household chores for the last three and a half years. I will try to make it up to you.

ing. J.Z.M. (Harry) Broeders
Delft, The Netherlands
15 December 2010

Contents

Abstract v

Acknowledgments vii

Contents ix

List of Figures xiii

List of Tables xv

Acronyms xvii

1 Introduction 1
- 1.1 Motivation 1
- 1.2 Goals 2
- 1.3 Contributions 3
- 1.4 Outline 3

2 SystemC 5
- 2.1 SystemC Introduction 5
  - 2.1.1 Elaboration Phase 6
  - 2.1.2 Inspecting the SystemC Module Hierarchy 7
  - 2.1.3 SystemC Data Introspection 7
  - 2.1.4 Configuration, Control and Inspection Working Group 8
- 2.2 Motivating Example 8

3 Related Work 15
- 3.1 SystemC Front-end Approaches 15
  - 3.1.1 SystemC Front-ends using the Static Approach 15
  - 3.1.2 SystemC Front-ends using the Dynamic Approach 18
  - 3.1.3 SystemC Front-ends using the Hybrid Approach 19
- 3.2 Conclusions Drawn from the Survey of SystemC front-ends 21

4 Analysis 23
- 4.1 Requirements 23
- 4.2 Considered Approaches 24
- 4.3 Our Approach 27
- 4.4 Information which has to be Extracted from a SystemC Model 28
  - 4.4.1 SystemC Metamodel 28
  - 4.4.2 SystemC Function Calls, Overloaded Operators, and Data Types used to Define the Behavior of a Model 35
- 4.5 Intermediate Representation 36
  - 4.5.1 SystemC Model Description Language (SCMDL) 39

4.5.2 Resource Directory for SCMDL . . . 41

5 Software Development Methodology: Test First **43**
5.1 Test Cases for Signals . . . 44
5.2 Test Cases for Modules . . . 46
5.3 Test Cases for Ports . . . 49
5.3.1 Test Cases for the Creation of Ports . . . 49
5.3.2 Test Cases for the Binding of Ports . . . 50
5.4 Test Cases for Exports . . . 51
5.5 Test Cases for Processes . . . 51

6 Systemc Hierarchy and Behavior Extractor (SHaBE) **55**
6.1 Extracting the Dynamically Generated Module Hierarchy . . . 55
6.1.1 Communicating with GDB . . . 56
6.1.2 Finding the SystemC Name . . . 57
6.1.3 Finding the SystemC Type . . . 57
6.1.4 Finding the C++ Type . . . 58
6.1.5 Finding the C++ Name . . . 58
6.1.6 Finding the C++ Name of a Top-Level Module or Channel . . . 61
6.1.7 Finding the Static Sensitivity of a SystemC Process . . . 62
6.1.8 Finding the Connections Between the Modules . . . 65
6.1.9 Finding the Reset Signal of a `SC_CTREAD` and its Active Level . . 66
6.2 Retrieving the Behavior of the SystemC Modules . . . 66
6.2.1 GCC Internals . . . 67
6.2.2 Communicating with SHaBEPlugin . . . 67
6.2.3 SHaBEPlugIn . . . 69
6.3 Combining the Hierarchical Information with the Behavioral Information 69

7 Results **71**
7.1 Implementation Status . . . 71
7.2 Dependencies . . . 73
7.3 Test Results . . . 74
7.4 Execution Time of SHaBE . . . 74

8 Conclusions and Future Work **77**
8.1 Conclusions . . . 77
8.2 Improving SHaBE . . . 78
8.3 Future Work . . . 79

References **81**

A C/C++/SystemC Synthesis Tools **89**
A.1 Commercial Tools . . . 89
A.2 Academic Tools . . . 91
A.2.1 CASH . . . 91
A.2.2 Fossy . . . 91

A.2.3 GAUT . . . 91
A.2.4 NISC . . . 92
A.2.5 ROCCC . . . 92
A.2.6 sc2v . . . 92
A.2.7 SPARK . . . 92
A.2.8 xPilot . . . 93
A.3 Conclusions Drawn from the Survey of C/C++/SystemC Synthesis Tools 93

B SystemC Metamodel 95

C Information which has to be Retrieved from a SystemC Model 97

D SystemC Model Description Language 99

E Resource Directory for SCMDL 109

F Test Cases 111

G Parser for GDB/MI Output Records 115

H Transformation of SCMDL documents using XSLT 117

I An SCMDL document produced by SHaBE. 121

J Execution Time of SHaBE 131

K SHaBE versus PinaVM 133

List of Figures

2.1 A function which displays the module hierarchy of a SystemC building block. . . . 7
2.2 A simple SystemC adder module. . . . 9
2.3 A SystemC constant amplifier module. . . . 9
2.4 A SystemC base module which can be used to derive synchronous modules. 10
2.5 The SystemC delay module `D` is derived from `SynchronousModule`. . . 10
2.6 A SystemC model for an Nth order FIR filter with symmetric coefficients. 12
2.7 Instantiating a 5th order FIR filter. . . . 12
2.8 A module hierarchy for a 5th order FIR filter as instantiated in Figure 2.7. 13
2.9 The AST of a) the adder module `S` and b) the adder module instantiation `*s[4]`. . . . 13

4.1 The architecture of SHaBE. . . . 27
4.2 Modules, ports, exports, processes, and channels are all derived from the same base class. . . . 29
4.3 A module can contain: channels, ports, exports, processes, and submodules. . . . 30
4.4 The associations between ports, exports, channels, and processes. . . . 32
4.5 The constrains for Figure 4.4 expressed in OCL. . . . 32
4.6 The primitive channels which are defined in the SystemC standard. . . 33
4.7 The ports which are defined in the SystemC standard. . . . 34
4.8 The processes which are defined in the SystemC standard. . . . 34
4.9 The AST for the expression `out = c * in`, where `out` is an output port, `in` is an input port, and `c` is a constant. . . . 39
4.10 The structure of a `<module>` element as defined in the XML Schema. . 40

5.1 A very simple test case for an `sc_signal` object and its expected output. 44
5.2 A test case for an `sc_signal` object with a user-defined SystemC name. 44
5.3 A test case for an array of `sc_signal` objects. . . . 45
5.4 A test case for a multidimensional array of `sc_signal` objects. . . . 45
5.5 A test case for a dynamically created `sc_signal` objects. . . . 45
5.6 A test case for an array of dynamically created `sc_signal` objects. . . . 46
5.7 A test case for a dynamically created array of `sc_signal` objects. . . . 46
5.8 A test case in which two `sc_signal` objects get the same C++ name but do not get the same SystemC name. . . . 47
5.9 A test case in which the address of a dynamically created `sc_signal` objects is stored in a global pointer. . . . 47
5.10 One of the test cases for submodules. . . . 48
5.11 A submodule can be declared without giving it a C++ name. . . . 49
5.12 A submodule can be created and stored in a base module. . . . 50
5.13 The UML diagram for the test case shown in Figure 5.12. . . . 50
5.14 An `And` gate using an input multiport. . . . 51

5.15 An instantiation of the module And which is declared in Figure 5.14. . . 52
5.16 A module with an SC_CTHREAD with an active high reset input port. . . 53

6.1 The communication between SHaBE, GDB, and the executable SystemC model. 57
6.2 The communication between SHaBE, GCC, and SHaBEPlugIn. 68

7.1 The execution time of SHaBE for different orders of the FIR filter presented in Figure 2.6. 75

B.1 The primitive channels which are defined in the synthesizable subset of SystemC. 95
B.2 The interfaces which are implemented by signals. 96
B.3 The input ports which are defined in the SystemC standard. 96

F.1 A generic $N : 1$ multiplexer. 111
F.2 A submodule can be created and stored in a base module. 112
F.3 A module which inherits ports from two different base classes. 113
F.4 The UML diagram for the test case shown in Figure F.3. 113
F.5 An instantiation of the module ResettableDff which is declared in Figure F.3. 114

G.1 The primitive channels which are defined in the synthesizable subset of SystemC. 115

H.1 The module hierarchy of the FIR filter which is instantiated in Figure 2.7 as shown by yEd. 119

I.1 The module structure used to test the FIR filter. 129
I.2 Part of the AST of the behavior function of the FIR filter. 130

List of Tables

4.1 The information which must be retrieved from every SystemC object. . 29
4.2 The information which must be retrieved from every SystemC module. 30
4.3 The process hierarchy and the top-level information which must be retrieved from a SystemC model. 31
4.4 The associations that must be retrieved from the SystemC module hierarchy. 32
4.5 The information which must be retrieved from the SystemC primitive channels. 33
4.6 The information which must be retrieved from each SystemC process. . 35
4.7 The SystemC function calls which must be recognized by a SystemC front-end. 37
4.8 The SystemC data types which must be recognized by a SystemC front-end. 38
4.9 The special operations on SystemC data types which must be recognized by a SystemC front-end. 38

7.1 The hierarchical information which can, and which can not, be extracted by SHaBE. 72
7.2 Execution times of SHaBE. 76

C.1 The hierarchical information which must be retrieved from a SystemC model. 98

J.1 The execution time of SHaBE for different orders of the FIR filter presented in Figure 2.6. 131

Acronyms

ANTLR	ANother Tool for Language Recognition
AST	Abstract Syntax Tree
BNF	Bachus Naur Form
CaS	Circuits and Systems
CCIWG	Configuration, Control, and Inspection Working Group
CDFG	Control Data Flow Graph
CFG	Control Flow Graph
CLI	Command Line Interface
DFG	Data Flow Graph
EBNF	Extended Backus-Naur Form
EDG	Edison Design Group
ESL	Electronic System-Level
FIR	Finite Impulse Response
FSMD	Finite State Machine with Data
GCC	GNU Compiler Collection
GDB	GNU DeBugger
GDB/MI	GNU Debugger Machine Interface
HLS	High-Level Synthesis
IDE	Integrated Development Environment
IP	Intellectual Property
IR	Intermediate Representation
KaSCPar	The Karlsruhe SystemC Parser Suite
LLVM	Low Level Virtual Machine
OCL	Object Constrain Language
OSCI	Open SystemC Initiative
ParSyC	Parser for SystemC
PCCTS	Purdue Compiler Construction Tool Set
PINAPA	PINAPA Is Not A PArser
RTL	Register Transfer-Level

RTTI	Run-Time Type Information
SCMDL	SystemC Model Description Language
SCV	SystemC Verification
SCXML	State Chart XML
SHaBE	Systemc Hierarchy and Behavior Extractor
SSA	Static Single Assignment
SUIF	Stanford University Intermediate Format
TLM	Transaction-Level Model
UML	Unified Modeling Language
VHDL	Very-high-speed integrated circuit Hardware Description Language
XML	eXtensible Markup Language
XSLT	eXtensible Stylesheet Language Transformations

1 Introduction

SystemC is a modeling language which can be used to describe embedded systems at different abstraction levels. An open-source SystemC simulator is available free of charge. But simulation is not the only thing for which a SystemC model can be used. Users of such a model may want to visualize, debug, statically verify, or synthesize it. Tools which fulfill these needs must have a SystemC front-end which is able to retrieve the dynamically generated hierarchy and its behavior from the model. The Circuits and Systems (CaS) group at Delft University of Technology wants to incorporate SystemC models in their design flow. A SystemC front-end must therefore be selected or developed, which is the subject of this thesis. In this introduction the problem context is given, the need to develop a SystemC front-end is explained, the main contributions of this thesis are listed and its organization is given.

1.1 Motivation

An embedded system is an electronic system which is integrated into a device or an appliance, the aim being to make the behavior of the device more intelligent. An embedded system makes the device or appliance in question easier to operate or use, more energy efficient, safer, friendlier for the environment, and/or perform better. Nowadays, almost every device with a power plug, solar cell, or battery contains an embedded system. To enlarge the flexibility and the maintainability most embedded systems not only contain hardware but also contain software. Many modern embedded systems are implemented as multi-processor systems on a single chip. Such systems are far to complex to describe their hardware and software at a low-level of abstraction. An established approach to cope with this complexity is to specify systems at the Electronic System-Level (ESL). A recent overview of ESL tools is given by Gajski et al. [38].

An ESL design flow typically starts with the development of a functional model of the system. This model is described, for example in C++ or Matlab, and verified by means of simulation. During the next step of the system-level design flow a design space exploration is performed to optimize design metrics under a given set of constraints. The resulting system specification usually consist of a complex hierarchical structure. According to Gajski et al. such an architecture-level system model will predominantly be a Transaction-Level Model (TLM) described in a system-level design language such as SystemC. This optimized model will then be further refined into implementation-level models for the system's software and hardware.

SystemC is developed by the Open SystemC Initiative (OSCI), is described in IEEE standard 1666-2005 [58], and is implemented as a C++ framework. The primary way to deal with complexity in SystemC is to apply modularization. The module hierarchy of a SystemC model is dynamically constructed during the execution of the elaboration

phase of the model. This means that a system designer can build regular structures using loops and conditional statements. Writing code which dynamically generates the hierarchical structure of a system instead of statically laying out this structure is strongly preferred because the dynamic generation can be parameterized which makes the model easier to modify, easier to extend, and easier to reuse. Currently, most ESL tools can not cope with SystemC models for which the hierarchy depends on dynamic parameters. In theory, there is no need to restrict the designer to a subset of C++ for the part of the SystemC code which describes the construction of the model i.e. the elaboration phase, but in practice, these ESL tools do have such restrictions.

The CaS group has developed a High-Level Synthesis (HLS) tool [107] which accepts a high-level description that specifies the data flow of a system. This tool can perform a design space exploration and the design can be synthesized as Register Transfer-Level (RTL) Very-high-speed integrated circuit Hardware Description Language (VHDL) using various scheduling algorithms. This tool is implemented in Matlab and also generates Matlab and VHDL test benches. The CaS group wants to include SystemC models somewhere in their future design flow. A necessary first step to incorporating SystemC code is the selection or development of a SystemC front-end. This SystemC front-end should convert the SystemC model into an Intermediate Representation (IR) which describes the dynamically generated module hierarchy and its behavior. The exact format of the IR depends on the purpose of the back-end of the tool for which this front-end is used. A SystemC visualization tool may need a different IR than a SystemC synthesis tool. Because the CaS group would like to use the SystemC front-end for more than one kind of tool, the IR needs to be as general as possible. The selection or development of such an IR is also described in this thesis.

1.2 Goals

The goal formulated at the start of this thesis project was to select or develop a SystemC front-end which can extract the dynamically generated module hierarchy and its behavior form a SystemC model. After a survey of SystemC front-ends, which will be presented in this thesis, we have drawn the conclusion that there is no open-source SystemC front-end available which can retrieve the module hierarchy as well as its behavior from a SystemC model. This conclusion is confirmed in [73] which was published after our survey was completed. Therefore we have decided to develop such a front-end.

Because of my experience with using debug information for High-Level Language (HLL) simulation of microcontroller programs [11] it occurred to me that most, maybe all, of the information which needs to be extracted from a model after the execution of the elaboration phase can be retrieved by using debug information. The only requirement to use this debug information is that the SystemC model is compiled with an option to include this information.

The main research question which will be addressed in this thesis is:

> *Is it possible to retrieve the dynamically generated module structure as well as the behavior of all modules from an executable SystemC model that includes debug information?*

1.3 Contributions

The contributions of this thesis are:

- A novel approach to develop a SystemC front-end. In our approach, the hierarchical information of a SystemC model is retrieved by executing the elaboration phase of the model under control of a debugger. Thereafter, the behavioral information of the model is retrieved by using a C++ compiler extension. Finally, the hierarchical information and the behavioral information are combined and stored as an IR which can be used by tools build upon this front-end. Our approach is completely non-intrusive, i.e., no changes are required in the standard tool flow. The SystemC model and OSCI's SystemC implementation can both be used as is. The only precondition is that both are compiled to include debug information.

- A SystemC front-end called Systemc Hierarchy and Behavior Extractor (SHaBE) implemented using this approach. This implementation is based on open-source development tools. SHaBE can extract all relevant hierarchical information and a well defined subset of all behavioral information from a model. The implementation is developed using a test-first approach during which more than 250 test cases were successfully implemented. The execution of SHaBE on an executable SystemC model takes only slightly longer than the compilation of this model. Furthermore, the extraction of the module hierarchy of the model has a time complexity of $O(n \cdot \log n)$, where n is the number of SystemC objects used in the model.

- A SystemC metamodel which models the module hierarchy of a SystemC model at the end of the elaboration phase. This model is described in the Unified Modeling Language (UML).

- An eXtensible Markup Language (XML) based format to describe the module hierarchy and the behavior of a SystemC Model. This XML-based language is called SystemC Model Description Language (SCMDL) and an XML Schema definition is provided which can be used for the verification of SCMDL documents.

- A parser for the output messages of the GNU Debugger Machine Interface (GDB/MI) which only depends on standard C++ data structures.

- A GNU Compiler Collection (GCC) plug-in which extracts the Abstract Syntax Tree (AST) in Static Single Assignment (SSA) form from the source code of a function.

1.4 Outline

Chapter 2 contains a brief SystemC primer and provides a motivating example of a SystemC model. Different categories of SystemC front-ends found in the literature are presented in Chapter 3. Chapter 4 analysis the problem of developing a SystemC front-end which can extract the dynamically generated module hierarchy and its behavior

from a model. Several solutions we have considered are discussed and our approach is introduced. The requirements for the SystemC front-end are given, the information which must be extracted is identified, and an IR which can represent this information is presented. The development method we used and the test cases we have developed are described in Chapter 5. The design and implementation of our SystemC front-end called SHaBE is presented in Chapter 6. The experimental results of using SHaBE are presented in Chapter 7. In the last chapter of this thesis conclusions are drawn and directions for future work are given.

SystemC 2

In the previous chapter the need to develop a SystemC front-end was explained. This chapter starts with a brief SystemC primer. Thereafter a motivating example is presented.

2.1 SystemC Introduction

SystemC is a modeling language [89] that can be used to describe the hierarchical structure and the behavior of complex embedded systems. SystemC can be used to describe the system at different levels of abstraction. Using SystemC a system can be described at functional level, architectural level, and implementation level. The SystemC modeling language is developed by the Open SystemC Initiative (OSCI) and is described in IEEE standard 1666-2005 [58]. OSCI provides an open-source implementation of the SystemC framework which is available free of charge. Using this implementation a SystemC model can be compiled by any standard conforming C++ compiler and can be executed. This execution simulates the model and provides information that can be used to dynamically verify the model.

SystemC is implemented as a C++ framework which consists of a class library and a simulation kernel. The library consists of classes, macros, and templates which can be used to model a concurrent system using hardware-oriented data types and communication mechanisms. A SystemC model is structured by using modules. A module encapsulates a part of the system which is being modeled and has communication ports to communicate with other modules within the model. A module can contain other modules. Communication ports can be interconnected by using channels. The simulation kernel can be used to execute a SystemC model. This execution is divided into two phases: the elaboration phase and the simulation phase. During the elaboration phase the modules are instantiated and initialized by executing their constructors. During this initialization the connections between the modules are set up. Because the modules are instantiated and connected by executing C++ code any valid C++ language construct can be used. For example: the configuration of the modules can be read from a file or may depend on command-line arguments.

The behavior of a SystemC module is defined by one or more SystemC processes. A SystemC process is defined in the form of a C++ member function that is registered with the SystemC simulation kernel by using the `SC_THREAD`, `SC_CTHREAD`, or `SC_METHOD` macro. Each of these macros has different semantics. Each process has a sensitivity list which is a list of SystemC events. An event is something that happens at a specific point in time, for example a change of value on an input port. An `SC_METHOD` process is started by the SystemC simulation kernel whenever one of the events on its sensitivity list occurs. It always runs to completion before it returns control to the

simulation kernel. An SC_THREAD process is only started once by the simulation kernel. An SC_THREAD process can suspend itself by calling the SystemC wait function. The SC_THREAD process is resumed by the simulation kernel when one of the events on its sensitivity list occurs. An SC_CTHREAD process is a special kind of SC_THREAD which is only sensitive to a certain edge of a clock input port. This explains the extra C in the macro name SC_CTHREAD. Because the behavior of the model is defined in C++ member functions, all valid C++ language constructs can be used. For example: the behavior of a module can be described by using advanced data structures and algorithms from the standard C++ library or any other C++ library. Any valid C++ language construct can also be used to annotate the behavior of the model. For example: some information that is useful for the verification of the model can be written to a file during the execution of the model.

2.1.1 Elaboration Phase

Modules are defined by deriving from the SystemC library class sc_module. Modules define connection points called ports as data members. A port is instantiated from the class sc_port or from a class derived from this class. Ports can be connected by means of channels. Several primitive channels are defined in the SystemC class library and are derived from the class sc_prim_channel. The module hierarchy of a model is built during the execution of the elaboration phase of the model using objects instantiated from classes which derive from sc_module, sc_port, or sc_prim_channel. All these building blocks have the same base class called sc_object.

A SystemC model is defined as a C++ program and must include the SystemC header file. When this program is compiled and linked with the SystemC library an executable version of the model is produced. This executable can be used to simulate and to dynamically verify the model. A SystemC program should not define a main function because this main function is defined inside the SystemC library. The program should define a sc_main function instead. This function must create and initialize the module hierarchy and call the SystemC library function sc_start to start the simulation of the model.

When an executable model is executed the elaboration phase starts with the execution of the function main inside the library. This function performs some initializations and then calls the sc_main function which is defined by the developer of the model. This function creates and initializes the module hierarchy by instantiation the top-level module and channel objects and their connections. The constructors of these objects are executed and these constructors can create and initializes submodules, ports, processes, channels and their connections. Eventually sc_start is called. On page 19 of the SystemC standard [58] it is incorrectly stated that the elaboration phases ends when sc_start is called. As is explained on page 21 of the standard, the sc_start function calls all before_end_of_elaboration callback functions which are defined by the developer of the model. Each class which is defined by the user and inherits from the base class sc_object can define such a callback. The developer of the model is allowed to extent the module hierarchy in these before_end_of_elaboration callback functions. After all before_end_of_elaboration callback functions have been executed, the sc_start function calls all end_of_elaboration callback functions which are de-

fined by the developer of the model. Each class which is defined by the user and inherits from the base class `sc_object` can define such a callback. The developer of the model is not allowed to extent or change the module hierarchy in these `end_of_elaboration` callback functions. These callback functions can be used to perform some design rule checks or to print some diagnostics at the end of the elaboration phase. After all `end_of_elaboration` callback functions have been executed, the elaboration phase is finished and the `sc_start` function will enter the simulation phase.

2.1.2 Inspecting the SystemC Module Hierarchy

In SystemC modules can only be instantiated and connected to each other before the end of the elaboration phase [58]. When the module hierarchy is created it can be inspected by using some member functions from the class `sc_object`. This class provides member functions which can be used to travel through the module hierarchy. For example: `get_child_objects` and `get_parent_object`. The member function `kind` can be used to identify the kind of a specific building block. The function `display_hierarchy` given in Figure 2.1 displays the name and kind of a building block that is passed as a `sc_object*` parameter. If a building block contains other building blocks then the name and kind of these building blocks are also displayed by calling the function recursively.

```
void display_hierarchy(sc_object* objp, int level=0) {
    for (int space(0); space<level*4; ++space)
        cout<<" ";
    cout<<objp->kind()<<" named "<<objp->name()<<endl;
    const vector<sc_object*>& children(objp->get_child_objects());
    for (vector<sc_object*>::size_type i(0); i < children.size(); ++i)
        display_hierarchy(children[i], level+1);
}
```

Figure 2.1: A function which displays the module hierarchy of a SystemC building block.

2.1.3 SystemC Data Introspection

Introspection is a capability of a programming language to determine some properties of its state at run-time. Reflection is a more powerful capability of a program language to modify its own structure and behavior. C++ only provides type introspection which is called Run-Time Type Information (RTTI). RTTI makes it possible to do dynamic type casting using the `dynamic_cast<>` operation. Using the `typeid` keyword, RTTI can also be used to determine at run-time the class from which an object is instantiated. The `typeid(obj)` operation provides a reference to an object from the class `type_info`. This class has a very limited interface. It only provides comparison operators and a member function `name` which returns a human readable-name of the class from which the object `obj` was instantiated.

A program written in a programming language which supports data introspection can observe objects within that program at run-time. C++ does not provide any direct

support for data introspection. The SystemC Verification (SCV) standard library [59] uses data introspection to enable the observation of arbitrary data types. Using a C++ meta programming technique called type traits [82] the SCV library provides an interface `scv_extensions_if` through which type information can be retrieved from all data objects within the SystemC model. This interface is provided for all standard SystemC data types and also for all standard C++ data types. A SystemC model designer can also provide this interface for user-defined types within the model by using template specialization. The SCV library also supports value access from, and value assignment to, data objects as well as value randomization. Although this is called introspection in the SCV library's documentation the library actually provides the more powerful concept of reflection.

Because SystemC does not provides data introspection in the core language a SystemC tool can only use introspection if the SCV library is used in the model. This means that SystemC data introspection cannot be utilized by a tool which aims to support all SystemC models.

2.1.4 Configuration, Control and Inspection Working Group

In February 2009 the OSCI Configuration, Control, and Inspection Working Group (CCIWG) was established. This working group's initial focus is on configuration according to the working group's chair Trevor Wieman [109]. As far as we know this working group has not performed any work in the area of SystemC introspection.

2.2 Motivating Example

In this section a SystemC model for a specific kind of Finite Impulse Response (FIR) filter is presented. This model shows why it is difficult to statically retrieve the SystemC module hierarchy from a model. The FIR filter module uses three different kind of submodules: module `S` models a simple adder, module `M` models a simple multiplier which multiplies an input value with a constant filter coefficient, and module `D` models a register. All modules are defined as templates so they can be used with different data types, e.g., `double`, `sc_fixed`, etc. The module `S` is shown in Figure 2.2. It has two input ports `in1` and `in2`, and one output port `out`. The behavior is described in an `SC_METHOD` process which executes the member function `behavior` when triggered. This process is statically sensitive to any change of value at one of its inputs. The member function `behavior` is therefore executed each time the value on one of the inputs changes. When this member function executes, it reads the input values from the input ports, calculates their sum, and writes this sum to the output port.

The module `M` is shown in Figure 2.3. This module multiplies the value on the input port with a constant value and writes the result to the output port, every time the input value is changed. Please note that the `write` and `read` functions are not called explicitly but are called implicitly by overloaded operators which are defined in the SystemC library. The constant value to be used in the multiplication is provided to the module upon creation as a constructor argument. Therefore, this module can not use the macro `SC_CTOR` but must explicitly define a constructor which explicitly calls

```
template<typename T>
SC_MODULE(S) {
    sc_in<T> in1, in2;
    sc_out<T> out;
    SC_CTOR(S) {
        SC_METHOD(behavior);
        sensitive << in1 << in2;
    }
private:
    void behavior() {
        out.write(in1.read() + in2.read());
    }
};
```

Figure 2.2: A simple SystemC adder module.

the base class `sc_module` constructor. Because this module defines a process and the macro `SC_CTOR` is not used the macro `SC_HAS_PROCESS` must be used instead.

```
template<typename T>
SC_MODULE(M) {
    sc_in<T> in;
    sc_out<T> out;
    M(const sc_module_name& name, const T& cc): sc_module(name), c(cc) {
        SC_METHOD(behavior);
        sensitive << in;
    }
private:
    void behavior() {
        out = c * in;
    }
    const T c;
    SC_HAS_PROCESS(M);
};
```

Figure 2.3: A SystemC constant amplifier module.

Figure 2.4 shows the module `SynchronousModule` which can be used as a base class for modules with a clock input, a reset input, and an `SC_CTHREAD` process. This process is sensitive to a falling edge on the input port `clk`. The input port `reset` is declared to be an active high reset signal for the `SC_CTHREAD` process. The member function which should describe the behavior of the `SC_CTHREAD` is defined as a pure abstract member function. This means that this function must be overridden in a derived class to create a concrete module which can be instantiated. The process is declared and the reset signal is defined in the member function `init` which is called from the constructor of the module. The use of this `init` member function makes it more complicated for a SystemC front-end to find the module that is associated with the process and the reset signal.

```
SC_MODULE(SynchronousModule) {
    sc_in_clk clk;
    sc_in<bool> reset;
    SC_CTOR(SynchronousModule) {
        init();
    }
private:
    void init() {
        SC_CTHREAD(behavior, clk.neg());
        reset_signal_is(reset, true);
    }
    virtual void behavior() = 0;
};
```

Figure 2.4: A SystemC base module which can be used to derive synchronous modules.

The module `D` which is shown in Figure 2.5 is derived from the base class `SynchronousModule` and models a simple register. The member function `behavior` is overridden to define the behavior of this module. This function will be called when the simulation phase is started and will be suspended when the function `wait` is called. The member function `behavior` is resumed upon a falling edge of the `clk` input port. If the `reset` input port is active during a falling edge of the `clk` input port, then the `SC_CTHREAD` process is reset by calling the member function `behavior` from the beginning. Defining the module `D` this way makes it more complicated for a SystemC front-end to find the code that is associated with the `SC_SCTHREAD` which is declared in the base class.

```
template<typename T>
struct D: public SynchronousModule {
    sc_in<T> in;
    sc_out<T> out;
    D(const sc_module_name& nm): SynchronousModule(nm) {
    }
private:
    virtual void behavior() {
        out.write(T());
        while(1) {
            wait();
            out.write(in.read());
        }
    }
};
```

Figure 2.5: The SystemC delay module `D` is derived from `SynchronousModule`.

The module `FIR`, which is shown in Figure 2.6, defines a hierarchical model for an Nth order FIR filter with symmetric coefficients. The order of the filter is passed to the module using the template parameter `ORDER` and the coefficients are passed to the

module using the constructor parameter `coeff`. These coefficients can be unknown at compile time because they can be, for example, read from a file during the execution of the elaboration phase of the model. The hierarchical structure of the filter is dynamically generated during the elaboration phase by executing the constructor of the FIR module. The multipliers, adders, and delay submodules are dynamically created by calling `new`. The hierarchical structure of the FIR filter is created by binding the ports of these submodules to internal signals of the FIR module or to ports of this module. These connections are made during the execution of the elaboration phase. For example, the output of the last adder submodule must be connected to the output port `result` of the filter. The output of every other adder submodule `*s[i]` must be connected to an internal signal `sout[i]`. This is described by the `if` statement shown in Figure 2.6. Please note that the submodules must be dynamically created by using `new` because they have a non empty constructor as required by the SystemC standard [58] p. 31 and therefore can not be placed in a static array.

Figure 2.7 shows an instantiation of the `FIR` module. The filter coefficients are read from a file and passed to the module `fir`. The date type to be used (`sc_fixed<32, 2>`) and the order of the filter (`5`) are passed as template parameters to the `FIR` module. This instantiation generates the module hierarchy shown in Figure 2.8 during the execution of the elaboration phase of the model. The names of the signals and ports are the expressions which are used to access these objects from the C++ code which describes the behavior of the model. The `clk` and `reset` inputs of the `fir` module are connected to all `d` modules but these internal signals are not shown in Figure 2.8. To prevent clutter only a few internal signal names are shown. Determining this structure from a static analysis of the code of Figure 2.6, i.e., without executing the code, is very difficult to say the least.

The behavior of the filter is determined by its hierarchical structure and by the behavior of its submodules. For example, the behavior of each adder is described in the member function behavior shown in Figure 2.2. This behavior can be visualized as an Abstract Syntax Tree (AST), see Figure 2.9a. This information can be combined with the hierarchical information. For example, the AST for the submodule `*s[4]` is shown in Figure 2.9b. Please note that the actual signals which are read and written are shown in the AST of Figure 2.9b. This makes it possible to construct the complete AST of the FIR filter by connecting all individual ASTs.

The complete code for the example presented in this section can be found at `http://shabe.sourceforge.net/test_programs/fir_trans_sym`.

```
template <typename T, unsigned int ORDER>
SC_MODULE(FIR) {
    sc_in_clk clk;
    sc_in<bool> reset;
    sc_in<T> sample;
    sc_out<T> result;
    FIR(sc_module_name name, const T coeff[ORDER]): sc_module(name) {
        for (unsigned int i(0); i<MULT; ++i) {
            m[i] = new M<T>("", coeff[i]);
            m[i]->in(sample);
            m[i]->out(mout[i]);
        }
        for (unsigned int i(0); i<ORDER; ++i) {
            s[i] = new S<T>("");
            s[i]->in1(dout[i]);
            s[i]->in2(i<MULT-1 ? mout[i+1] : mout[ORDER-(i+1)]);
            if (i==ORDER-1)
                s[i]->out(result);
            else
                s[i]->out(sout[i]);
            d[i] = new D<T>("");
            d[i]->clk(clk);
            d[i]->reset(reset);
            d[i]->in(i==0 ? mout[0] : sout[i-1]);
            d[i]->out(dout[i]);
        }
    }
    ~FIR() {
        for (unsigned int i(0); i<MULT; ++i) {
            delete m[i];
        }
        for (unsigned int i(0); i<ORDER; ++i) {
            delete s[i];
            delete d[i];
        }
    }
private:
    static const unsigned int MULT = ORDER/2 + 1;
    M<T> *m[MULT];
    S<T> *s[ORDER];
    D<T> *d[ORDER];
    sc_signal<T> mout[MULT], sout[ORDER-1], dout[ORDER];
};
```

Figure 2.6: A SystemC model for an Nth order FIR filter with symmetric coefficients.

```
sc_fixed<32, 2> coeff[3];
read_from_file(coeff, 3);
FIR<sc_fixed<32, 2>, 5> fir("fir", coeff);
```

Figure 2.7: Instantiating a 5th order FIR filter.

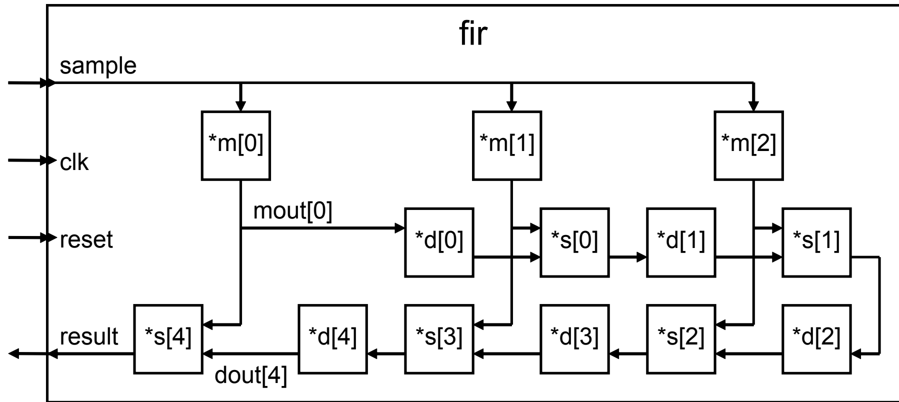


Figure 2.8: A module hierarchy for a 5th order FIR filter as instantiated in Figure 2.7.

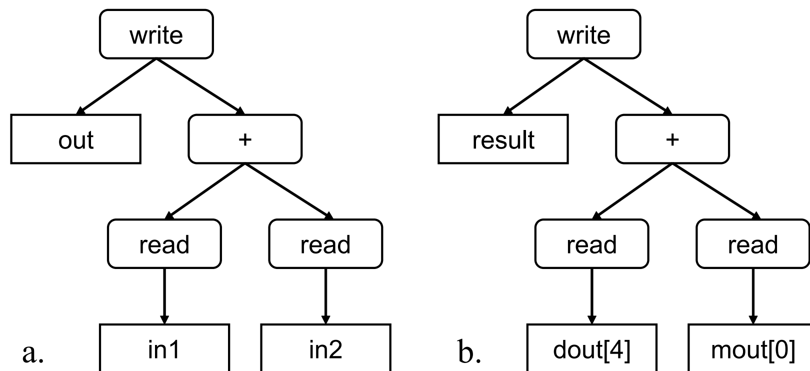


Figure 2.9: The AST of a) the adder module `S` and b) the adder module instantiation `*s[4]`.

Related Work 3

In this chapter, three different categories of SystemC front-ends found in the literature are identified. Examples from each category are described and discussed. We were not able to find a SystemC front-end which can retrieve the module hierarchy as well as the behavior of all modules from a SystemC model. An approach for developing such a SystemC front-end is described in the next chapter.

Because the Circuits and Systems (CaS) group is especially interested in the synthesis of SystemC models we have also studied some academic and commercial C/C++/SystemC synthesis tools. We were aiming to find a tool with a well defined Intermediate Representation (IR) which we could use as the output format of our SystemC front-end, but did not succeed. This study is presented in Appendix A.

3.1 SystemC Front-end Approaches

We have identified three different approaches taken by existing SystemC front-ends:

- develop a SystemC parser. This parser can be based on an existing C++ parser or an existing tool containing a C++ parser or it can be developed from scratch. A SystemC parser must be able to parse C++ because SystemC can be seen as a language extension of C++. Therefore a SystemC parser can easily extract the behavior of a SystemC model because this behavior is specified in C++ code. A SystemC parser recognizes the SystemC classes and constructs and can statically retrieve part of the module hierarchy from the SystemC model by using pointer analysis [52]. We call this the static approach.

- avoid the use of a SystemC parser by providing a modified version of the SystemC framework. When the SystemC model is compiled and executed within this modified framework the output produced will not be the simulation results. Instead, the execution will produce the hierarchical and behavioral information. We call this the dynamic approach.

- combine the two methods described above. We call this the hybrid approach.

3.1.1 SystemC Front-ends using the Static Approach

Front-ends which follow the static approach look upon SystemC as a C++ language extension. This approach can be relatively simple if the SystemC parser is based on an existing C++ parser. The biggest challenge for the SystemC font-ends in this category is to retrieve the module hierarchy of the SystemC model. In fact, it is impossible to do this in the front-end if the module hierarchy depends on certain information which is only supplied to the model at run-time, like for instance when the structure depends on

a run-time argument. Advocates of this approach argue that this is not a major defect because models for which the module hierarchy cannot be retrieved without running the model are not used very much in practice. But this is a sophism because almost all current tools use the static approach so the causality should be reversed.

The SystemC Front-ends described in the literature which follow this static approach are: KaSCPar, ParSyC, SCOOT, and SystemCXML.

3.1.1.1 KaSCPar

The Karlsruhe SystemC Parser Suite (KaSCPar) [90] was developed at the Forschungszentrum Informatik (FZI). This parser suite consists of two components. SC2AST is a SystemC parser which retrieves the behavioral information from the SystemC model and saves the AST in a file formated in eXtensible Markup Language (XML) code. SC2XML uses the AST created by SC2AST to interpret the elaboration phase of the SystemC model. The hierarchical information retrieved in this way is saved in a file also formated in XML code. The SC2AST tool is distributed as an open source but the SC2XML tool is only distributed as a compiled Java program. KaSCPar is implemented in the Java programming language [45] using JavaCC [67]. Java Compiler Compiler (JavaCC) is an open-source parser generator which generates a parser in Java source code from a formal grammar provided in Extended Backus-Naur Form (EBNF). In the SC2XML documentation [90] several limitations are acknowledged. For example, not all SystemC functionality is implemented and the C++ conditional operator (`?:`) is not implemented either.

The FERMAT research group at Virginia Tech used KaSCPar in the development of MCF: a Meta modeling based visual Component composition Framework [77]. In this framework the components from a SystemC Intellectual Property (IP) library can be composed in a visual way, like in a schematic entry tool. KaSCPar was used to extract meta-information about the SystemC IP models in an XML format.

The University of Berlin has used KaSCPar to develop a tool for the formal verification of a SystemC model [51]. To make this possible, the SystemC model is transformed into an UPPAAL model. UPPAAL [5] is an integrated tool environment for the modeling, validation, and verification of real-time systems modeled as networks of timed automata, extended with data types. It is developed in collaboration between the Uppsala University and the Aalborg University which explains the name UPPAAL.

3.1.1.2 ParSyC

Parser for SystemC (ParSyC) [36] is a SystemC front-end which was developed at the University of Bremen. The Purdue Compiler Construction Tool Set (PCCTS) [92] was used to build ParSyC. ParSyC inputs a SystemC model and produces an AST which captures the behavioral information from the model.

ParSyC is part of SystemC Environment (SyCE) [33], an Integrated Development Environment (IDE) for system design in SystemC developed at the University of Bremen. ParSyC is used in several parts of this IDE. For example it is used in CheckSyC [46], a formal verification tool for the equivalence checking and the property checking of SystemC models.

The SyCE also contains ViSyC [47], a tool which can be used to create a schematic view of the structure of a SystemC model. Curiously, this tool did not originally use ParSyC but used a modified SystemC simulation kernel to extract the hierarchical information needed in the visualization tool at execution time. Later [43] this tool used ParSyC to visualize the structure as well as the behavior of a SystemC model. The ParSyC parser generates a symbol table that is linked to an AST. While the symbol table represents the structure of user-defined types and functions, the AST describes the behavior of the model. In the following step, this information is used as an input for the interpreter. An interpretation simulates the SystemC elaboration phase that instantiates and interconnects the modules. This tool still has some limitations since it only deals with dataflow elements which means that iterations and conditional statements have to be transferred into expressions. All conditional statements are transferred into switch statements which are visualized as multiplexers. Loops are unrolled which means that the number of iterations must be determinable after the elaboration phase.

The source code of ParSyC is not publicly available as Professor Drechsler explained to us by email:

> "The software is not available, since parts resulted in the context of industrial projects."

3.1.1.3 SCOOT

SCOOT [13] is a tool which statically analyses systems described using SystemC and extracts models that can be passed to verification tools. It was developed in cooperation between the "Eidgenössische Technische Hochschule" (ETH) Zürich and Oxford University. SCOOT uses its own C++ front-end to translate the SystemC model into a Control Flow Graph (CFG). Subsequently, static analysis techniques such as field-sensitive pointer analysis [93] are used to determine the module hierarchy, the sensitivity list of the processes, and the port bindings. After extraction this information SCOOT can re-synthesize a C++ program that does not depend on the SystemC library and that can be recompiled to produce a simulator for the original SystemC model. This simulator executes the model faster than the OSCI SystemC simulator. The source code for SCOOT is not publicly available.

3.1.1.4 SystemCXML

SystemCXML [9] was originally developed as part of the INRIA Espresso project. It is also part of the CARH framework [8] from the FERMAT research group at Virginia Tech. CARH is named after the famous computer scientist C.A.R. Hoare and it is used for the validation of system-level SystemC models. SystemCXML is a SystemC front-end that uses Doxygen a tool that generates documentation formatted in XML code. The static part of the hierarchical information can be easily retrieved from a SystemC model when the model is first processed by Doxygen. The XML file produced by Doxygen is transformed into an XML file which describes the structure of each module. SystemCXML is not capable of determining which modules are instantiated

and how they are connected. SystemCXML does create an internal data structure which can be accessed via an API for further processing. Using Doxygen eliminates the need to interface with a complex SystemC/C++ parser. SystemCXML is not able to retrieve the behavioral information from a SystemC model as acknowledged by the authors [9]:

> "In the current release version, we ignore behavioral information, disallowing use of SystemCXML for some applications such as synthesis."

3.1.2 SystemC Front-ends using the Dynamic Approach

The main appeal of the dynamic approach is the the fact that a complicated SystemC/C++ parser is not needed. Technically this approach uses a C++ parser because the C++ compiler which is used to compile the SystemC model within the SystemC framework must obviously contain one. But there is no need for a SystemC front-end in the dynamic category to interact with the parser. Because in this approach the model is actually executed, it is not so difficult to retrieve the module hierarchy of the SystemC model because the SystemC standard defines a very simple API for navigating around and discovering the module hierarchy, see Section 2.1.2.

The biggest challenge for the SystemC font-ends in the dynamic category is the retrieval of the behavior of the SystemC model. The SystemC API for module navigating can be used to find the process handles of the processes which are used to implement the behavior of the module. These process handles reveal some properties of these processes, like for example the process type: `SC_METHOD`, `SC_THREAD`, or `SC_CTHREAD`. The process handle also refers to the machine code which implements the module behavior but it does not contain any reference to the C++ code which was used to specify this behavior.

A dynamic approach can obviously only be used for a SystemC model which can be compiled and run within the modified framework. This can be a disadvantage if the front-end is used as a visualization tool. For example, a user of a visualization tool might want to view a partially developed model.

A SystemC Front-end described in the literature which follows this dynamic approach is: Quiny.

3.1.2.1 Quiny

Quiny [98] is a SystemC front-end developed as part of the "Interface and Communication based Design of Embedded Systems" (ICODES) European project. Quiny makes use of a complete run-time approach to retrieve the hierarchical and the behavioral information from the SystemC model. The name Quiny, is a blend of "Quine" and "tiny". A Quine is a program which prints out its own source code and was an inspiration for the self-reflective approach used by Quiny. Quiny was used in a SystemC synthesis tool which produces a VHDL Register Transfer-Level (RTL) description. The SystemC model is compiled and linked against the Quincy library which replaces the SystemC library. When this model is executed it subsequently produces the VHDL code. All C++ types, operators and statements as well as all SystemC types, overloaded operators etc.

must be replaced by C++ code which builds the IR of the SystemC model during the execution of the model. For example: the execution of the expression $a = b + c$ should not perform a calculation but should build the AST for the expression instead. If the variables used in the expression have a SystemC data type, for example `sc_int`, then this can be accomplished simply by operator overloading. If the variables used in the expression have a C++ built-in type, for example `int`, then operator overloading cannot be used. To solve this problem, Quiny uses C++ preprocessor macros to replace all built-in data types with user-defined data types. Built-in types which are specified by using concatenated keywords such as `unsigned int` cause a problem because they cannot be replaced by a preprocessor macro. Quiny can only overcome this problem with the help of the end-user. The end-user must use, for example, the type `Q_UINT` instead of the built-in type `unsigned int`. A similar problem arises for pointer and array declarations. There is no way Quiny can detect pointer and array variable declarations at run-time because `operator*` and `operator[]` can only be overloaded in the context of an expression and cannot be overloaded in the context of a declaration. Therefore the end-user must use the generic abstract data types `Array` and `Pointer` which are defined in the Quiny library instead of built-in arrays and pointers.

3.1.3 SystemC Front-ends using the Hybrid Approach

SystemC front-ends that fall into the hybrid category try to combine the best features of the static and the dynamic approach. The biggest challenge for this approach is to find the link between the information retrieved by the parser and the information found by executing the elaboration phase of the SystemC kernel.

SystemC Front-ends described in the literature which follow this hybrid approach are: an unnamed successor of ParSyC, PINAPA, and its successor PinaVM.

3.1.3.1 Unnamed Successor of ParSyC

The authors of ParSyC, see Section 3.1.1.2 briefly describe a hybrid approach in [42]. They use a PCCTS [92] based parser to collect the static information and a code generator to evaluate run time information. The described approach is split into four phases. First, the SystemC model is converted into an AST by the parser. During the second phase recorder functions are added to this AST and an instrumented version of the original model is generated. The elaboration phase of this model is executed during the third phase. The injected recorder functions now record the state of all variables of the model after a change of their value. This dynamic information is added to the AST during the last phase. An implementation of this approach is not available nor presented.

3.1.3.2 PINAPA

PINAPA Is Not A PArser (PINAPA) [81] is an open-source SystemC front-end which was originally developed to support LusSy [80]. LusSy is a toolbox for the analysis of a System-on-a-Chip (SoC) model described as a SystemC TLM. PINAPA provides an API to the back-end application which uses the visitor pattern [39]. PINAPA stores

the IR in the form of an AST. This AST also contains hierarchical information of the SystemC model after the elaboration phase. PINAPA is implemented as a patch for SystemC (which is only available for the old SystemC versions 2.1.1 and 2.0.1 but not for the current SystemC version 2.2.0) and a patch for GNU Compiler Collection (GCC) (only available for version 3.4.1). The patched version of GCC produces the AST of the SystemC model and the patched SystemC version is used to execute the elaboration phase to retrieve the hierarchical information from the model. PINAPA then links the behavioral information from the AST with the hierarchical information.

In the PINAPA download package there is a directory called SPINAPA which contains a SystemC to SPIRIT IP-XACT converter. The PINAPA web page gives the following information about this tool:

> "It's a prototype (understand this as "half finished" if you wish). Currently, it just reads a platform with one module, and generate the SPIRIT description for the module and its registers, if the module complies with some coding standards. The tool can hardly be useful by itself, but it's an example of usage of PINAPA."

No further information about SPINAPA could be found. A detailed description of the implementation of PINAPA can be found in Chapter 4 of the Ph.D. thesis of Matthieu Moy [79]. In 2006 Moshe Vardi gave a seminar at Rice University in which he discussed this thesis. As part of this seminar his students had to write a review of the thesis. One of the students wrote:

> "The weakest part of this thesis is the way tools were implemented. It is clear that PINAPA's implementation for example is a hack that involved modifying a specific version of GCC. This basically means that as soon as the used GCC version becomes obsolete, PINAPA will have to be rewritten."

We share this observation.

3.1.3.3 PinaVM

There have been some cross-pollinations between the different SystemC front-ends described in the literature.

David Berner the author of SystemCXML, see Section 3.1.1.4 is the co-author of a paper [64] which describes an approach for the translation of a SystemC model into the synchronous formalism SIGNAL [41], in order to use a model-checker to verify properties of the source code. The translation uses Static Single Assignment (SSA) [31] as an intermediate formalism, and the GCC compiler as a front-end. This tool is part of the French institute "Institut national de recherche en informatique et en automatique" (INRIA) Espresso project. Another paper [10] about the same tool is co-authored by Matthieu Moy, the author of PINAPA. The last sentence of this paper reads:

> "We are working on a new version of PINAPA that would use an SSA-based compiler, but this requires a substantial rework of the approach, and a complete rewrite of the code itself."

The result of this work is PinaVM [72]. PinaVM uses LLVM-GCC [69] to compile the SystemC source code into Low Level Virtual Machine (LLVM) bitcode. It uses this bitcode to execute the elaboration phase which reveals the module hierarchy of the SystemC model. The SystemC constructs in the bitcode which describes the behavior of the model are recognized by PinaVM and this information is linked to the module hierarchy which was found by executing the elaboration phase. To do this PinaVM recognizes the mangled names of calls into the SystemC library such as `read`, `write`, and `wait`. When PinaVM has identified the function calls into the SystemC library it must link specific parameters of such a call with the SystemC module hierarchy. For example, in a call to the `write` function of an output port, the parameter `this` which specifies the port which is written to must be identified. The value of this parameter can be the result of any arbitrary computation. The key idea of PinaVM is to identify the bitcodes which are used to compute the parameter of interest and then to construct a new LLVM function which contains those bitcodes and produces the value of the parameter. Once build, this function is executed and the value of the parameter can be linked with the appropriate object in the model's module hierarchy. According to the authors, this approach is limited to models in which the ports which are being used in the behavioral description can be determined statically.

3.2 Conclusions Drawn from the Survey of SystemC front-ends

The results of our survey of SystemC front-ends can be summarized as follows: KaSC-Par is not capable of parsing all C++ code and the part of KaSCPar which is capable of retrieving the module structure (SC2XML) is only distributed as a compiled Java program. The source code of ParSysC is not publicly available. The same holds for SCOOT. SystemCXML is not capable of retrieving the behavior of a SystemC module. Quiny offers a unique dynamic approach but cannot retrieve the behavior of a SystemC module without the help of the end-user. In other words, Quiny is intrusive, it cannot be used for general SystemC code. PINAPA patches GCC and it also patches the SystemC library. It uses outdated versions of GCC and the SystemC library and is therefore itself outdated. The use of PinaVM is limited to models in which the ports which are being used in the behavioral description can be determined statically. Therefore, we conclude that currently no open-source SystemC front-end is available which can retrieve the module hierarchy and its behavior from a SystemC model if the module hierarchy depends on dynamic parameters. This conclusion is confirmed in [73][1]. For all currently available open-source SystemC tools, the system designer can not utilize the full expressive power of C++ to describe the dynamic generation of the hierarchical structure of the model. Therefore, we have decided to develop an open-source SystemC front-end ourselves.

[1] This article in which several SystemC front-ends are reviewed was published June 2010 after we finished our initial literature study in November 2009.

4 Analysis

We reported in the previous chapter our failure to find an open-source SystemC front-end, which can extract the dynamically generated module hierarchy and its behavior from a model, and our decision to develop such a front-end. In this chapter, the requirements for the SystemC front-end that we want to develop are specified, the information which must be extracted from a SystemC model is identified, and an IR which can represent this information is presented. We have considered several approaches to develop a SystemC front-end which fulfills our requirements. These approaches are discussed and the final approach we have taken is introduced in this chapter.

4.1 Requirements

We want to give a designer of a SystemC model as much expressive power as possible. In particular we want to make it possible to use the full expressive power of C++ during the elaboration phase of the model. This enables the system designer to dynamically generate the hierarchical structure of the model using loops and conditional statements. A designer must be able to use this tool in combination with the freely available SystemC framework provided by OSCI and a freely available C++ compiler. The following requirements for a SystemC front-end are specified.

The front-end should:

1. be able to retrieve the dynamically generated module hierarchy as well as the behavior of all modules from a SystemC model at the end of the elaboration phase of the execution of the model.
2. be able to handle models with parameters which are unknown when the model is compiled but are provided to the model during the execution of the elaboration phase of the model. For example, via command-line arguments, console input or file input.
3. be able to process every SystemC model which complies to the C++ standard ISO/IEC 14882:1998 [60] and the SystemC standard IEEE 1666-2005 [58].
4. not be intrusive. This means that any SystemC model can be processed as it is, without any modification.
5. be developed within the time available for an Embedded Systems master's project (40 ECTS points[1]).
6. be as independent from other tools as possible. Modifying the SystemC library or kernel is not desirable, modifying a compiler is not desirable either.

[1] 40 European Credit Transfer System (ECTS) points represent a workload of 2/3 man-year.

The last two requirements may seem to contradict one another. Given the limited development time (the fifth requirement) the use of existing tools is inevitable. But if we can use these tools without modifying them via the interfaces provided by these tools we can still fulfill the sixth requirement.

To fulfill the first and second requirement listed above using the static approach (see Section 3.1.1) a parser capable of parsing SystemC/C++ as well as an interpreter which can interpret the elaboration phase of a SystemC model has to be developed. We do not think this is possible if we take the fifth requirement into account. The only implementation of a SystemC front-end using the hybrid approach (PINAPA, see Section 3.1.3.2), found at the time we made these considerations[2], does not fulfill the sixth requirement. The only implementation of a SystemC front-end using the dynamic approach (Quiny, see Section 3.1.2.1) does not fulfill the fourth requirement and it also does not fulfill the sixth requirement because it uses a modified SystemC framework. We conclude that none of the approaches used so far are able to fulfill our requirements.

4.2 Considered Approaches

If the hierarchy of a SystemC model can be dynamically generated then an obvious way to find this hierarchy is to execute the elaboration phase of the model. In the OSCI SystemC implementation there is no way to stop the execution after the elaboration phase. Therefore all currently available front-ends require a modification of the SystemC kernel to stop the execution of a model just before the simulation phase is started. After or during the execution of the elaboration phase the SystemC module hierarchy must be retrieved. The SystemC library has primitive support for retrieving the module hierarchy after the elaboration phase, see Section 2.1.2. Although these functions can reveal some of the properties of a SystemC model, other properties e.g. the type of a channel or the C++ name of a port remain hidden. This hidden information is crucial though, because without this information it is impossible to link the behavior of the model with the module hierarchy.

Introspection is the capability of a programming language to determine some properties of its state at run-time. As explained in Section 2.1.3, C++ only provides type introspection which is called RTTI. RTTI is far too limited to extract useful information from a SystemC model. A program written in a programming language which supports data introspection can observe objects within that program at run-time. The SystemC Verification (SCV) standard library, also described in Section 2.1.3, uses data introspection to enable the observation of arbitrary data types. Because SystemC does not provides data introspection in the core language, a SystemC tool can only use introspection if the SCV library is used in the model. This means that SystemC data introspection cannot be utilized by a tool which aims to support all SystemC models.

We conclude that the relevant information from the module hierarchy can not completely be retrieved after the execution of the elaboration phase. Therefore, the creation of this hierarchy must somehow be monitored during the execution of the elaboration phase. This can be implemented by calling recorder functions from the SystemC library

[2]November 2009

to register important event such as the creation of a SystemC object or the binding of a port to the interface of a channel. If we do not want to modify the SystemC library we could use Aspect-Oriented Programming (AOP) [65] to add the recorder functions without changing the library source. Standard C++ does not support AOP but it would be possible to use AspectC++ [99]. This aspect weaver is used in [25] to implement a test coverage tool for SystemC models. Assuming that we will be able to record the relevant events during the elaboration phase it still will not be possible to retrieve information which is lost when the model is compiled, such as the C++ names of ports end channels. As mentioned in Section 3.1.3.1, the authors of ParSyC briefly describe an approach to solve this problem [42]. They purpose to add the recorder functions in the AST of the code which is executed during the elaboration phase. Because the recorder code is added in the compiler, compile time information such as the C++ names of ports and channels can be passed on to the recorder functions. An implementation of this idea is not available at the moment and we do not think this idea is easy to implement because it involves modification of the AST.

It occurred to us that the compile time information needed by a SystemC front-end can be included in the executable of a model by compiling the model to include debug information. If the model is linked with a debug version of the SystemC library then it is also possible to monitor and control the execution of the elaboration phase using breakpoints and watchpoints. So we decided to use a debugger to retrieve the dynamically generated module hierarchy from a SystemC model.

Using an executable of the model as input for the front-end will enable it to process SystemC models which depend on run-time arguments or other end-user input during the elaboration phase of the execution of the model. But it limits its use to SystemC models which can be compiled without any compilation errors and which can run the elaboration phase without crashing.

We have investigated the possibility to extract the behavior of the model directly from the executable. In [103] an survey of binary synthesis techniques is presented. Binary synthesis generates a hardware implementation from an executable and uses decompilation techniques [26] to extract the Control Data Flow Graph (CDFG) from the executable. Decompilation is needed because otherwise the parallelism in the extracted behavioral description will be limited to the instruction level parallelism of the binary code. For example, loops unrolling without knowledge of the loop structure and its bounds is not possible. In [102] and [6] techniques are described to recreate the loop constructs used in the source code from the executable code. An algorithm to reconstruct basic and compound data types from the executable of a program written in C is described in [32]. We identified the following problems which prevents us to extract the behavior of a SystemC model from the executable of that model:

- Specific types are defined as template classes in the SystemC library e.g. `sc_fixed`. We were not able to find an algorithm capable of reconstructing such complicated template data types from the executable of a C++ program.

- The description of the behavior of the model is intertwined with calls into the SystemC kernel. Some calls into the SystemC kernel are implemented via virtual function calls which makes it difficult to identify such calls. In [35] a method to

decompile indirect call instructions is described which, according to the author, has the potential for enabling the recovery of object oriented virtual function calls.

Therefore we have searched for an alternative method to retrieve the behavior from a model. The locations of the source files are present in the debug information which can be incorporated in the executable of the model. Our front-end already requires the model to be compiled with an option to include debug information, so we can use this debug information to locate the source files in which the behavior of the model is defined.

The behavior of a model can be retrieved from the source code of the model by a parser. A parser converts (parses) the C++ source code into some form of IR. Therefore it is important to have some knowledge of the IRs that are being used by C++ parsers. All parsers use some kind of AST as IR. This AST is a representation of the simplified syntactic structure of the source code. An AST can be transformed into SSA form [31]. This SSA form is currently used in almost every modern compiler because several optimizations such as constant propagation, dead code elimination, strength reduction, and register allocation can be preformed more easily when the AST is in SSA form. In an AST in SSA form all variables are read and written only once and all native operations are represented by 3-address instructions $x = f(y, z)$.

Only the member functions which describe the behavior of the SystemC model need to be parsed. Because these member functions can use any valid C++ language construct, our front-end must be able to parse the complete C++ language. C++ is a complex language and so it is not easy to develop a C++ parser from scratch. Therefore we have chosen to use an existing compiler to generate the AST in SSA form of the member functions which are used to define the behavior of the SystemC model. Several open-source C++ compiler infrastructures where considered: GCC-XML [66], LLVM [69], ROSE [94], ANother Tool for Language Recognition (ANTLR) [91], and GCC [100].

GCC-XML was developed for the purpose of generating an XML description of a C++ program. Unfortunately GCC-XML does not transfer function bodies to XML. Because we are only interesting in parsing function bodies, GCC-XML is of no use to us. LLVM uses a modified version of GCC i.e., LLVM-GCC by default because the project's own front-end called Clang is reported to be still somewhat immature for parsing C++. Therefore it is more straightforward to use GCC directly instead of using LLVM-GCC. ROSE uses the Edison Design Group (EDG) front-end[3] to parse C++ code. Although the EDG source code and interfaces are protected, they may be distributed freely in binary form. Because we want to develop a open source tool we decided not to use ROSE. ANTLR is the successor of PCCTS [92] which was used to develop the SystemC front-end ParSyC, see Section 3.1.1.2. There is a C++ grammar for ANTLR available so using ANTLR for the second step of our front-end is feasible. But due to time limitations we have not investigated this any further. GCC was used in the development of the SystemC front-end PINAPA Section 3.1.3.2. PINAPA modified a specific version of GCC. This version of GCC (3.4.1) is now outdated and therefore PINAPA is outdated too. Since version 4.5.0[4] GCC can be extended and/or modified

[3] http://www.edg.com

[4] Version 4.5.0 of GCC was released on April 14, 2010.

without modifying its source code [21]. A new option `-fplugin=name.so` tells GCC to load the shared object *name*`.so` and execute it as part of the compiler. We therefore decided to write a plug-in for GCC to extract the behavior of the SystemC model.

4.3 Our Approach

We present our solution which 1) controls the elaboration phase of the execution of the model and extracts the model hierarchy, 2) retrieves the AST from the SystemC model, and 3) combines all collected information to produce the dynamically generated hierarchy and the behavior of the SystemC model. During the first step the generated hierarchy is retrieved by executing the elaboration phase of the SystemC model under the control of the open-source GNU DeBugger (GDB) [101]. The C++ member functions used to describe the behavior of the SystemC processes, and the source files in which these member functions are defined, are also identified during the first step. In step two the AST of the functions which describe the behavior of the model are retrieved from their source files. We have written a plug-in for open-source GCC [100] which can extract these ASTs. In the last step of our approach the ASTs of all analyzed functions are combined with the information found in the first step. Finally an IR is produced which contains all hierarchical and behavioral information from the model. We have implemented our approach in a SystemC front-end called Systemc Hierarchy and Behavior Extractor (SHaBE), Figure 4.1. In Chapter 6 the three steps of our approach are fully explained.

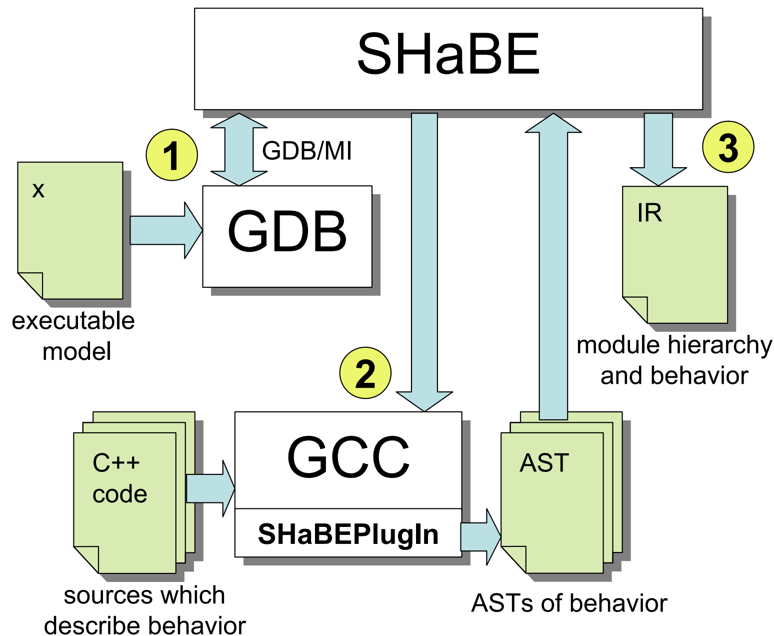


Figure 4.1: The architecture of SHaBE.

4.4 Information which has to be Extracted from a SystemC Model

The SystemC standard [58] has been carefully studied to determine which hierarchical information is encapsulated in a SystemC model at the end of the elaboration phase. This information must be retrieved by the SystemC front-end that we want to develop. It would have been very convenient if a model of the modeling language SystemC, i.e., a metamodel, would have been available. The SystemC standard does not provide such a metamodel. We have search for such a metamodel in the literature and found a few papers which describe a simplified SystemC metamodel [95][24]. These models are far to simple to use for our purpose so we decided to develop our own SystemC metamodel. The Unified Modeling Language (UML) [87] is used to create this metamodel. The metamodel is presented in Section 4.4.1 and is split up in several different figures to prevent clutter. For each class defined in the SystemC standard it was determined which data and which relations are relevant to extract. Using this metamodel the information which has to be retrieved from a SystemC model at the end of the elaboration phase is identified and listed in tables. These information items are prioritized using the following priorities:

1. An information item with this priority must absolutely be retrieved from the model because otherwise the module hierarchy can not be constructed or the hierarchical information can not be linked with the behavioral information.

2. An information item with this priority item is nice to know and will be useful for several kind of back-ends, including a synthesizer.

3. An information item with this priority is not included in the SystemC synthesizable subset [88] and is therefore not necessary to retrieve if the front-end will be used with a synthesizer back-end.

4. An information item with this priority is only needed by a specific kind of back-end e.g., a debugger.

The behavior of a SystemC model is described in member functions which are registered with the SystemC simulation kernel. These member functions can use any valid C++ language construct and any valid C++ data type. In addition to this, these member functions can use several specific function calls, overloaded operators, and data types which are defined in the SystemC standard. These function calls, overloaded operators, and data types are described in Section 4.4.2 and must be recognized by the SystemC front-end that we want to develop. There are restrictions on the C++ language constructions and data types as well as on the SystemC function calls, operators, and data types which can be used if the model must conform to the SystemC synthesizable subset.

4.4.1 SystemC Metamodel

SystemC was already introduced in Chapter 2. A SystemC model is structured by using modules which are derived from the base class `sc_module`. A module can communicate

to other modules via ports and exports. These ports and exports can be interconnected by using channels. A port requires an interface which must be provided by the external channel which is bound to the port. An export, on the other hand, provides an interface which is implemented by the internal channel which is bound to the export. The behavior of a module is defined by one or more processes. As can be seen in Figure 4.2 all building blocks for the module hierarchy are derived from the base class `sc_object`.

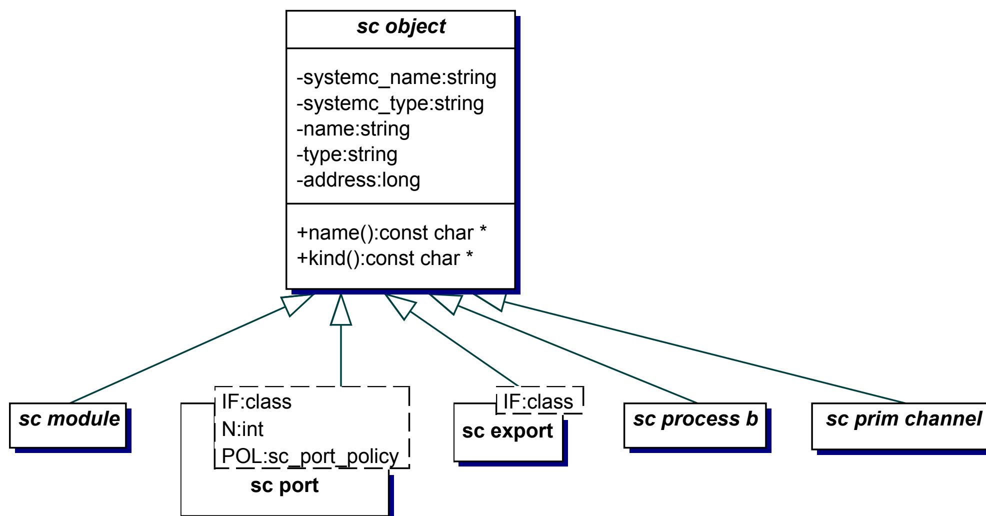


Figure 4.2: Modules, ports, exports, processes, and channels are all derived from the same base class.

Every `sc_object` which has been created during the elaboration phase of the model has an unique hierarchical SystemC name and a SystemC type. The member function `name()` returns the hierarchical name and the member function `kind()` returns the SystemC type. For example, the second port declared in Figure 2.6 has the SystemC name `fir.port_1` and has the SystemC type `c_in`. An object which has to be identified elsewhere in the model must be stored as a C++ variable. The C++ name of this variable is essential to find the relation between the hierarchical and the behavioral information of the model. The type of this variable is useful to know because it reveals the values of the template parameters used, in case the type is a template. For example, the second port declared in Figure 2.6 has the C++ name `reset` and has the SystemC type `sc_core::sc_in<bool>`. The address of the variable is only relevant in case the back-end is a debugger.

Table 4.1: The information which must be retrieved from every SystemC object.

object	information	cardinality	priority
`sc_object`	SystemC name	1	1
`sc_object`	SystemC type	1	1
`sc_object`	C++ name	0..1	1
`sc_object`	C++ type	1	2
`sc_object`	address	1	4

A module is used as a container of other SystemC objects and can contain: ports, exports, processes, submodules, and channels, see Figure 4.3. There are two types of channels in SystemC. A primitive channel is derived from `sc_prim_channel` and `sc_interface` and can not contain any other SystemC objects. A hierarchical channel is derived from `sc_module` and `sc_interface`. Because it is a module, a hierarchical channel can contain other SystemC objects such as submodules and processes.

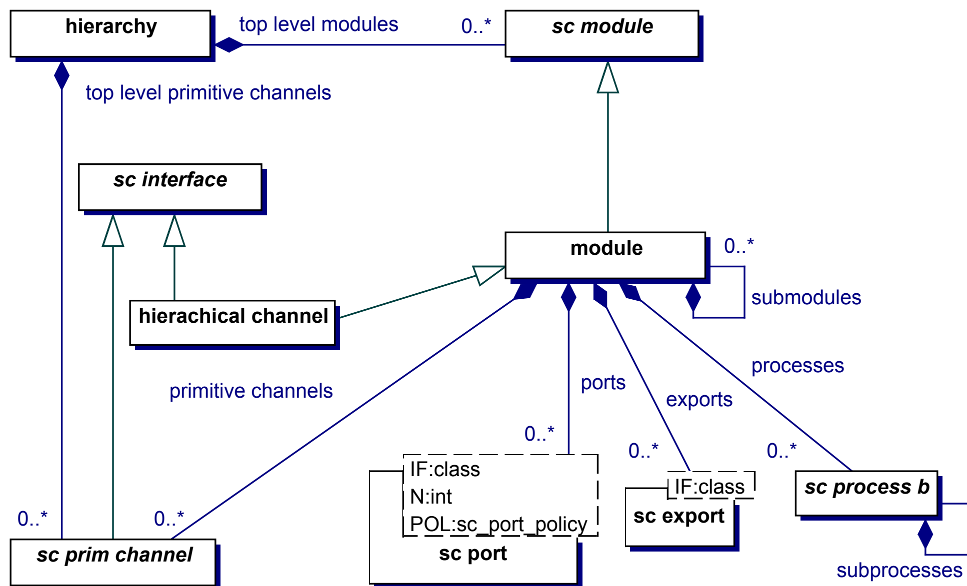


Figure 4.3: A module can contain: channels, ports, exports, processes, and submodules.

Table 4.2: The information which must be retrieved from every SystemC module.

object	information	cardinality	priority
module	ports	0..*	1
module	exports	0..*	1
module	processes	0..*	1
module	primitive channels	0..*	1
module	submodules	0..*	1

Processes can be created inside modules during the elaboration phase of the execution of the model by using the macros `SC_METHOD`, `SC_THREAD`, and/or `SC_CTHREAD`. These processes are called unspawned processes. Unspawned processes are also called static processes because they are created during the elaboration phase. Processes can also be created by calling the `sc_spawn` SystemC library function. This function can be called from the constructor of a module and, in addition, can be called from a process. If `sc_spawn` is called from a process this process contains a subprocess as shown in Figure 4.3. Processes which are created by calling `sc_spawn` are called spawned processes. Spawned processes can be static or dynamic processes. A spawned process which is created during the elaboration phase is called a static process. On the other hand,

a spawned process which is created during the simulation phase is called a dynamic process. The SystemC synthesizable subset only supports unspawned processes, see Table 4.3.

As can be seen in Figure 4.3, modules and primitive channels can be declared outside a module and are in this case called top-level objects. The SystemC module hierarchy consists of these top level objects and of all the objects which are declared inside the top-level modules.

Table 4.3: The process hierarchy and the top-level information which must be retrieved from a SystemC model.

object	information	cardinality	priority
process	subprocess	0..*	3
hierarchy	top-level primitive channels	0..*	1
hierarchy	top-level modules	0..*	1

Figure 4.4 shows the different associations that can be made between the different kind of SystemC objects. An export can be bound to a channel or to an other export. An export must be bound exactly once. A port can be bound to zero or more channels, zero or more ports, and/or zero or more exports. A port can be bound at most N times. This parameter N is provides as the second template parameter of the `sc_port` template. If N is larger than one, then the ports is called a multiport. The different bindings of such a multiport can be selected by using the `operator[]` of the port. If N is zero the port is a multiport which can be bound an arbitrary number of times. The third parameter of the `sc_port` template determines the binding policy of the port. The policy `SC_ONE_OR_MORE_BOUND` means that the port must be bound at least once, the policy `SC_ZERO_OR_MORE_BOUND` means that port binding is optional, and the policy `SC_ALL_BOUND` means that the port must be bound N times. Object Constrain Language (OCL) [86] can be used to describe constrains which apply to UML models. The constrains for Figure 4.4 described in OCL are shown in Figure 4.5.

Each process has a static sensitivity list which is defined during the elaboration phase of the model. A process is sensitive to certain events. An event is something that happens at a specific point in time during the execution phase of the model. During the execution the static sensitivity list can be replaced by a dynamic sensitivity list but this is not relevant for a front-end which only needs to analyze the elaboration phase of the model. A process can be declared to be statically sensitive to an event which is provided by a channel, an export or a port. If a process is made sensitive to an event which is provided by a channel then this event must be declared in the interface which is implemented by the channel. If a process is made sensitive to a channel the event can be explicitly specified. If an event is not explicitly specified then the process is made sensitive to the default event provided by the channel. If a process is made sensitive to an export then the process is made sensitive to the default event of the channel which is bound to the export. If a process is made sensitive to a port an explicit event finder can be used to make the process sensitive to a specific event at the time that the port is bound to a channel. This event finder is needed if the port has not been

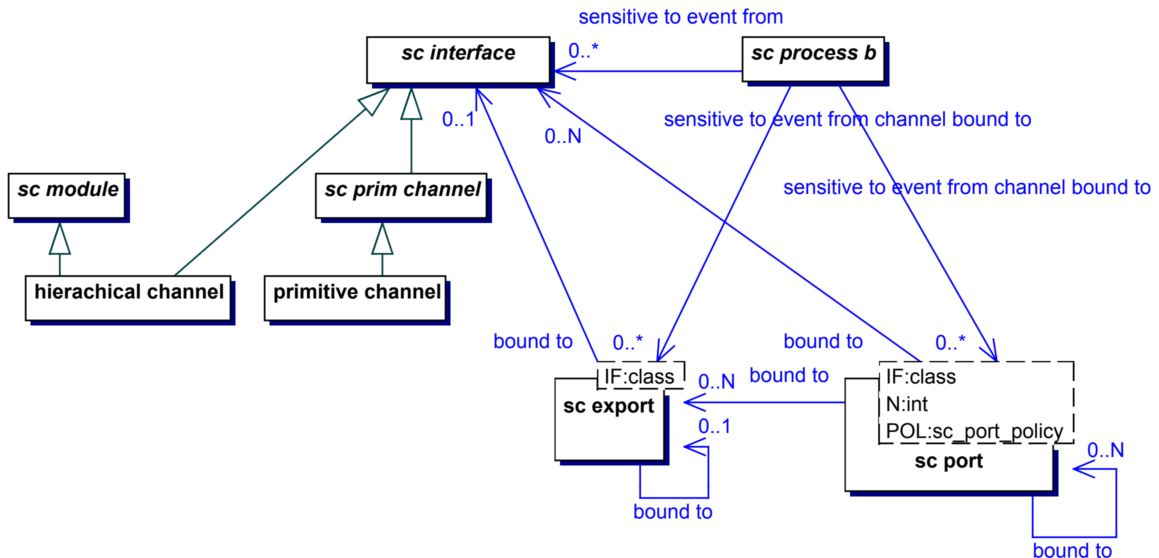


Figure 4.4: The associations between ports, exports, channels, and processes.

```
context sc_export
inv: self.bound to->size=1
context sc_port
inv: POL=SC_ALL_BOUND and self.bound to->size=N or
     POL=SC_SC_ONE_OR_MORE_BOUND and self.bound to->size>=1 and self.
        bound to->size<=N
     POL=SC_SC_ZERO_OR_MORE_BOUND and self.bound to->size<=N
```

Figure 4.5: The constrains for Figure 4.4 expressed in OCL.

bound to a channel at the time that the static sensitivity list is declared. If a process is made sensitive to a port without using an explicit event finder the process will be made sensitive to the default event of the channel instance to which the port is bound

If a process is made sensitive to a multiport then the process will be sensitive to the default events of all ports, exports and channels which are bound to the multiport.

Table 4.4: The associations that must be retrieved from the SystemC module hierarchy.

object	information	cardinality	priority
export	bound to export	0..1	1
export	bound to channel	0..1	1
port	bound to ports	0..N	1
port	bound to exports	0..N	1
port	bound to channels	0..N	1
process	sensitive to events from channels bound to ports	0..*	1
process	sensitive to events from channels bound to exports	0..*	1
process	sensitive to events from channels	0..*	1

There are several primitive channels defined in the SystemC standard as shown in Figure 4.6.

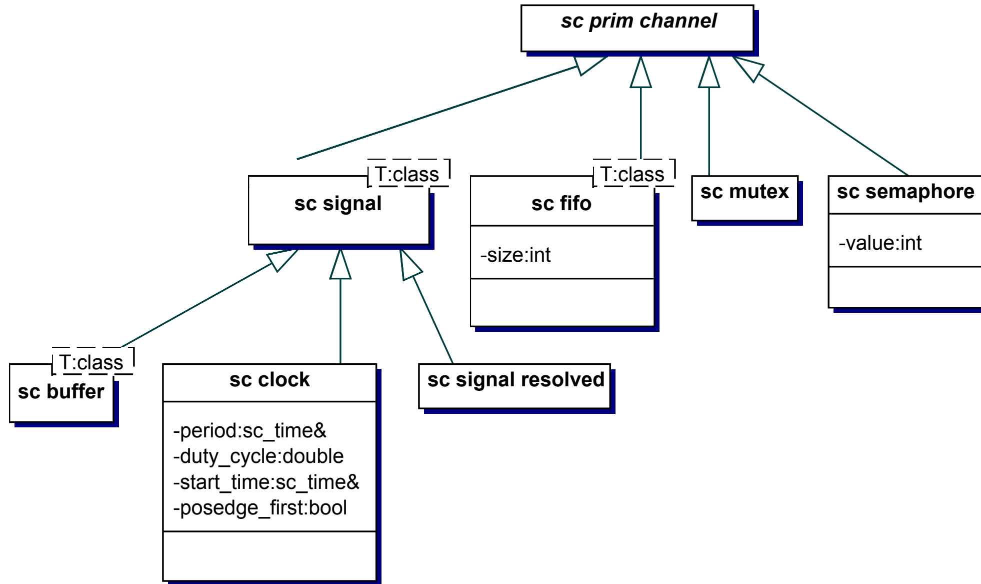


Figure 4.6: The primitive channels which are defined in the SystemC standard.

Table 4.5: The information which must be retrieved from the SystemC primitive channels.

object	information	cardinality	priority
clock	period	1	2
clock	duty cycle	1	2
clock	start time	1	2
clock	posedge first	1	2
fifo	size	1	3
semaphore	value	1	3

The primitive channel `sc_signal` and its derivatives are the only channels which are included in the synthesizable subset of SystemC. A signal can generate a `value_changed_event` which is the default event for a signal. There are template specializations provided for `sc_signal<bool>` and `sc_signal<sc_logic>`. These specialized signals can, in addition, generate a `posedge_event` and a `negedge_event`. An `sc_signal<T>` is derived from the interface class `sc_signal_inout_if<T>` which is derived from the interface classes `sc_signal_in_if` and `sc_signal_write_if`.

Figure 4.7 shows the different kind of ports which are defined in the SystemC standard. The fifo ports are the only once which are not included in the synthesizable subset. Each port requires to be bound to a channel which implements a specific interface. For example, an input port `sc_in<T>` requires to be bound to a channel which implements the interface `sc_signal_in_if<T>`. This interface is implemented by the standard primitive channel `sc_signal`. An `sc_in` provides an event finder `value_changed` which

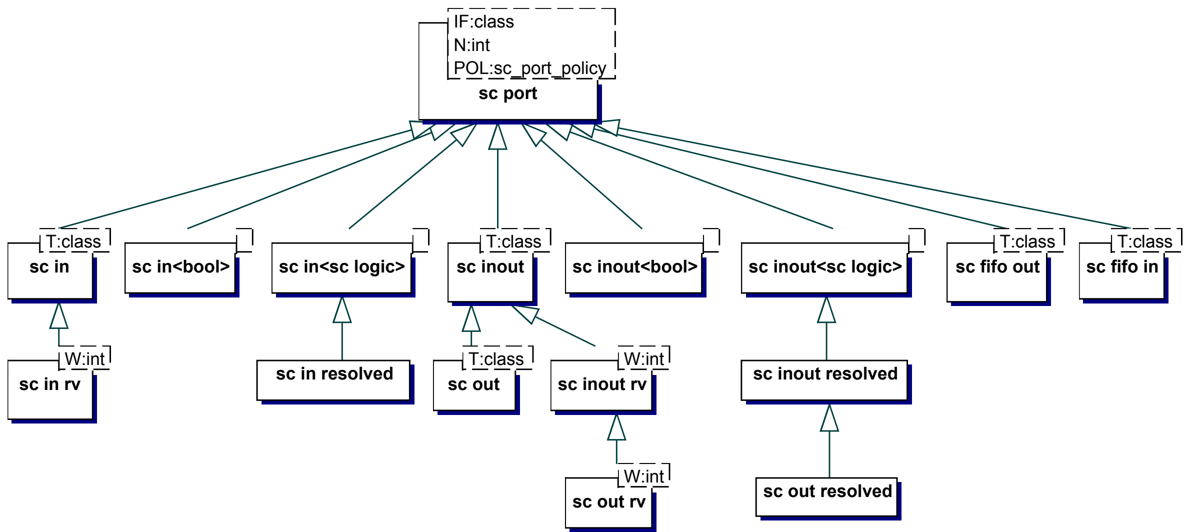


Figure 4.7: The ports which are defined in the SystemC standard.

can be used to find the `value_changed_event` of a channel at the time the input port is bound to this channel. There are template specializations provided for `sc_in<bool>` and `sc_in<sc_logic>`. These specialized inputs provide, in addition, event finders `pos` and a `neg` which can be used to find the `posedge_event` or `negedge_event` respectively. So a SystemC front-end for a SystemC synthesizer must recognize the following events: `value_changed_event`, `posedge_event`, and `negedge_event`.

More details about signals, the interfaces they implement, and the ports to which they can be bound can be found in Appendix B.

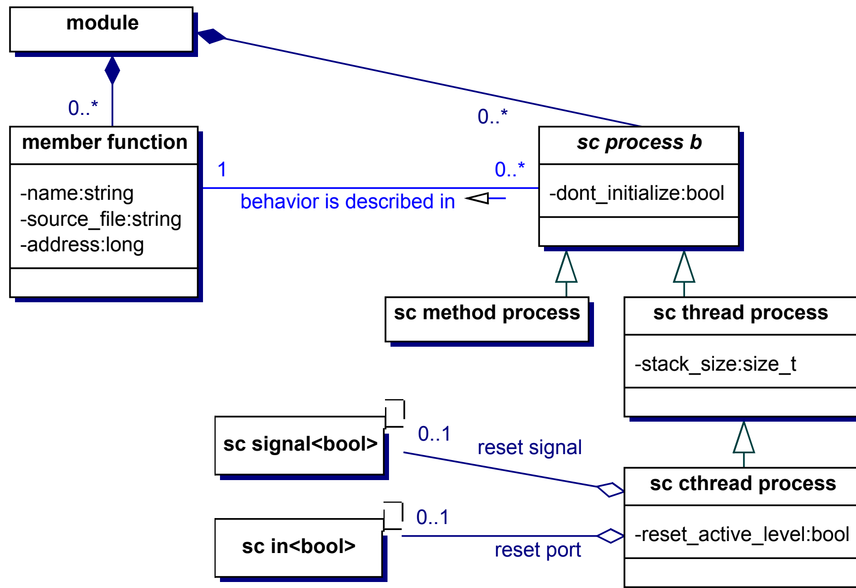


Figure 4.8: The processes which are defined in the SystemC standard.

As already explained in Chapter 2, there are several kind of processes defined in the

SystemC standard, see Figure 4.8. The behavior of a process is defined in a member function of the module in which the process is declared. The qualified name of this member function and the source file in which this member function is defined must be retrieved from the model because this member function must be parsed to retrieve its behavior. The address of this member function can be useful in case the back-end is a debugger. If the variable `dont_initialize` is true the process is not called automatically at the start of the simulation phase but is only called when an event which is on the static sensitivity list of the process occurs. The stack size of a thread process can be set during the elaboration phase. A clocked thread process can have a reset signal or a reset port, but not both. If a reset is defined, the active level of this reset must be specified.

Table 4.6: The information which must be retrieved from each SystemC process.

object	information	cardinality	priority
process	name of the member function which describes the behavior	1	1
process	source file of the member function which describes the behavior	1	1
process	address of the member function which describes the behavior	1	4
process	don't initialize	1	2
thread process	stack size	1	4
clocked thread process	reset signal or reset port	0..1	1
clocked thread process	reset active level	0..1	2

All hierarchical information items which must be retrieved from the SystemC model at the end of the elaboration phase have been identified in this section. An combined list of these information items is presented in Appendix C.

4.4.2 SystemC Function Calls, Overloaded Operators, and Data Types used to Define the Behavior of a Model

The behavior of a SystemC model is described in member functions which are registered with the SystemC simulation kernel. These member functions can use several specific function calls, overloaded operators, and data types which are defined in the SystemC standard. There are restrictions on the C++ language constructions and data types as well as on the SystemC function calls, operators, and data types which can be used if the model must conform to the SystemC synthesizable subset [88]. The SystemC standard [58] has been carefully studied to determine which operations and data types can be used to describe the behavior of a SystemC model. The SystemC operations which can be used on objects within the module hierarchy are listed in Table 4.7 and the SystemC data types and their special operations are listed in Table 4.8 and Table 4.9 respectively. The operations and data types which are included in the synthesizable subset are given priority 1 and the once which are not included in this subset are given priority 2. The return types and all `const` qualifiers are hidden in Table 4.7 to prevent clutter. This

table refers to input and output objects. An input object is instantiated from any of the derived port classes shown in Figure 4.7. An output object is instantiated from any of the port classes shown in Figure 4.7 that have the word "out" in their name. The operations which can be performed on a fifo, mutex, or semaphore are not specified in the table but can be found in [58].

All C++ data types can be used to define the behavior of a SystemC model. If the model must conform to the synthesizable subset then only the integral types (with the exception of `wchar_t`) can be used. A static array of integral type is included in the synthesizable subset. A pointer to an integral type is also included in the synthesizable subset with the restriction that during compilation, synthesis must be able to determine the actual object to which the pointer refers, i.e., the pointee must be statically determinable. Table 4.8 shows the most important data types which are defined in the SystemC standard. A logic variable can be `'0'`, `'1'`, `'Z'` (high-impedance), or `'X'` (unknown) in SystemC. Limited-precision integers are limited to a certain width which is implementation dependent but must be at least 64 bits. The OSCI implementation of SystemC uses 64 bits for this limit. The limited-precision fixed-point data types are implemented by using the C++ `double` data type. The mantissa of this type is limited to 53 bits, so bit-true behavior cannot be guaranteed with the limited-precision fixed point types. These types are meant for fast simulations which do not need bit-true behavior.

All arithmetic, relational, logical, bitwise, and compound-assignment operators are overloaded for the SystemC data-types given in Table 4.8. In addition to these operations some special operation are defined which can be used on objects of these types. These special operations are listed in Table 4.9 The details can be found in [58] and the restrictions for synthesis can be found in [88].

4.5 Intermediate Representation

The SystemC front-end has to convert the SystemC model into an IR which contains the hierarchical and behavioral information described in the model. The preferred format of this IR depends on the purpose of the back-end of the tool for which this front-end is used. Because we want to develop a generic front-end, we have chosen to store all information extracted from the model in an XML document. The XML format is chosen because this format can be easily read and parsed by future tools which use our front-end. To our best knowledge there is no XML markup language for SystemC models defined at the moment. The IP-XACT format [7] can be used to describe the module hierarchy of a SystemC model but can not be used to describe the behavior of the model. The IP-XACT description just refers to files written in some HDL which describe the behavior of the IP. The DotML format is an XML based syntax for the input language of the Dot graph drawing tool from the AT&T GraphViz suite [34]. DotML can be used to describe the behavior of a SystemC module as an AST or CDFG but can not be used to describe the module hierarchy of the model. DotML can not describe a module hierarchy because it does not provide ports or some other form of named connection points. State Chart XML (SCXML) [3] can be used to describe the behavior of a SystemC module as a finite state machine but can not be used to

Table 4.7: The SystemC function calls which must be recognized by a SystemC front-end.

object	operation	description	priority
process	`sc_stop()`	stop the execution of all processes within the model	2
process	`sc_spawn(...)`	create a child process	2
method	`next_trigger(...)`	define the dynamic sensitivity of the method	2
thread	`wait(...)`	define the dynamic sensitivity of the thread	2
cthread	`wait()`	wait for the next active clock edge	1
cthread	`wait(int n)`	wait for the next `n` active clock edges	1
signal	`read()`	read the current value of the signal	1
signal	`operator T&()`	return the current value of the signal	1
signal	`write(T& v)`	write the value `v` to the signal	1
signal	`operator=(T& v)`	write the value `v` to the signal	1
signal	`operator=(signal& s)`	write the value read from `s` to the signal	1
port	`operator->()`	return a pointer to the first channel to which the port is bound	1
port	`operator[](int n)`	return a pointer to the $\mathtt{n}^{\mathrm{th}}$ channel to which the port is bound	2
export	`operator->()`	return a pointer to the channel to which the export is bound	1
input	`read()`	read from the channel to which the input port is bound	1
input	`operator T&()`	read from the channel to which the input port is bound	1
output	`write(T& v)`	write the value `v` to the channel to which the output port is bound	1
output	`operator=(T& v)`	write the value `v` to the channel to which the output port is bound	1
output	`operator=(signal& s)`	write the value read from `s` to the channel to which the output port is bound	1
output	`operator=(port& p)`	write the value read from `p` to the channel to which the output port is bound	1
fifo	`...`	all operations defined for a fifo	2
mutex	`...`	all operations defined for a mutex	2
semaphore	`...`	all operations defined for a semaphore	2

describe the module hierarchy of the model. GraphML [15] can describe hierarchical graphs and also supports ports but the graph editor yEd that uses GraphML does not support ports (last checked for version 3.6). Therefore, we decided to define our own

Table 4.8: The SystemC data types which must be recognized by a SystemC front-end.

data type	description	priority
sc_bit	single bit value, this type is deprecated, use bool instead	1
sc_bv	bit vector	1
sc_logic	single logic value	1
sc_lv	logic vector	1
sc_int, sc_uint	limited-precision integer	1
sc_bigint, sc_biguint	finite-precision integer	1
sc_fix_fast, sc_ufix_fast	limited-precision fixed-point with constructor parameters	2
sc_fixed_fast, sc_ufixed_fast	limited-precision fixed-point with template parameters	2
sc_fix, sc_ufix	finite-precision fixed-point with constructor parameters	2
sc_fixed, sc_ufixed	finite-precision fixed-point with template parameters	1

Table 4.9: The special operations on SystemC data types which must be recognized by a SystemC front-end.

operation	description	priority
operator[int]	bit select	1
range(int, int)	range select	1
operator()(int, int)	range select	1
concat(..., ...)	concatenation	1
operator,(..., ...)	concatenation	1
*xxx*_reduce(...)	reduction operators *xxx* can be and, nand, or, nor, xor, or xnor	1
to_*xxx()*	conversion operators *xxx* can be int, uint, long, ulong, int64, or uint64	1
"..."	conversion from string literal	2

XML format to describe SystemC models. This XML format which we have named SystemC Model Description Language (SCMDL) is based on the SystemC meta-model we have described in Section 4.4.1 and on the GCC AST format [78].

We have chosen to describe the behavior of a SystemC process as an AST because most compiler front-ends also use an AST as an IR. The AST is an exact representation of the original code and algorithms exists to transform an AST into an CDFG [83] or Finite State Machine with Data (FSMD) [96] [50] in case a tool build upon our SystemC font-end needs one of these representations.

A tool build upon our SystemC front-end should be able to easily parse the IR it produces. Therefore, this IR must represent SystemC objects and operations directly. For example, the behavior of the module M shown in Figure 2.3 was described

as: out = c * in; in which out is an output port, in is an input port and c is a constant. The IR for this expression should represent this and nothing more. When a FIR filter is instantiated as given in Figure 2.7, the C++ compiler will expand this expression into something like: operator=(out, sc_fixed<32, 2>(operator*(c, operator sc_fixed<32, 2>(in)));. The operator= is implemented in the C++ library and performs a write on the channel bound to the output port. The conversion operator operator sc_fixed<32, 2> is implemented in the C++ library and performs a read from the channel bound to the input port. The operator* is implemented in the C++ library for objects of the class sc_fxnum which is a base class for the sc_fixed template. This operator* returns an sc_fxval which is converted to an sc_fixed<32, 2> by calling the constructor of this template class. All these implementation details should be hidden in the IR and the expression given as an example should be represented as the AST presented in Figure 4.9.

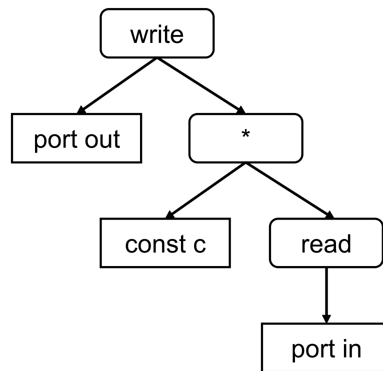


Figure 4.9: The AST for the expression out = c * in, where out is an output port, in is an input port, and c is a constant.

An XML Schema [105] [12] for SCMDL is provided which describes its format and which can be used to validate SCMDL documents. Using eXtensible Stylesheet Language Transformations (XSLT) [27] specific parts of a SystemC model described in SCMDL can be transformed in one of the aforementioned formats.

4.5.1 SystemC Model Description Language (SCMDL)

We have defined SCMDL, an XML format which can be used to describe SystemC models. This format is described in an XML Schema which is presented in Appendix D. An SCMDL document has a root element called <systemc-model>. This element contains a <hierarchy> element which describes the module hierarchy and a <behavior> element which describes the behavior of the model.

The structure which can be used within the <hierarchy> element strictly follows the SystemC metamodel which was defined in Section 4.4.1. The <hierarchy> element can contain zero or more <primitive-channel> elements and zero or more <module> elements which is consistent with Figure 4.3. A <module> element can contain zero or more of the following elements: <port>, <export>, <process>, <primitive-channel>, and/or <module>, which is also consistent with Figure 4.3. Each of these elements

has the required attributes **name**, **systemc-name**, **type**, **systemc-type**, and **address**, which is consistent with Figure 4.2. A **<process>** has a required attribute **function** which specifies the fully qualified function name which describes the behavior of the function. This attribute can be used to find the description of the behavior of this process. A complete description of all elements and attributes which can be used to describe the structure of the SystemC model is not presented here but can be found at http://shabe.sourceforge.net/systemc-model/systemc-model.html. This documentation also contains visual representations of all element types. For example the structure of a **<module>** element is shown in Figure 4.10.

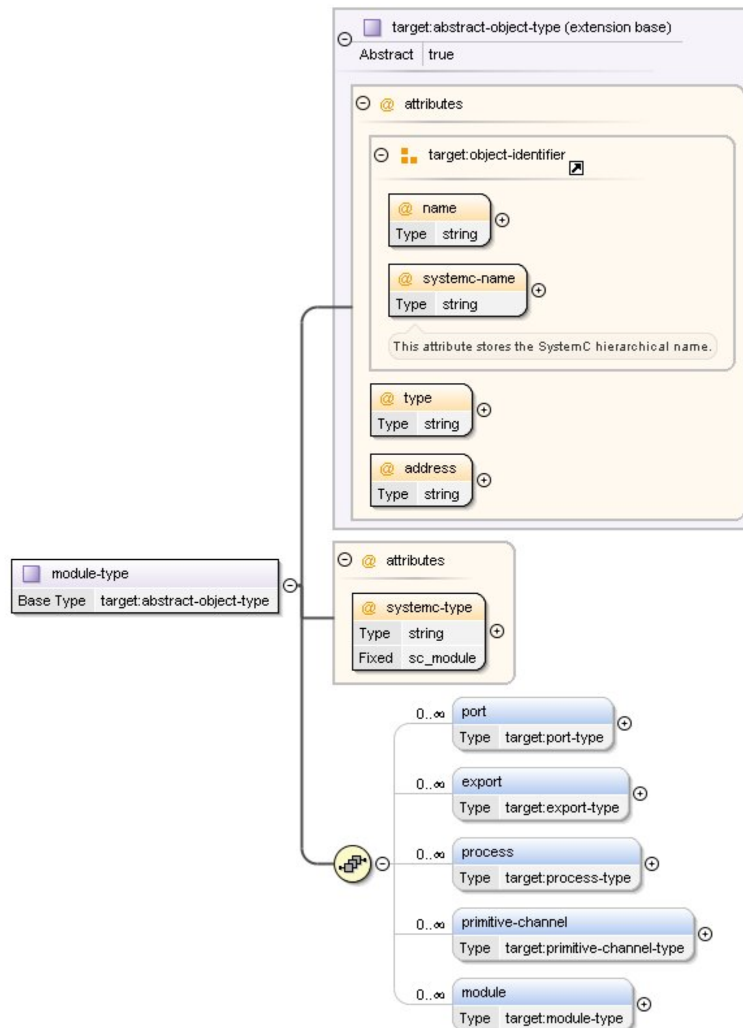


Figure 4.10: The structure of a **<module>** element as defined in the XML Schema.

The **<behavior>** element can contain zero or more **<function>** elements. Each **<function>** element has a required **name** attribute which specifies the fully qualified function name. This attribute can be used to link this behavior with a process which is located inside the module hierarchy. The structure which can be used within the **<function>** element follows the structure of the AST in SSA form used

within the GCC compiler. The data types, function calls, and overloaded operators which are defined in the SystemC library and which can be used to define the behavior of a SystemC model, see Section 4.4.2 are recognized and represented as a simple element. A `<function>` element can contain zero or more of the following elements: `<basic-block>`, `<edge>`, and/or `<condition>`. A `<basic-block>` element can contain `<wait>`, `<read>`, `<write>`, and `<assign>` elements. A complete description of all elements and attributes which can be used to describe the behavior of the SystemC model is not presented here but can be found at `http://shabe.sourceforge.net/systemc-model/systemc-model.html`.

4.5.2 Resource Directory for SCMDL

All elements defined in SCMDL are placed inside the XML namespace http://shabe.sourceforge.net/systemc-model. This makes it possible to combine SCMDL elements with elements from other XML namespaces such as, for example http://www.w3.org/1999/xhtml. An XML namespace is identified by using an Uniform Resource Identifier (URI). This URI does not have to be accessible. We have chosen to place a Resource Directory Description Language (RDDL) [14] document, called a resource directory, at the location of the URI. This resource directory contains a human-readable description and a machine readable link which describes the resources available at the URI. In our case the available resource is the XML Schema for SCMDL documents. This resource directory can be found in Appendix E.

5 Software Development Methodology: Test First

Both the SHaBE program as well as the SHaBEPlugIn are developed without the use of a formal development method. But, during the development of this software several eXtreme Programming (XP) practices [4] have been used. The SHaBE program is developed by the author of this thesis and the first version of the GCC plug-in was developed by Bas van den Aardweg under the supervision of the author. Bas is a Bachelor student of The Hague University which worked full-time on the GCC plug-in as an internship for a period of 3 months in the spring of 2010. The report he wrote (in Dutch) [106] is available at http://shabe.sourceforge.net/.

The XP practices followed during the development of the software are: planning game, small releases, simple design, tests, and refactoring. See [4] for a short description of these practices. The planning game was played between Bas and the author during the development of the GCC plug-in. The SHaBE program and the GCC plug-in were both developed in short iterations. Every iteration implemented a single requirement. Using a test-first approach, a test case was written for each requirement before any implementation code was written. During the development all code was continuously refactored to keep the design simple and the code clean [76]. The SHaBE program which extracts the hierarchical information and the SHaBEPlugIn which extracts the behavioral information were at first developed as separate programs and were only integrated later on.

Because both programs read an input file and produce an output file, testing was fairly simple. For every information item which must be extracted by the program a SystemC input file was written which contained only a single simple test case. An output file with the expected output for this test case was created before any implementation code was written. A simple script was used to compare the actual output of all test programs with their expected output. When the program produced the expected output for the simple test case, the input file was extended with an other test case. The file with the expected output was also extended with the expected output for the new test case, etc. Several test cases for the extraction of the module hierarchy are described in this chapter because these test cases together with their expected output can be seen as the specification for the SystemC front-end. Table C.1 lists all information items for which test cases must be developed.

The SystemC input files used to test the GCC plug-in were not written by the student who implemented this plug-in but by the author of this thesis. The document in which these test cases are described [16] can be found at http://shabe.sourceforge.net/.

This simple test method was extended with unit testing [75] to test the tree data structure which is used in the GCC plug-in and to test the GNU Debugger Machine Interface (GDB/MI) message parser which is used in the main program.

This chapter gives an overview of the 288 test cases which are used during the development of SHaBE. A selection of these test cases are presented here, some can

be found in Appendix F and all others can be found at http://shabe.sourceforge.net/test_programs.

5.1 Test Cases for Signals

A signal is a simple primitive channel which can be declared as a top level object. Therefore very simple SystemC programs which only declare a signal can be written. This is a good starting point for the development of the program which has to extract the module hierarchy from such a SystemC program. The first test case with its expected output in SCMDL is given in Figure 5.1. For this first test we only expect to find the SystemC name and SystemC type of the signal.

```
#include <systemc>
using namespace sc_core;
int sc_main(int argc, char* argv[]) {
    sc_signal<bool> s0;
    sc_start();
    return 0;
}
```

```
<?xml version="1.0" encoding="UTF-8" ?>
<systemc-model xmlns="http://shabe.sourceforge.net/systemc-model"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://shabe.sourceforge.net/systemc-model http://
    shabe.sourceforge.net/systemc-model/systemc-model.xsd" name="signals">
    <hierarchy>
        <primitive-channel systemc-name="signal_0" systemc-type="
            sc_signal" />
    </hierarchy>
    <behavior />
</systemc-model>
```

Figure 5.1: A very simple test case for an sc_signal object and its expected output.

Starting with this simple test case several other variations of signal declarations can be added. For example, Figure 5.2 shows a signal with a user-defined SystemC name The expected output line in SCMDL is also shown. For this second test we also expect to find the C++ name and C++ type of the signal.

```
sc_signal<bool> s1("mySignal");
```

```
<primitive-channel name="s1" type="sc_core::sc_signal&lt;bool&gt;
    " systemc-name="mySignal" systemc-type="sc_signal" />
```

Figure 5.2: A test case for an sc_signal object with a user-defined SystemC name.

When an array of signals is declared the array must be split up into individual elements which are identified by attaching the index operator with the appropriate

index to the array name. We have chosen to handle arrays in this way because the signals in the array can only be used as individual elements when building the module hierarchy and in describing the behavior of the model. The test case for an array of signals is shown in Figure 5.3.

```
sc_signal<bool> a[3];

    <primitive-channel name="a[0]" type="sc_core::sc_signal&lt;bool&
       gt;" systemc-name="signal_0" systemc-type="sc_signal" />
    <primitive-channel name="a[1]" type="sc_core::sc_signal&lt;bool&
       gt;" systemc-name="signal_1" systemc-type="sc_signal" />
    <primitive-channel name="a[2]" type="sc_core::sc_signal&lt;bool&
       gt;" systemc-name="signal_2" systemc-type="sc_signal" />
```

Figure 5.3: A test case for an array of `sc_signal` objects.

Figure 5.4 shows a test case for a multidimensional array of signals.

```
sc_signal<bool> ma[2][2];

    <primitive-channel name="ma[0][0]" type="sc_core::sc_signal&lt;
       bool&gt;" systemc-name="signal_0"  systemc-type="sc_signal" />
    <primitive-channel name="ma[0][1]" type="sc_core::sc_signal&lt;
       bool&gt;" systemc-name="signal_1"  systemc-type="sc_signal" />
    <primitive-channel name="ma[1][0]" type="sc_core::sc_signal&lt;
       bool&gt;" systemc-name="signal_2"  systemc-type="sc_signal" />
    <primitive-channel name="ma[1][1]" type="sc_core::sc_signal&lt;
       bool&gt;" systemc-name="signal_3"  systemc-type="sc_signal" />
```

Figure 5.4: A test case for a multidimensional array of `sc_signal` objects.

A signal can also be dynamically created by using `new`. The address returned by `new` can be stored in a pointer. In this case the C++ name for this object is the name of the pointer prefixed with the dereference operator. We have chosen to handle pointers in this way because the signal to which the pointer is referring can only be used in combination with a dereference operator when building the module hierarchy and in describing the behavior of the model. The test case for a dynamically created signal is shown in Figure 5.5.

```
sc_signal<bool>* p(new sc_signal<bool>("myDynamicSignal"));

    <primitive-channel name="*p" type="sc_core::sc_signal&lt;bool&gt;
       " systemc-name="myDynamicSignal" systemc-type="sc_signal" />
```

Figure 5.5: A test case for a dynamically created `sc_signal` objects.

Figure 5.6 shows a test case for an array of dynamically created signals.

Figure 5.7 shows a test case for a dynamically created array of signals. In this case each individual signal is expected to be identified by the name of the pointer followed

```
sc_signal<bool>* pa[2];
for (int i(0); i<2; ++i)
    pa[i]=new sc_signal<bool>;

    <primitive-channel name="*pa[0]" type="sc_core::sc_signal&lt;bool
        &gt;" systemc-name="signal_0"  systemc-type="sc_signal" />
    <primitive-channel name="*pa[1]" type="sc_core::sc_signal&lt;bool
        &gt;" systemc-name="signal_1"  systemc-type="sc_signal" />
```

Figure 5.6: A test case for an array of dynamically created `sc_signal` objects.

by the index operator with the appropriate index. This is expected because in C++ a pointer to the first element of an array can be used as if it is the name of the array.

```
sc_signal<bool>* pda(new sc_signal<bool>[2]);

    <primitive-channel name="pda[0]" type="sc_core::sc_signal&lt;bool
        &gt;" systemc-name="signal_0"  systemc-type="sc_signal" />
    <primitive-channel name="pda[1]" type="sc_core::sc_signal&lt;bool
        &gt;" systemc-name="signal_1"  systemc-type="sc_signal" />
```

Figure 5.7: A test case for a dynamically created array of `sc_signal` objects.

In all seven test cases for signals discussed so far, the signals are declared as local variables of the function `sc_main`. It is also possible to call a function from `sc_main` and declare all signals as local variables of this function. Of course `sc_start`, see Section 2.1.1, must also be called from this function because otherwise the local objects of the function will be destroyed upon return of the function. Seven more test cases, which are not shown here, have been written to test this.

As can be seen in Figure 5.8 it is possible to declare two signals with the same C++ name. The SystemC name of the signal will always be unique, even if we try to give two object the same SystemC name. During the execution of the elaboration phase of this model the OSCI SystemC implementation reports: "`Warning: (W505) object already exists: mySignal. Latter declaration will be renamed to mySignal_0`".

According to the SystemC standard [58] p. 12, signals may only be created within a module or within function `sc_main`. So it is illegal to define a global signal. But it is legal to store the address of a dynamically created signal in a global pointer as shown in 5.9 or in a global array of pointers which is not shown here.

5.2 Test Cases for Modules

A module can be declared as a top-level object. This means that similar test cases as presented for signals can be written for models. These test cases are not shown here. A module can not have a default constructor [58] p. 31, therefore a module can not be stored in an array. Although, pointers to models can be placed in arrays as

```
void init() {
    sc_signal<bool> s1("mySignal");
    sc_start();
}
int sc_main(int argc, char* argv[]) {
    sc_signal<bool> s1("mySignal");
    init();
    return 0;
}
```

```
<primitive-channel name="s1" type="sc_core::sc_signal&lt;bool&gt;" systemc-name="mySignal" systemc-type="sc_signal" />
<primitive-channel name="s1" type="sc_core::sc_signal&lt;bool&gt;" systemc-name="mySignal_0" systemc-type="sc_signal" />
```

Figure 5.8: A test case in which two `sc_signal` objects get the same C++ name but do not get the same SystemC name.

```
sc_signal<bool>* gp;
int sc_main(int argc, char* argv[]) {
    gp = new sc_signal<bool>;
    sc_start();
    return 0;
}
```

```
<primitive-channel name="*gp" type="sc_core::sc_signal&lt;bool&gt;" systemc-name="signal_0" systemc-type="sc_signal" />
```

Figure 5.9: A test case in which the address of a dynamically created `sc_signal` objects is stored in a global pointer.

shown in Figure 2.6. Modules can be declared inside other modules. The SystemC standard strongly recommends to store submodules as data members of their parent module or to store their addresses in pointer members of their parent module, but it does not require this. Static submodules can only be created in the constructor of their parent module and they must be initialized in the member initialization list of this constructor. Dynamic submodules, which are created with `new`, can be created in the constructor of a module, in a function which is called from the constructor of the module, or from the `before_end_of_elaboration()` callback member function of the module, or in a function which is called from that callback. As explained in Section 2.1.1, the `before_end_of_elaboration()` callback member function of the module is called by the SystemC kernel just before the execution of the elaboration phase is finished. The address of such a dynamically created submodule can be stored in a pointer data member of its parent module, in an array of pointers data member, in a local variable, or not at all. There are $1 + 4 \cdot 4 = 17$ test cases needed to test all possibilities for static and dynamic submodules.

One of these test cases is shown in Figure 5.10. In the module `Module2` there are

two submodules created. Both submodules are dynamically created and the pointers which are returned by `new` are stored in an array of pointers called `subModules`. One submodule is created in the constructor of `Module2` and the other one is created in the `before_end_of_elaboration()` callback member function of this module.

```
SC_MODULE(Module1) {
    SC_CTOR(Module1) {
    }
};
SC_MODULE(Module2) {
    SC_CTOR(Module2) {
        subModules[0] = new Module1("subm0")
    }
private:
    Module1* subModules[2];
    virtual void before_end_of_elaboration() {
        subModules[1] = new Module1("subm1")
};
int sc_main(int argc, char* argv[]) {
    Module2 m2("m2");
    return 0;
}
```

```
<module name="m2" type="Module2" systemc-name="m2" systemc-type="
    sc_module">
    <module name="*subModules[0]" type="Module1" systemc-name="m2
        .subm0" systemc-type="sc_module" />
    <module name="*subModules[1]" type="Module1" systemc-name="m2
        .subm1" systemc-type="sc_module" />
</module>
```

Figure 5.10: One of the test cases for submodules.

If the address of a dynamic submodule is stored in a local variable or if this address is not stored at all, then this object will not have a C++ name at the end of the elaboration phase. A test case is shown in Figure 5.11. As said before, this way of declaring a submodule is not recommended, but is allowed in the SystemC standard.

Modules can be derived from other modules. This means that submodules of a module can also be created and stored in its base modeles. A test case is shown in Figure 5.12 and the UML diagram of this test class is shown in Figure 5.13. Class `Module3` is not directly derived from `sc_module` but it is derived from `Module2` which is derived from `sc_module` by using the macro `SC_MODULE`. An object instantiated from `Module3` contains two submodules both of type `Module1`. One submodule is created in the constructor of `Module3` and is stored in its private data member `m1`. The other submodule is inherited from its base class `Module2`. This base class creates this submodule in its constructor and stores it in its private member `m1`. Please note that both submodules have been given the same name. The SystemC kernel detects this and renames the second submodule which is created. It issues the following message: `Warning: (W505) object already exists: m3.m1. Latter`

```
SC_MODULE(Module1) {
    SC_CTOR(Module1) {
    }
};
SC_MODULE(Module3) {
    SC_CTOR(Module3) {
        new Module1("hidden")
    }
};
int sc_main(int argc, char* argv[]) {
    Module3 m3("m3");
    return 0;
}
```

```
<module name="m3" type="Module3" systemc-name="m3" systemc-type="
    sc_module">
    <module name="" type="Module1" systemc-name="m3.hidden"
        systemc-type="sc_module" />
</module>
```

Figure 5.11: A submodule can be declared without giving it a C++ name.

`declaration will be renamed to m3.m1_0`. As can be seen in Appendix F, Figure F.3 deriving a module from an other module can be a useful coding idiom.

5.3 Test Cases for Ports

Ports can only be created inside a module and every port must be bound to a channel, an export, or an other port. Test cases for the creation of ports are given in Section 5.3.1 and test cases for the binding of ports are given in Section 5.3.2.

5.3.1 Test Cases for the Creation of Ports

A port can be statically created in its parent module. Because a port has a default constructor, it is also possible to declare a static array of ports in a module. It is also possible to dynamically create a port, in the constructor of its parent module, in a member function which is called from the constructor of its parent module, from the `before_end_of_elaboration()` callback member function of its parent module, or in a function which is called from that callback. The address of such a dynamically created port can be stored in a pointer data member of its parent module or in an array of pointers data member. There are $2 + 4 \cdot 2 = 10$ test cases needed to test all possibilities for static and dynamic ports.

A useful implementation for a static array of ports is given in Appendix F, Figure F.1. It is also possible to create ports in the base classes of a module. A test case which utilizes this useful coding idiom can be found in Appendix F, Figure F.3.

```
SC_MODULE(Module1) {
    SC_CTOR(Module1) {
    }
};
SC_MODULE(Module2) {
    SC_CTOR(Module2): m1("m1") {
    }
private:
    Module1 m1;
};
class Module3: public Module2 {
public:
    Module3(const sc_module_name& nm): Module2(nm), m1("m1") {
    }
private:
    Module1 m1;
};
```

```
<module name="m3" type="Module3" systemc-name="m3" systemc-type="
    sc_module">
    <module name="m1" type="Module1" systemc-name="m3.m1" systemc
        -type="sc_module" />
    <module name="m1" type="Module1" systemc-name="m3.m1_0"
        systemc-type="sc_module" />
</module>
```

Figure 5.12: A submodule can be created and stored in a base module.

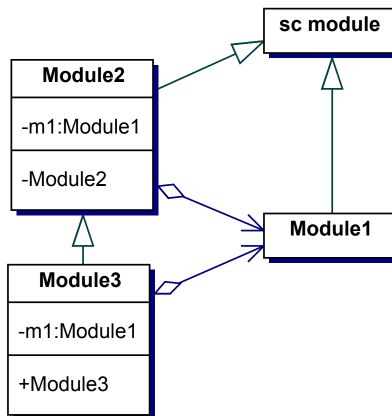


Figure 5.13: The UML diagram for the test case shown in Figure 5.12.

5.3.2 Test Cases for the Binding of Ports

Ports can be bound by name or by position. A port can be bound to a primitive channel, a hierarchical channel, an other port, or an exports. The binding of ports can be performed in the function `sc_main`, in the constructor of a module, in the

`before_end_of_elaboration()` callback member function, or in an other function called from any of these. There are $2 \cdot 4 \cdot 6 = 48$ test cases needed to test all possibilities for port bindings. These test cases are not shown here.

Multiports can be bound more than once. A simple test case for a multiport is shown in Figure 5.14. An instantiation of this module can be found in Figure 5.15. The expected output in SCMDL of the SystemC front-end for this instantiation is also shown.

```
SC_MODULE(And) {
    sc_port<sc_signal_in_if<bool>, 0> in;
    sc_out<bool> out;
    SC_CTOR(And) {
        SC_METHOD(run);
        sensitive << in;
    }
private:
    void run() {
        bool value(in[0]->read());
        for (int i(1); i<in.size(); ++i) {
            value = value && in[i]->read();
        }
        out.write(value);
    }
};
```

Figure 5.14: An `And` gate using an input multiport.

5.4 Test Cases for Exports

Exports can only be created inside a module and every export must be bound to a channel or another export. Exports creation is completely analog with port creation, see Section 5.3.1. Exports can be bound by name only. An export can be bound to a primitive channel, a hierarchical channel, or an other exports. The possible ways to bind an export is a subset of the possible ways to bind a port. Therefore, the test cases for exports are a subset of the test cases for ports. These test cases are not shown here.

5.5 Test Cases for Processes

The second argument of an `SC_CTHREAD` macro call must be an event finder, as explained in Section 4.4.1. We have only written test cases for event finders which are returned by the member functions `value_changed()`, `pos()`, and `neg()` member functions of ports. The `pos()` and `neg()` member functions are only provided for the port template specializations for types `bool` and `sc_logic`. The port which is used can be a member variable, an array element, or a pointee. The `SC_CTHREAD` macro can be called inside the constructor of a module, in a member function which in called from this constructor, in the `before_end_of_elaboration()` callback member function, or in a function called

```
sc_signal<bool> in0, in1, in2, out;
And and("and");
and.in(in0); and.in(in1); and.in(in2); and.out(out);
```

```xml
<module name="and" type="And" systemc-name="and" systemc-type="
    sc_module">
    <port name="in" type="sc_core::sc_port&lt;
        sc_core::sc_signal_in_if&lt;bool&gt;, 0, (
        sc_core::sc_port_policy)0&gt;" systemc-name="and.port_0"
        systemc-type="sc_port">
        <bound-to to="primitive-channel" name="in0" systemc-name=
            "signal_0" />
        <bound-to to="primitive-channel" name="in1" systemc-name=
            "signal_1" />
        <bound-to to="primitive-channel" name="in2" systemc-name=
            "signal_2" />
    </port>
    <port name="out" type="sc_core::sc_out&lt;bool&gt;" systemc-
        name="and.port_1" systemc-type="sc_out">
        <bound-to to="primitive-channel" name="out" />
    </port>
</module>
```

Figure 5.15: An instantiation of the module `And` which is declared in Figure 5.14.

from this callback. There are $3 \cdot 3 \cdot 4 = 36$ test cases needed to test the sensitivity of an `SC_CTHREAD`. These test cases are not shown here. In Appendix F, Figure F.3 an `SC_CTHREAD` macro call is used, The expected output in SCMDL is shown in Figure F.5
.

The static sensitivity of an `SC_METHOD` or an `SC_THREAD` is set by using a sensitivity list. In this sensitivity list the following objects can be used: signals, ports, event finders, exports and events. We have limited ourself to the events which are returned by the `value_changed_event()`, `posedge_event()`, and `negedge_event()` member functions which are provided by signals and the event finders which are returned by the member functions `value_changed()`, `pos()`, and `neg()` member functions of ports. Sensitivity lists can be defined inside the constructor of a module, in a member function which in called from this constructor, in the `before_end_of_elaboration()` callback member function, or in a function called from this callback. The object which is used in the sensitivity list, or which is used to call a member function which returns an event or an event finder, can be a member variable, an array element, or a pointee. Therefore, there are $(1 + 1 + 3 + 1 + 3) \cdot 4 \cdot 3 = 108$ test cases needed to test all combinations. In this case we only used a subset of all possible test cases. These test cases are not shown here. In Figure F.2 an `SC_METHOD` macro call which is sensitive to several ports and the expected output in SCMDL is shown.

A reset signal or reset port can be defined for an `SC_CTHREAD` process, as can be seen in Figure 4.8. A sample test case is shown in Figure 5.16.

```
SC_MODULE(D) {
    sc_in<bool> clk, reset, in;
    sc_out<bool> out;
    SC_CTOR(D) {
        SC_CTHREAD(run, clk.pos());
        reset_signal_is(reset, true);
    }
private:
    void run();
};
```

```
<module name="d" type="D" systemc-name="d" systemc-type="
    sc_module">
    <port name="clk" type="sc_core::sc_in&lt;bool&gt;" systemc-
        name="d.port_0" systemc-type="sc_in" />
    <port name="reset" type="sc_core::sc_in&lt;bool&gt;" systemc-
        name="d.port_1" systemc-type="sc_in" />
    <port name="in" type="sc_core::sc_in&lt;bool&gt;" systemc-
        name="d.port_2" systemc-type="sc_in" />
    <port name="out" type="sc_core::sc_out&lt;bool&gt;" systemc-
        name="d.port_3" systemc-type="sc_out" />
    <process name="run" type="D::run()" systemc-name="d.run"
        systemc-type="sc_cthread" function="D::run">
        <sensitive-to to="port" name="clk" systemc-name="d.port_0
            " event="positive-edge" />
        <reset-to to="port" name="reset" systemc-name="d.port_1"
            active-level="high" />
    </process>
</module>
```

Figure 5.16: A module with an SC_CTHREAD with an active high reset input port.

6 Systemc Hierarchy and Behavior Extractor (SHaBE)

Our approach to develop a SystemC front-end was introduced in Section 4.3. SHaBE will retrieve the hierarchical and behavioral information from the SystemC model in three steps, see Figure 4.1.

In the first step the hierarchical information is retrieved by executing the elaboration phase of the SystemC model under the control of the debugger GDB. This will reveal the module hierarchy of the SystemC model including compile time information like, for example the C++ variable name used for a SystemC object. Also the complete C++ type of a SystemC object is revealed. For example, the template parameter `T` used in an `sc_in<T>` input port of a SystemC module is found. The C++ member functions used to define the behavior of the SystemC processes are also identified during this first step. The location of the source files in which these member functions are defined are available in the debug information.

These source files are analyzed in the second step of SHaBE. In this second step GCC is called with a special plug-in we have developed and named SHaBEPlugIn. This plug-in transforms the AST in SSA [31] form produced by GCC into the SCMDL format defined in Section 4.4.1. GCC creates a separate AST for each function so we provide the plug-in with a list of member functions which must be analyzed. The plug-in recognizes calls to the SystemC library such as `read`, `write` and `wait` and also recognizes the specific SystemC data types such as `sc_logic` and `sc_fixed`. Operations on these specific types are also recognized.

In the last step of SHaBE the IRs of all analyzed member functions are combined with, and linked with, the information found in the first step. Finally an SCMDL document is produced which contains all hierarchical and behavioral information from the SystemC model. This IR can be used as input for a variety of SystemC tools. In Section 6.1, 6.2, and 6.3 the three steps of SHaBE are further explained.

6.1 Extracting the Dynamically Generated Module Hierarchy

SHaBE uses GDB to retrieve the module hierarchy from a SystemC model by running and monitoring the elaboration phase of the executable model. During this execution SHaBE builds up an internal data structure which represents the module hierarchy of the model. This is achieved by using GDB commands to set breakpoints, continue the execution, inspecting the stack, inspecting function arguments, inspecting the members of an object, etc. The source code of the SystemC library and kernel has been carefully analyzed to determine the function calls which need to be monitored. Section 6.1.1 explains how SHaBE communicates with GDB. How SHaBE extracts the information items listed in Table C.1 is explained in Section 6.1.2 to 6.1.9. This information is important for readers who want to extend, or modify our open source implementation

of SHaBE. Readers who are not interested in these implementation details may skip these sections.

6.1.1 Communicating with GDB

SHaBE executes the elaboration phase of the SystemC model under the control of GDB. This debugger has a special interface called GDB/MI which is specifically intended to support the development of systems which use GDB as a component. GDB/MI is a line based machine oriented text interface which can be used to send commands to GDB. The debugger will reply to these commands with lines of text which are formatted in a specific way and are named GDB/MI output records. GDB will also send status messages informing the application which uses the GDB/MI about important events. The syntax of all possible output records is specified in Bachus Naur Form (BNF) in the GDB user manual [101].

We tried to reuse an existing parser for GDB/MI output records. Some open-source debuggers and IDEs which use GDB/MI were investigated. The IDE RHIDE and the debugger DDD both use the Command Line Interface (CLI) of GDB instead of the GDB/MI. The Emacs editor uses the GDB/MI and its parser for output records is written in Lisp. The Eclipse IDE also uses the GDB/MI and its parser for output records is written in Java. The IDEs Qt Creator and KDevelop both use the GDB/MI and they both have a parser for output records written in C++. Because we have chosen to implement SHaBE in C++ the last two parsers where further investigated. The parser from Qt Creator uses Qt specific data types such as `QByteArray`, `QList`, `QVariant`, and `QTextStream`. The parser from KDevelop uses KDevelop specific data types such as `QString`, `QValueList`, `QMap`, and `QMemArray`. Because we do not want to depend on these specific data types and because the syntax for output records is quite simple (only 22 production rules) we decided to develop our own parser. This parser only uses standard C++ data types like `string`, and `vector`. The implementation of this parser is not further discussed in this report, its UML class diagram can be found in Appendix G.

SHaBE starts GDB in a separate process, and uses two pipes to communicate with it. GDB controls the input and output of the executable SystemC model as shown in Figure 6.1.

A complication is the fact that the executable SystemC model which is executed by GDB can also produce output which is intertwined with the output records which are produced by GDB. The executable program can also read input from its standard input stream. Therefore, SHaBE uses multiplexed input when the executable model is running by calling the system call `select` to wait for input from GDB or from the user. When user input is received it is send to GDB which will send it to the executable model which is running. When SHaBE receives input from GDB which can not be parsed as an output record, the output is send to the output stream of SHaBE, normally a console window. This approach has two minor problems. If a SystemC program produces output which can be parsed as a GDB output record then SHaBE will be confused. But this is unlikely to happen by accident because the GDB output records have a specific structure. The second minor problem is caused by the fact that

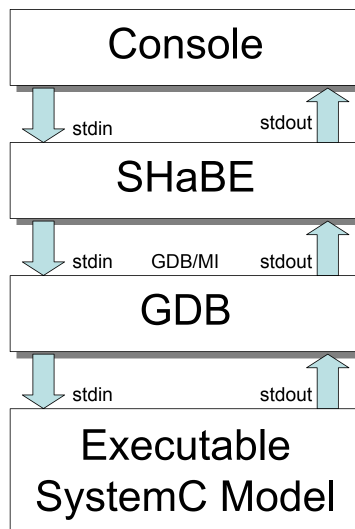


Figure 6.1: The communication between SHaBE, GDB, and the executable SystemC model.

the output of the SystemC program is only received by SHaBE and presented to the user line by line. The well known IDE Eclipse suffers from the same minor problems.

6.1.2 Finding the SystemC Name

Every object which is a member of a SystemC model's module hierarchy is derived from the base class `sc_object`. Therefore it is possible to record the creation of all `sc_objects` by setting a breakpoint on the constructors of `sc_object`. The class `sc_object` has two constructors `sc_object()` and `sc_object(const char* nm)`. Both constructors call the private member function `sc_object_init`, so a breakpoint on this function will catch the creation of all `sc_objects`. Because the SystemC name for the object is determined in the beginning of this function and this name must be retrieved, the breakpoint is not set at this function. At the end of this function the function `sc_simcontext::add_child_object` is called for a top level `sc_object`, the function `sc_module::add_child_object` is called for an `sc_object` which is located inside an `sc_module`, and the function `sc_process_b::add_child_object` is called for an `sc_object` which is located inside an `sc_process`. When breakpoints are set on these functions the `sc_object` which is being created can be inspected via the `object_` function argument and the enclosing object (module or process) can be inspected by the `this` function argument. Via the `object_` function argument the SystemC name of the `sc_object` can be found.

6.1.3 Finding the SystemC Type

To find the most derived SystemC type of the `sc_object` the stack is inspected. The sequence of function calls leading to the breakpoint is traced back until a user-defined function or `sc_main` is found. The function called from there will be the most derived SystemC constructor, for example `sc_module`, `sc_in`,

`sc_out`, or `sc_signal`. For an `sc_object` which represents a SystemC process there is no constructor found using the just described procedure. Instead one of the member functions: `create_method_process`, `create_thread_process`, or `create_cthread_process` from class `sc_simcontext` is found. This fact is used to identify the three different kinds of SystemC processes.

6.1.4 Finding the C++ Type

When the constructor of the most derived SystemC type is found then the type of the `this` pointer will reveal the complete C++ type of this object. For example, when the output port `result` from the `fir` filter which is instantiated in Figure 2.7 is created then the type of the `this` pointer of the most derived SystemC constructor, which is `sc_out`, is reported by GDB as: `class sc_core::sc_out<sc_dt::sc_fixed<32, 2, SC_TRN, SC_WRAP, 0> >`. In this way SHaBE can retrieve the template parameters used in the instantiation of the object.

If the `sc_object` which is being created has an `sc_module` as its most derived SystemC type then the most derived C++ type of this `sc_module` must be found. This is necessary because other SystemC objects, such as ports and channels, can be stored inside these user-defined modules. The most derived C++ type is found by tracing the sequence of function calls further back. The trace is continued as long as a function is found which is a member function and therefore does have a `this` function argument and as long as the `this` function argument equals the address of the `sc_object` which is being inspected. Using this procedure the most derived C++ type for the SystemC module is found. Because a SystemC submodule can also be constructed from within the `before_end_of_elaboration` callback which is called by the SystemC kernel just before the end of the elaboration phase the sequence of function calls is traced further back to see if this `sc_module` constructor is called from this callback. This fact is recorded for later use.

When the constructor of the most derived C++ type is found then the type of the `this` pointer will reveal this type. For example, when the `fir` filter which is instantiated in Figure 2.7 is created then the type of the `this` pointer of the most derived C++ constructor is reported by GDB as: `class fir<sc_dt::sc_fixed<32, 2, SC_TRN, SC_WRAP, 0>, 5>`. In this way SHaBE can retrieve the template parameters used in the SystemC model.

6.1.5 Finding the C++ Name

The C++ names of all channels, ports, and exports within the SystemC module hierarchy must be found because these C++ names will be used in the member functions which are used to describe the behavior of the SystemC modules in the SystemC processes. These member functions will be analyzed in the second step of SHaBE and this will reveal for example a `write` to a SystemC output port named `out`. So the C++ name of this SystemC output port must be found in the first step of SHaBE because otherwise the hierarchical information and the behavioral information for this port can not be linked together. A complication is the fact that the channels, ports, and exports declared within a SystemC module can also be stored in a pointer or array variable.

To find the most derived C++ type of a SystemC module we have to set a breakpoint and trace back the sequence of function calls as described above. Because the base class `sc_object` is constructed before the derived class it is not possible to inspect the data members of the derived class. It is possible though, by using GDB, to find the names and addresses of all data members declared in the derived class (and all off its base classes). This is possible because in C++ the memory for an object is allocated before the constructor for this object is called. For plain data members the name and address is stored in a look-up table inside SHaBE. This table can be used to find the name of an `sc_object`, using its address, when it is created later on.

If the data member is an array each element of the array is treated as a separate data member. If for example the name of the array is `portArray` then the elements are treated as separate variables named `portArray[0]`, `portArray[1]`, etc. These array elements are processed again to facilitate multidimensional arrays and arrays filled with pointers.

If the data element is a pointer a problem occurs. It is not possible to determine the pointee (i.e. the object to which this pointer is pointing). The pointer is not assigned a value yet. This value will probably be assigned inside the constructor of the SystemC module which is not called yet. Assigning a value to the pointer can even be postponed until the `before_end_of_elaboration` callback. This problem is solved as follows. Using GDB a watchpoint is set for each pointer data member of the module under investigation. If this watchpoint is triggered later on, the address assigned to the pointer is stored in the look-up table together with the name of the pointer preceded by a the dereference operator, i.e., a "`*`". So, if a SystemC input port is stored in a data member pointer named `pin` then the name stored in the lookup table will be `*pin`.

In C++ the name of an array is equivalent to a constant pointer to the first element of the array. This enables some weird syntax. The second element of an array named `a` can for example be accessed with the expressions `a[1]`, `1[a]`, and `*(a+1)`. In a similar way a pointer variable can be used in C++ as if it was declared as an array. For example the object pointed to by a pointer `p` can be accessed with the expressions `*p`, `p[0]`, and `0[p]`. At the moment SHaBE assumes that an array element is accessed by using a subscript and that a pointee is accessed by dereferencing the pointer.

Watchpoints in GDB can be hardware watchpoints or software watchpoints. A hardware watchpoint uses specific hardware in the target platform to implement the watchpoint. The SystemC model can be run at full speed when hardware watchpoints are set. A software watchpoint is implemented in GDB by executing the SystemC model step by step and checking the variable that is being watched after each step. Therefore the execution of the SystemC model will be very slow when software watchpoints are set. The number of available hardware watchpoints depends on the platform on which SHaBE is executed. To keep the performance of SHaBE acceptable only hardware watchpoints are used. When the number of pointers to watch exceeds the number of available hardware watchpoints some pointers will not be watched directly. When a watchpoint is triggered all pointers which need to be watched are inspected. As soon as a pointer is assigned it is marked as being updated and the available hardware watchpoints are used to watch pointers which are not updated yet.

A further complication is the fact that a module can contain submodules. The

submodule is called the child module and the surrounding module is called the parent module. The constructor of the child module can be called somewhere during the constructor of the parent module. If the child module also has pointer data members then the watchpoints for the parent module must be disabled and other watchpoints must be set for the child module. When the watchpoints for the child module get out of scope this is reported by GDB. SHaBE will then enable the watchpoints for the parent module again. A child module can also have submodules etc. To facilitate this SHaBE keeps a stack of pointer groups. Each group consists of the pointer data members of a specific module. If a module constructor is analyzed by SHaBE the watchpoints for all pointers in the group currently on the top of the stack are disabled and a new group of pointers is placed on this stack and the watchpoints for these pointers are enabled. As a watchpoint is reported to be out of scope all currently enabled watchpoints are deleted, the group of pointers is removed from the stack, and the watchpoints for the group of pointers currently on top of the stack are enabled.

If an `sc_object` is created in the `before_end_of_elaboration` callback of a SystemC module then this module is inspected for array and pointer data members. Array data members are split into separate data members as described above. If one or more pointers are found all pointers are read and stored in the look-up table as described above. Watchpoints are set to catch any assignment to these pointers during the execution of the callback in the same way as described above.

There are two exceptional situations in which SHaBE is not able to retrieve the C++ names of channels, ports, and exports. The first problem occurs when a pointer is used which points to an array of channels, ports or exports. An example of such a situation is shown in the test case which can be found in Figure 5.6. This problem is caused by the fact that the C++ language does not differentiate between a pointer to a single object and a pointer to the first object of an array. The debugger GDB therefore can not detect the difference between a pointer which points to a single object and a pointer which points to an array of objects. Because SHaBE depends on GDB it has the same limitation.

The second problem occurs when channels, ports or exports are stored in container objects.

If a standard container class is used, the implementation of this container is known, and the debug information is available then the data stored in the container can be found by GDB. Because the way elements of a standard container are accessed is known it is possible in the second step of SHaBE to determine which elements are accessed in the member function which describes the behavioral of the module. For example when input ports are stored in a `std::vector` called `vin` then the expressions `vin[1]` and `*(vin.begin()+1)` will both access the second input port stored in the vector. Storing channels, ports, or exports in a standard container, e.g. `std::vector`, can be useful in a SystemC model. SHaBE does not support the use of standard containers to store channels, ports, or exports at the moment.

If a user-defined container class is used to store channels, ports, or exports it is not possible to find the data stored in the container using GDB because the implementation of the container is unknown. Because the way elements of a user-defined container are accessed is not known, it is impossible to determine in the second step of SHaBE which

elements are accessed by the member function which describes the behavioral of the module. Therefore SHaBE can not support the use of user-defined containers to store channels, ports, or exports.

6.1.6 Finding the C++ Name of a Top-Level Module or Channel

It may be useful for a tool, e.g., a SystemC debugger, which uses SHaBE as its front-end to also know the C++ names of SystemC modules and channels which are not enclosed inside of a module. These ”stand-alone” modules and channels are called top-level objects in SystemC. These top level objects can be statically or dynamically created. A statically created object can be stored as a local variable in `sc_main` or any other function called from `sc_main`. The address of a dynamically created object can be stored in a local variable in `sc_main` or any other function called from `sc_main`, or can be stored in a global variable.

Of course, top-level objects can also be stored in an array or the addresses of top-level objects can be stored in an array of pointers. SHaBE can detect the creation of a top-level object because in this case the breakpoint on `sc_simcontext::add_child_object` is triggered. In this case, the function which calls the most derived constructor is inspected for local variables. For plain local variables the name and address is stored in the look-up table. As explained before, this table will be used to find the name of an `sc_object`, using its address, when it is created later on. Local arrays and pointers are treated in exactly the same way as array and pointer data members.

After the elaboration phase is finished GDB can list the names and types of all global objects. This is a long list which is filtered to find all global objects with a user-defined type or with one of SystemC's predefined channel types, for example `sc_core::sc_signal` or `sc_core::sc_clock`. Array type variables are recursively split into elements to support multidimensional arrays. The address of a plain global variable or plain global array element is retrieved by using GDB and the SystemC module hierarchy is searched for this address. When the address is found the name of the global variable or the subscript expression for the array element is stored as the C++ name of the SystemC object. This means that SHaBE can find the name of globally defined top-level object despite the fact that such object are not allowed by the SystemC standard. SystemC models which define top-level objects globally work without problems when the OSCI SystemC implementation is used.

A global pointer variable or a global pointer array element is recursively dereferenced to support pointers to pointers. The address of the farmost pointee is retrieved by using GDB and the SystemC module hierarchy is searched for this address. When the address is found the dereference expression which is needed to access the object via the pointer is stored as the C++ name of the SystemC object.

Because not all tools which will use SHaBE as a front-end will be interested in the C++ names of globally accessible top-level SystemC objects the retrieval of these names can be conditionally compiled into SHaBE.

6.1.7 Finding the Static Sensitivity of a SystemC Process

The sensitivity of a SystemC process is the set of events that can potentially cause the process to be resumed or triggered. The static sensitivity of an unspawned process is fixed during elaboration. SHaBE only supports unspawned pocesses i.e., processes declared by using SC_METHOD, SC_THREAD or SC_CTHREAD. The static sensitivity is recorded during the first step of SHaBE.

The sensitivity of an SC_CTHREAD is determined by the second argument supplied to this macro. This must be the return value of an event finder. An event finder is a member function of a port class which permits a specific event from the channel to be retrieved through the port instance before the port is bound to a channel. At the time the port is bound the specific event is added to the static sensitivity of the process instance.

The sensitivity of an SC_METHOD and an SC_THREAD is set by using the sensitive data member of the base class sc_module. This data member can only be used as the left operand of an overloaded operator<<. An SC_METHOD or SC_THREAD can be made statically sensitive to zero or more events. On the right side of the operator<< which is used to define the static sensitivity of an SC_METHOD or an SC_THREAD objects of four different SystemC types can be used: sc_event, sc_interface, sc_port_base, or sc_event_finder.

6.1.7.1 Finding the Sensitivity if an sc_event is used in the Sensitivity List

If an object of type sc_event is used the process will be statically sensitive to this event. For example a process can be made sensitive to the rising edge of a sc_logic signal named sig as follows: sensitive<<sig.posedge_event(). At the moment SHaBE can only handle events from the channels sc_signal, sc_buffer and, sc_clock. Both sc_buffer and sc_clock are derived from sc_signal and all three classes are described with the general name signal in this thesis report.

To associate each event with the signal which can notify this event the creation of each event is registered by SHaBE in a lookup table for events. Using GDB a breakpoint is set on the constructor of the class sc_event. When this breakpoint is hit SHaBE will inspect the stack to retrieve the name of the function which called the sc_event constructor. If this function name is default_event or value_changed_event then the event is identified to be of type change. If the name of this function is posedge_event then the event is identified to be of type pos. Finally, if the name of this function is negedge_event then the event is identified to be of type neg. For all these functions, the function argument this points to the sc_signal which can notify the event. The address of the sc_event together with the address of the sc_signal and the type of the event i.e., change, pos, or neg are placed in a lookup table for events.

SHaBE sets a breakpoint on sc_sensitive::operator<<(const sc_core::sc_event&) and when this breakpoint is hit it retrieves the address of the function argument event_. Using this address the signal which will notify the event and the type of event can be found in the lookup table for events. Because the static sensitivity of a process can only be set in the constructor of a module, in the before_end_of_elaboration or end_of_elaboration callbacks of a module, or in a

member function called from the constructor or callback, the data member m_handle from the class sc_module can always be used to identify the process for which the static sensitivity is being defined.

It is not possible to retrieve the events that a signal can notify at the time this signal is constructed because an event is only created and stored inside a signal when some process is sensitive to this event. The functions default_event, value_changed_event, posedge_event, and negedge_event which were mentioned above dynanically create an event when they are called for the first time for a specific signal. This event is then stored inside the signal to be notified during simulation time. If the same function is called more than once for a specific signal the event which was created when the function was called for the first time, is returned.

6.1.7.2 Finding the Sensitivity if an sc_interface is used in the Sensitivity List

If an object derived from the type sc_interface is used on the right side of the operator<< the process will be statically sensitive to the default event of this object. The object derived from sc_interface must be the channel which implements the interface. For example a process can be made sensitive to the default event of the signal named sig as follows: sensitive<<sig. SHaBE sets a breakpoint on sc_sensitive::operator<<(const sc_core::sc_interface&) and when this breakpoint is hit it retrieves the address of the function argument interface_. This address can be used to find the channel which implements this interface. The process for which the static sensitivity is being defined is identified as described above.

Exports are provided by SystemC to export the interface of a channel located inside a module to the part of the model outside that module. A process can be made sensitive to an export. But there is no overloaded sc_sensitive::operator<< available for sc_export objects. A note on page 50 of the SystemC standard [58] explains that the operator<< which is overloaded for objects of type sc_interface can be used instead, because of the existence of the conversion operator sc_export<IF>::operator IF&. This operator is intended for use during elaboration as an implicit conversion when passing an object of class sc_export in a context that requires an sc_interface, for example, when binding a port to an export or when adding an export to the static sensitivity of a process.

The consequence of this specification is that SHaBE is not able to detect that a process is sensitive to an export. SHaBE detects that the process is sensitive to the interface which is exported by the export. Although the effect of making a process sensitive to the interface exported by an export and making a process sensitive to the export itself is completely the same, it is still a pity that this implementation detail of the SystemC model is lost. But, as explained, this is due to the fact that SystemC does not provide a properly overloaded sc_sensitive::operator<< for export objects. It is remarkable inconsequent that the set_sensitivity function which is used to set the sensitivity of spawned processes is overloaded for export objects.

6.1.7.3 Finding the Sensitivity if an sc_port_base or an sc_event_finder is used in the Sensitivity List

If an object of type `sc_port_base` is used on the right side of the `operator<<` the process will be statically sensitive to the default event of the channel which will be bound to the port later on. For example a process can be made sensitive to the default event of an input port named `in` as follows: `sensitive<<in`.

If an object of type `sc_event_finder` is used on the right side of the `operator<<` the process will be statically sensitive to a specific event of the channel which provides the `sc_event_finder`. An event finder is necessary to create static sensitivity to a specific event defined in the channel. For example a process can be made sensitive to the rising edge of an input port named `clk` by using an event finder as follows: `sensitive<<clk.pos()`.

At the time that a process is made statically sensitive to a port the port does not have to be bound to a channel yet. So it is not possible to retrieve the event to which the process will eventually be made sensitive. The process will be made sensitive to this event after the port binding has been completed. If a process is made sensitive to a port eventually the `make_sensitive` member function of the class `sc_port` is called. There are two overloaded versions of this member function one version which is called if an `SC_METHOD` is made sensitive to a port and an other version which is called if an `SC_THREAD` is made sensitive to a port. The latter is also called if an `SC_CTHREAD` is created which is always made sensitive to a port via an event finder.

Using GDB a breakpoint is set on these two functions and the function arguments are inspected. The function argument `this` points to the port to which the process is made sensitive. The function argument `handle_p` identifies the process which is made sensitive to this port. Finally, the function argument `event_finder_` points to the `event_finder` which will be used to find the event later on when this port is bound. If the `event_finder_` argument is zero, then the process will be sensitive to the default event of the channel to which it will be bound later on.

If the `event_finder_` argument is pointing to an `event_finder` object this object must be identified to determine the kind of event the process will be sensitive to. Because SHaBE only supports `sc_signal`, `sc_buffer`, and `sc_clock` channels at the moment, this `event_finder` object must have been returned by a event finder member function of an `sc_in`, `sc_out`, or `sc_inout` port i.e., `value_changed()`, `pos()` or `neg()`. The `event_finder` objects returned by these functions are saved in private data members respectively named `m_change_finder_p`, `m_pos_finder_p`, and `m_neg_finder_p`.

The addresses of these `event_finder` objects can not be retrieved at the time a port is created because the SystemC library only creates a specific `event_finder` object for a port when the event finder member function is called for the first time. If an event finder member function for a specific port is called more than once, the `event_finder` object created during the first call is returned. Every `sc_in`, `sc_out`, and `sc_inout` template instantiation provides the `value_changed()` event finder member function. Only the `sc_in`, `sc_out`, and `sc_inout` template specializations for `sc_logic` and `bool` provide the `pos()` and `neg()` event finder member functions.

The type of the event i.e., `change`, `pos`, or `neg` to which the process will be sensitive after the port is bound can be found by comparing the `event_finder_` function

argument of the make_sensitive member function with the private data members m_change_finder_p, m_pos_finder_p, and m_neg_finder_p. This is not straight forward because the make_sensitive member function is defined in the class sc_port_b which is an abstract base class for class sc_port which is a base class for sc_in, sc_out, and sc_inout. It is impossible to find the data members of the derived class from a pointer to its base class object. The this pointer of the make_sensitive member function points to an object of type sc_port_b.

This problem is solved as follows. Because a process can only be made sensitive to an existing port, the C++ name of the port can be looked up in the part of the module hierarchy which is already extracted from the model. The address of the derived port class will be the same as the address for the sc_port_b object, so the C++ name of the derived port can be found by using this address. Because make_sensitive is called from sc_sensitive::operator<< which is called from the constructor or member function in which the sensitivity for the process is being set the C++ name for the port must be known in the third stack frame found on the stack when the breakpoint on make_sensitive is hit. Using GDB the private data members m_change_finder_p, m_pos_finder_p, and m_neg_finder_p of this variable are retrieved and compared with the event_finder_ function argument of the make_sensitive member function. In this way the type of the event i.e., change, pos, or neg to which the process will be sensitive after the port is bound is found by SHaBE.

In SystemC a port can be declared to be a multiport. A process can be made statically sensitive to a specific event of one specific channel or port which is bound to a multiport. Alternatively, a SystemC process can be made statically sensitive to the default event of all channels and/or ports which are bound to a multiport. SHaBE fully supports multiports.

6.1.8 Finding the Connections Between the Modules

In SystemC modules are connected with one another via channels using the ports and exports of the modules. Ports can be bound to channels, to other port instances, or to export instances. Exports can be bound to channels or to other export instances, but not to port instances. Ports can be bound by name or by position, but exports can only be bound by name. A port can be declared as a multiport. A single multiport can be bound to multiple channel or ports. An export can only be bound once.

SHaBE can reveal all these bindings by setting breakpoints on appropriate functions in the SystemC library and by analyzing the function arguments when these breakpoints are hit. The binding of a port to another port is found by setting a breakpoint on the member function bind(sc_port_base&) from class sc_port_base. When this breakpoint is hit, the function argument this reveals the address of the port being bound and the function argument parent_ reveals the port to which the port pointed to by this is bound. The binding of a port to a channel is found by setting a breakpoint on the member function bind(sc_interface&) from class sc_port_base. When this breakpoint is hit, the function argument this reveals the address of the port being bound and the function argument interface_ reveals the channel to which the port pointed to by this is bound. The actual type of the interface_ function argument is derived from sc_interface. It is the channel which implements the interface.

There is no overloaded bind member function for exports defined in the class sc_port_base. When a port is bound to an export the conversion operator sc_export<IF>::operator IF& is called and the function bind(interface&) is called. The consequence of this implementation is that SHaBE is not able to detect that a port is bound to an export. SHaBE detects that the port is bound to the interface which is exported by the export. Although the effect of binding a port to the interface exported by an export and binding a port to the export itself is completely the same, it is still a pity that this implementation detail of the SystemC model is lost. But, as explained, this is due to the fact that SystemC does not provide a properly overloaded sc_port_base::bind for export objects. This problem is completely analog with the problem of detecting the sensitivity of a process to an export as described in Section 6.1.7.2.

The binding of a export to another export is found by setting a breakpoint on the member function bind(IF&) from class sc_export<IF> as well as setting a breakpoint on the operator()(IF&) from class sc_export<IF>. When one of these breakpoints is hit the function argument this reveals the address of the export being bound and the function argument interface_ reveals the channel to which the export pointed to by this is bound. The actual type of the interface_ function argument is derived from sc_interface. It is the channel which implements the interface.

Again, there is no overloaded bind member function for exports defined in the class sc_export. The consequence of this implementation is that SHaBE is not able to detect that an export is bound to an export. SHaBE detects that the export is bound to the interface which is exported by the (second) export. Although the effect of binding an export to the interface exported by an export and binding a export to the (second) export itself is completely the same, it is again a pity that this implementation detail of the SystemC model is lost.

All these bindings are administered by SHaBE in its internal representation of the SystemC module hierarchy.

6.1.9 Finding the Reset Signal of a SC_CTREAD and its Active Level

In SystemC a reset port or reset signal can be defined for an SC_CTHREAD process. The active level of this reset signal must also be defined. An SC_CTHREAD process is reset when the clock event to which the process is statically sensitive is notified and the reset signal is active. SHaBE can retrieve the reset signal or port and its active level by setting a breakpoint on the appropriate functions in the SystemC library and analyzing the function arguments. The reset signal or reset port and the active level of the reset are administered by SHaBE as an attribute of the SC_CTHREAD description in its internal representation of the SystemC module hierarchy.

6.2 Retrieving the Behavior of the SystemC Modules

The behavior of the SystemC model is extracted by using a GCC plug-in. A GCC plug-in can schedule the execution of specific code in between, or as a replacement of, some passes of the GCC compilation process using the pass manager. We have developed a

plug-in for GCC named SHaBEPlugIn which executes code after the SSA pass of GCC is finished. SHaBEPlugin converts the AST in SSA form which is produced by GCC into SCMDL. As already mentioned in the intoduction of Chapter 5 the first version of SHaBEPlugIn was developed by Bas van den Aardweg under the supervision of the author of this thesis. Many implementation details can be found in the report he wrote (in Dutch) [106].

Section 6.2.1 gives a very short overview of the internal structure of the GCC C++ compiler. The communication between SHaBE, GCC, and SHaBEPlugIn is explained in Section 6.2.2. Some implementation details of SHaBEPlugIn are given in Section 6.2.3.

6.2.1 GCC Internals

The internal structure of GCC is described in [85]. The GCC compiler has three main parts: front-end, middle-end, and back-end. The C++ front-end of GCC produces an AST in the GENERIC format. In this GENERIC format the C++ specific features are explicitly represented. This GENERIC format is transformed into the GIMPLE format [78]. GIMPLE is a simplified GENERIC, in which all constructs are lowered into a simpler form. In transforming the GENERIC tree into GIMPLE, complex expressions are split into a three address code using temporary variables. In the middle-end the GIMPLE format is transformed into SSA form and a large number of powerful language-independent and architecture-independent optimizations are performed on this SSA GIMPLE tree [84]. In the back-end of the GCC compiler the optimized GIMPLE tree is converted into an architecture dependent RTL which is finally converted into assembly code for a specific target architecture. A GIMPLE tree consists of basic blocks which contain GIMPLE instructions. These basic blocks are connected with edges which represent the control flow of the source code. The pass manager of GCC will call each pass once for each function or member function.

6.2.2 Communicating with SHaBEPlugin

The C++ member functions which define the behavior of the SystemC model are already identified during the first step of SHaBE. SHaBE used GCC and SHaBEPlugIn to extract the AST in SSA form from the source codes of these functions. The communication between SHaBE, GCC, and SHaBEPlugIn is shown in Figure 6.2. All communication takes place via command line parameters and by using temporally files.

The locations of the source files in which the member functions that must be analyzed are defined, are extracted from the debug information during the first step of SHaBE. SHaBE creates an internal list of files which must be analyzed and their locations. For every source file which is identified an internal list of fully qualified function names which must be analyzed is produced by SHaBE.

For every source file which must be analyzed SHaBE creates a subdirectory with an unique name. The list of function names located in this specific source file is written into the file `shabeplugin-arguments.xml` inside this directory. This list is formated in a simple XML format. Every function name is presented in this XML document by a `<function>` element. These `<function>` elements are

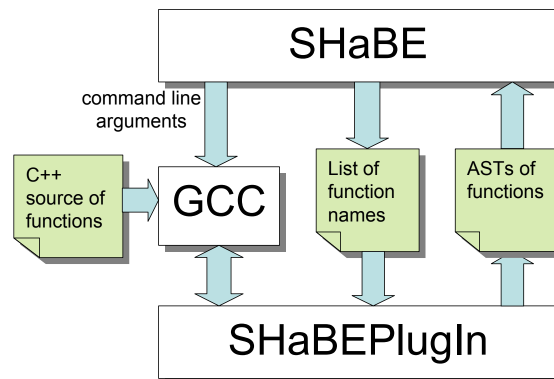


Figure 6.2: The communication between SHaBE, GCC, and SHaBEPlugIn.

enclosed by a `<shabeplugin-arguments>` element. SHaBE starts GCC in a separate process using the following command line: `gcc -fplugin=shabeplugin.so -fplugin-arg-shabeplugin-arguments=shabeplugin-arguments.xml -fno-inline -S -o /dev/null -I`*`path-to-systemc-include filename`*. The command line argument `-fplugin=shabeplugin.so` tells GCC to use the specified plug-in and the argument `-fplugin-arg-...` is used to pass an argument to this plug-in. Because we want to recognize all calls into the SystemC library we do not want GCC to inline these calls. Inlining is prevented by using the command line argument `-fno-inline`. The `-S` command line argument is used to stop GCC after the compilation phase is finished. This prevents GCC from calling the assembler and linker. Because we do not need the output which is produced by GCC we send it to the bit bucket by using the command line argument `-o /dev/null`. The path to the SystemC include file is provided by the `-I...` command line argument. This path is extracted from the debug information included in the executable SystemC model by SHaBE.

When GCC starts, it dynamically links the library `shabeplugin.so` and it calls the `plugin_init` function which must be present in this dynamic library. The argument for this plug-in which are provided on the command line is passed as an argument of this `plugin_init` function. The `plugin_init` function of SHaBEPlugin retrieves this argument and reads the function names from this XML document. Then SHaBEPlugIn calls the `register_callback` function within GCC to register a callback function and returns control to GCC. GCC calls the registered callback function inside SHaBEPlugIn for every function which is compiled just after the SSA AST for that function is created. When this callback function is called, SHaBEPlugIn checks if the function which is being compiled is on the list of functions which must be analyzed. If not, the callback return immediately. Otherwise, SHaBEPlugIn retrieves the AST from GCC and stores it into its own internal structure. When GCC has processed the complete source file it will exit. We have declared a global variable inside SHaBEPlugIn. The destructor of this object is called just before GCC exists. Inside this destructor the AST of all analyzed functions is written into an SCMDL document `shabeplugin-output.xml` inside the temporally directory.

SHaBE waits for the GCC process to finish, reads the information from the

`shabeplugin-output.xml` file, and finally destroys the temporally directory.

6.2.3 SHaBEPlugIn

SHaBEPlugIn executes code after the SSA pass of GCC is finished. At that moment the GENERIC tree is completely converted into GIMPLE and all conditional statements will have both a `then` and an `else` part, but no optimizations are performed yet. We have chosen to use the non optimized GIMPLE tree so the IR is simplified into SSA form but still resembles the source code as close as possible. SHaBEPlugin converts this non optimized GIMPLE tree into an internal tree structure.

The GIMPLE tree is simplified by SHaBEPlugin. Calls to functions and overloaded operators of the SystemC library must be represented as simple nodes in the resulting SCMDL document, see Section 4.5. These functions and overloaded operators are recognized by their fully qualified name i.e. a concatenation of their namespace, class name, and function or operator name. At this moment SHaBEPlugin can only recognize a limited number of functions and operators. All `read`, `write`, and `wait` function calls are recognized. The operators which call `read` or `write` in their implementations are recognized and transformed into an explicit `read` or `write` node in SCMDL. The operators `*`, `+` and `-` are recognized for standard C++ integral and floating types and for the SystemC `sc_fixed` type.

The GIMPLE tree is stored inside SHaBEPlugin in a tree data structure which implements the visitor pattern [39]. Using this pattern separate visitors are written which perform simplifications on this internal tree. The GIMPLE form retrieved from GCC uses a lot of unnecessary temporally variables which are removed by one of the visitors. An other visitor will replace every operator which calls `read` or `write` in its implementation into a `read` or `write` node in the internal tree. Finally this internal tree is stored in an SCMDL document.

A member function which defines the behavior of a process can call other user-defined functions. At this moment these calls appear in the internal tree as call nodes.

6.3 Combining the Hierarchical Information with the Behavioral Information

As described in Section 6.1, SHaBE retrieves the C++ names of all ports and channels in a SystemC model. These ports and channels can be used in the description of the behavior of the model which is analyzed in the second step of SHaBE. Ports are used to access the channels which are bound to these ports during the elaboration phase of the execution of the model. These bindings are also retrieved in the first step of SHaBE. If the channels and ports used in the member functions that describe the behavior of the model are stored in a plain data member, then the name found in the analysis of the member function are the C++ names of these data members. In this case ports and channels found in the second step can be simply linked with the port and channels found in the first step by using the fully qualified C++ names of these ports and channels.

If ports and/or channels are stored inside arrays or accessed via pointers the association of ports and channels between the two steps of SHaBE is a bit more involved.

Arrays of ports and arrays of channels are split into individual objects in the first step of SHaBE. The name of these objects is the expression needed to access such an individual object within the array, for example `a[2]`. When such an array element is accessed in the member function which describes the behavior of a SystemC process then the subscript operator appears in the IR produced by the second step of SHaBE. If the subscript used in the subscript operator is a compile time constant then SHaBE can associate the access with a specific array element found during the first step. In this case the subscript operator can be removed from the behavioral IR produced by the second step of SHaBE. But, if the subscript used in the subscript operator is a variable then the subscript operator models a multiplexer or demultiplexer. A multiplexer is modeled if the subscript operator is used as an rvalue and a demultiplexer is modeled if the subscript operator is used as an lvalue. In this case, the array access found in the second step of SHaBE will be associated with all elements of the array found during the first step of SHaBE. Which of these elements is selected during run-time depends on the values of run-time variables and this behavior is captured in the IR produced by the second step of SHaBE. Figure F.1 shows the SystemC model of a simple N:1 multiplexer.

If a port of channel is accessed via a pointer then the name of this object found in the first step of SHaBE is the expression needed to access the object, for example `*p`. When such a pointee is accessed in the member function which describes the behavior of a SystemC process then the dereference operator appears in the IR produced by the second step of SHaBE. If the dereference operator is applied to a pointer variable which was identified by SHaBE in step one, then SHaBE can associate the access with a specific object. In this case the dereference operator can be removed from the behavioral IR produced by the second step of SHaBE. But, if the dereference is used on a pointer which is not identified by SHaBE in step one, or if pointer arithmetic is used then the dereference operator models some behavior of the model. In this case the pointer dereference found in the second step of SHaBE can not be associated with an object found during the first step of SHaBE and the dereference behavior is captured in the IR produced by the second step of SHaBE.

A tool which is build upon SHaBE can replace reads and writes to ports with reads and writes to the channels which are bound to these ports. This makes it possible to construct the complete CDFG of the model.

7 Results

Due to the limited time available to us SHaBE is not fully implemented. The first section of this chapter gives an overview of what is implemented and what is not. The implementation of SHaBE is build upon GCC, GDB and the OSCI SystemC library. This creates certain dependencies which are listed in this chapter, see Section 7.2. The test results of using SHaBE are also presented in this chapter. In Section 7.4 the theoretically determined run-time complexity is compared with the actual execution time of SHaBE. During the time we were developing SHaBE an other SystemC front-end called PinaVM was introduced, see Section 3.1.3.3. In Appendix K SHaBE is compared with PinaVM.

7.1 Implementation Status

Almost all of the hierarchical information which must be retrieved from a SystemC model, see Appendix C is extracted by SHaBE. The hierarchical information which actually can be extracted from a SystemC model at the end of the elaboration phase by SHaBE is reported in Table 7.1. The information items are sorted by priority. The priority values used are explained in Chapter 4. An information item which is extracted by SHaBE is marked in the column SHaBE in Table 7.1 with the symbol $\surd$. The symbol $\varnothing$ is used to mark information items which can not be extracted due to limitations in the SystemC standard. Information items which are currently not extracted by SHaBE due to the limited time available to us are marked with the symbol ⌚. These information items can be extracted in the future in a similar manner as other information items which are already successfully extracted. All information items which were given the highest priority in Section 4.4 and which can be extracted are in fact successfully extracted by SHaBE.

It is not possible to extract the binding to an export and the sensitivity to an export due to limitations in the SystemC standard. Section 6.1.7.2 explains why it is not possible to detect that a process is sensitive to an export. In this case SHaBE extracts the information that the process is sensitive to the interface which is exported by the export. This has no consequences for the behavior of the model but an implementation detail of the SystemC model is lost. As explained in Section 6.1.8 it is not possible to detect that a port is bound to an export and it is also not possible to detect that an export is bound to another export. In these cases SHaBE extracts the information that the port or export is bound to the interface which is exported by the (second) export. This has no consequences for the behavior of the model but an implementation detail of the SystemC model is lost.

As explained in Section 6.1.5, SHaBE is not able to find the names of objects which are accessed via a pointer to an array of objects. This is caused by the limitation

Table 7.1: The hierarchical information which can, and which can not, be extracted by SHaBE.

object	information	cardinality	priority	SHaBE
`sc_object`	SystemC name	1	1	√
`sc_object`	SystemC type	1	1	√
`sc_object`	C++ name	0..1	1	√
module	ports	0..*	1	√
module	exports	0..*	1	√
module	processes	0..*	1	√
module	primitive channels	0..*	1	√
module	submodules	0..*	1	√
hierarchy	top-level primitive channels	0..*	1	√
hierarchy	top-level modules	0..*	1	√
export	bound to export	0..1	1	∅
export	bound to channel	0..1	1	√
port	bound to ports	0..N	1	√
port	bound to exports	0..N	1	∅
port	bound to channels	0..N	1	√
process	sensitive to event from channels bound to ports	0..*	1	√
process	sensitive to event from channel bound to exports	0..*	1	∅
process	sensitive to event from channels	0..*	1	√
process	name of the member function which describes the behavior	1	1	√
process	source file of the member function which describes the behavior	1	1	√
clocked thread process	reset signal or reset port	0..1	1	√
`sc_object`	C++ type	1	2	√
clock	period	1	2	◷
clock	duty cycle	1	2	◷
clock	start time	1	2	◷
clock	posedge first	1	2	◷
process	don't initialize	1	2	◷
clocked thread process	reset active level	0..1	2	√
process	subprocess	0..*	3	◷
fifo	size	1	3	◷
semaphore	value	1	3	◷
`sc_object`	address	1	4	√
process	address of the member function which describes the behavior	1	4	√
thread process	stack size	1	4	◷

of the C++ language to differentiate between a pointer to an array of objects and a pointer to a single object. Currently, SHaBE is not able to detect the names of objects which are stored in standard or user-defined containers. The detection of the names of objects which are stored in standard containers is not implemented due to time limitations. We estimate this to be at least a man-month of work due to the amount of standard containers which are available and the various ways in which the elements stored in those containers can be accessed. The detection of the names of objects which are stored in user-defined containers is not possible due to the fact that the way these objects are stored and can be accessed is unknown. More details were given in Section 6.1.5.

Most of the information items which were given a lower priority in Section 4.4 can not be extracted by SHaBE at the moment. These information items can be extracted in the future in a similar manner as other information items which are already successfully extracted. The extraction of all information items which are currently not extracted, because of time limitations, can be fairly easy implemented. We estimate that this work can be performed in a few man-weeks.

Only a limited amount of the behavioral information which can be retrieved from a SystemC model, see Section 4.4.2, is actually extracted by SHaBE at the moment. The first version of SHaBEPlugIn which was implemented by Bas van den Aardweg recognizes all `read`, `write`, and `wait` function calls into the SystemC kernel. The overloaded operators which call `read` or `write` in their implementations are recognized and transformed into an explicit `read` or `write` node in SCMDL. The operators `*`, `+` and `-` are recognized for standard C++ integral and floating types and for the SystemC `sc_fixed` type. A test case used to test SHaBEPlugIn together with the relevant part of the actual SCMDL document produced by SHaBE is shown in Appendix I. Quite a lot of work has to be performed if all behavioral information described in Section 4.4.2 must be recognized and extracted by SHaBE. We estimate this amount of work to take at least 3 man-months.

The third step of SHaBE described in Section 6.3 is not implemented yet. We estimate this to be a few man-weeks of work.

7.2 Dependencies

Because SHaBE is build upon GDB it has to be used in combination with a C++ compiler which is capable of generating the debug information GDB needs. SHaBE uses GDB to set breakpoints on certain functions in the SystemC library and to inspect certain data members of classes defined in the SystemC library. Therefore, SHaBE depends on a specific implementation of the SystemC library i.e. the OSCI implementation of SystemC version 2.2.0. SHaBE is tested with GDB version 7.1.

SHaBE uses a GCC plug-in to extract the behavior of a SystemC process. GCC plug-ins are only supported for GCC 4.5 or higher and only on platforms which use the ELF object file format at the moment. Therefore SHaBE can not be used with Gygwin nor with MinGW on Microsoft Windows.

7.3 Test Results

We have used a test-first approach while developing SHaBE, see Chapter 5. More than 250 test cases are used during the development of SHaBE. Each test case was implemented immediately after it was written. A test case was either successfully implemented or it was established that it was impossible to implement it due to limitations of the SystemC language or the C++ language. All earlier developed test cases were executed after a new test case was implemented to prevent breaking code which was already correctly working.

Almost all test cases presented in Chapter 5 can be successfully executed by SHaBE. Only the test case presented in Figure 5.7 can not be successfully executed because signals are stored in an array which is dynamically created. As explained in Section 6.1.5, SHaBE is not able to find the names of objects which are accessed via a pointer to an array of objects due to a limitation of the C++ language.

The SCMDL documents produced by SHaBE must follow the structure which is defined in Section 4.5.1. This can be checked by validating these documents with the XML Schema which is presented in Appendix D. All documents produced by SHaBE for all test cases are successfully validated using Xerces[1].

It is mentioned in Section 4.5 that an XSLT [27] script can be used to transform specific parts of a SystemC model described in SCMDL into an other format. To test this, an XSLT stylesheet is written which can transform the hierarchical module structure described in SCMDL that is produced by SHaBE into GraphML. Appendix H presents this XSLT stylesheet and also shows the graph produced for the FIR filter which is instantiated in Figure 2.7 as shown by the graph editor yEd.

During the acceptance test for SHaBE it was found that the current implementation strongly depends on the GDB version used. SHaBE did not work with a version of GDB with the same major and minor version number (7.1) but with a different subversion number. This was unexpected because we only communicate with GDB via GDB/MI which is meant to be a stable interface.

7.4 Execution Time of SHaBE

The first step of SHaBE has a time complexity of $O(n \cdot \log n)$, where n is the number of SystemC objects used in the model. A breakpoint is set and some inspections are performed at the creation of each object. Such an inspection can include a lookup by address of an object found earlier. A search tree, implemented in the standard C++ type `map` is used to store all objects and their address is used as the search key. Therefore, the lookup will take $O(\log n)$ time and will have to be applied for a maximum of n times which results in an overall time complexity of $O(n \cdot \log n)$. The time required by the second step of SHaBE is comparable with the time needed by the GCC C++ front-end. The time complexity of the third step of SHaBE is $O(n \cdot \log n)$ where n is the number of SystemC objects used in the model. All objects found in the second step must be linked by name to the objects found in the first step.

[1] http://xerces.apache.org/

Figure 7.1 shows the execution time[2] of SHaBE for the FIR filter presented in Figure 2.6 for different values of the template parameter `ORDER` which specifies the order of the filter. The data can be found in Appendix J. The duration of the second step of SHaBE is independent of the value of `ORDER` because the functions which describe the behavior of the FIR filter are independent from the order of the filter. The number of `sc_objects` in the model (n) is directly proportional to order of the filter. Figure 7.1 shows that the execution time of the first and last step of SHaBE equals $0.1822 \cdot order + 6.1541$. This formula and the coefficient of determination (R^2) are calculated by the spreadsheet program which was used to create the graph. R^2 is a statistic that shows how well the regression line approximates the real data points. The R^2 value of 1 indicates that the regression line perfectly fits the data. Therefore, the time complexity of the first and last step of SHaBE seems to be $O(n)$. This seems to be better than the true complexity of $O(n \cdot \log n)$ which can be explained by the fact that the look-up time is very small compared with the time needed to communicate with GDB.

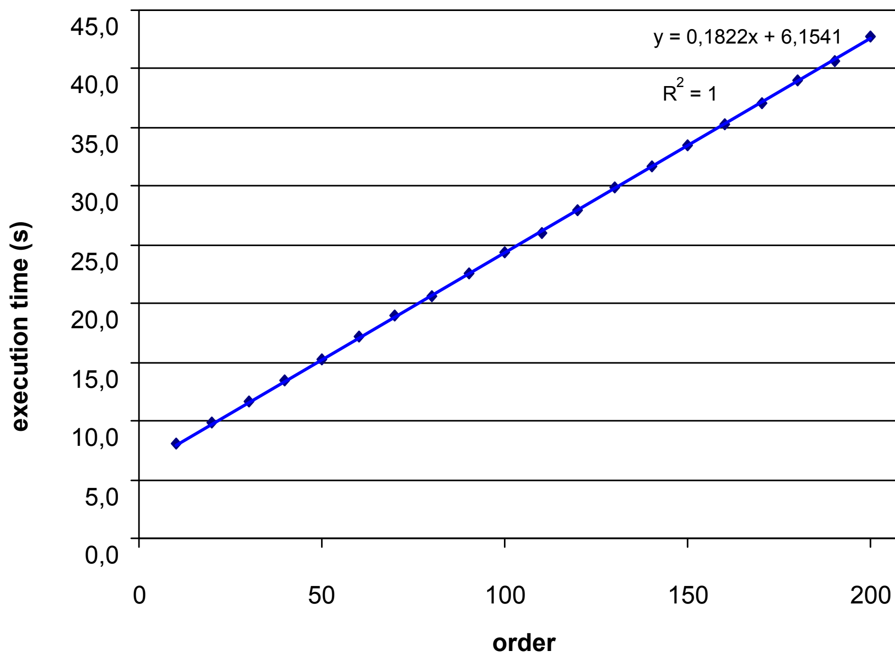


Figure 7.1: The execution time of SHaBE for different orders of the FIR filter presented in Figure 2.6.

Table 7.2 shows the execution time of SHaBE for some SystemC models. The first model is the FIR model which is instantiated in Figure 2.7. The other two models are provided by OSCI and can be found in the examples directory of their SystemC 2.2.0 distribution. A modified version of the RISC CPU[3] was used because the version provided by OSCI produces run-time errors. The column gcc -S shows the execution

[2] All execution times were measured on a PC equipped with 2 GB of RAM and an AMD Athlon 64 processor running at 2 GHz.

[3] http://funningboy.blogspot.com/2010/09/risc-cpu-systemc.html

time of the front-end of the GCC compiler. The execution time of SHaBE is at most 1.6 times the execution time which is needed by the front-end of the GCC C++ compiler.

Table 7.2: Execution times of SHaBE.

Model	# SystemC objects	# Source lines	Execution time (s)	
			gcc -S	SHaBE
FIR	94	214	4.7	7.5
FIR OSCI	37	296	8.3	8.4
RISC CPU OSCI	310	1960	20.8	29.0

8 Conclusions and Future Work

In this chapter some conclusions are drawn, some suggestions for improving SHaBE are put forward, and some directions for future work are given.

8.1 Conclusions

The open-source SystemC front-end SHaBE presented in this thesis is currently the only freely available SystemC front-end which is capable of extracting the module hierarchy and its behavior from a SystemC model when this hierarchy depends on dynamic parameters. This enables SystemC designers to write C++ code which dynamically generates the hierarchical structure of their models. SHaBE can handle models which use dynamic input such as console input, file input, and command-line arguments.

SHaBE uses a novel approach. In our approach the hierarchical information of a SystemC model is retrieved by executing the elaboration phase of the model under control of a debugger. Thereafter, the behavioral information of the model is retrieved by using a C++ compiler extension. Finally, the hierarchical information and the behavioral information are combined and stored as an IR which can be used by tools build upon this front-end. Our approach is completely non-intrusive, i.e., no changes are required in the standard tool flow. The SystemC model and OSCI's SystemC implementation can both be used as is. The only precondition is that both are compiled to include debug information.

The implementation of SHaBE is based on open-source development tools. SHaBE can extract all relevant hierarchical information and a well defined subset of all behavioral information from a model. The implementation is developed using a test-first approach during which more than 250 test cases were successfully implemented. The execution of SHaBE on an executable SystemC model takes only slightly longer than the compilation of this model. Furthermore, the extraction of the module hierarchy of the model has a time complexity of $O(n \cdot \log n)$, where n is the number of SystemC objects used in the model.

To identify the information which must be extracted by a SystemC front-end a SystemC metamodel is defined. This metamodel, which is described in UML, models the module hierarchy of a SystemC model at the end of the elaboration phase. Currently, no other detailed SystemC metamodel has been published.

The output of SHaBE is saved in an XML based format which describes the module hierarchy and the behavior of a SystemC Model. This XML-based language is called SCMDL and an XML Schema definition is provided which can be used for the verification of SCMDL documents. Presently, there is no other XML format available which can be used to describe the module hierarchy of a SystemC model as well as its behavior.

To implement SHaBE we have implemented a parser for the output messages of the GDB/MI which only depends on standard C++ data structures and a GCC plug-in which extracts the AST in SSA form from the source code of a function. These components can be useful in the development of other tools.

SHaBE can be used as a front-end for future SystemC development tools which can visualize, debug, statically verify, or synthesize the model. A designer can use this tool in combination with the freely available SystemC framework provided by OSCI and the freely available GNU compiler and debugger.

8.2 Improving SHaBE

When we look back on the development of SHaBE we suggest the following improvements:

- When extracting the module hierarchy, the C++ names and C++ types of ports, exports, channels, and submodules, which are declared inside an user-defined module are retrieved. Other data members of such an user-defined module are discarded. It will be an improvement if these data members would also be described in the hierarchical part of the SCMDL document produced by SHaBE. These data members can be used in the description of the behavior of the model. The type of these data members can be retrieved by SHaBEPlugIn but it is much simpler to do this in the first step of SHaBE because all data members of a user-defined module are already inspected during this step, see Section 6.1.5.

- A member function which is registered as a SystemC process can call other user-defined functions. At the moment these calls are included as `call` AST nodes in the behavioral part of the SCMDL document produced by SHaBE. It will be an improvement if SHaBEPlugIn would be recursively called for these user-defined functions. This would make the ASTs which defines the behavior of these functions available to the tools build upon our front-end.

- If the source codes of the member functions are divided among several source files, GCC is called once for every source file, see Section 6.2.2, during the second step of SHaBE. At the moment, GCC is called several times consecutively. Because the files which are used to communicate between SHaBE and the GCC plug-in SHaBEPlugIn are placed in an unique directory for each source file these calls to GCC can be performed in parallel. This can significantly speed up the execution time of SHaBE especially on multi-core machines. For example, to process the model of the OSCI RISC CPU SHaBE spends more than 70% of its execution time in the second step (executing GCC), see Table 7.2. The source code of the OSCI RISC CPU model is distributed over more than ten files, so the second step of SHaBE GCC can be parallelised. For example, if a speedup of 4 can be achieved in the second step for this model, then the total speedup will be $1/(0.3 + 0.7/4) = 2.1$.

8.3 Future Work

A possible application which uses SHaBE as a front-end, would be a tool which converts a SystemC model comprising a parameterized dynamic hierarchy into a SystemC model with a fully expanded hierarchy. The resulting SystemC model can then be further processed using existing tools.

Also, it would be interesting to investigate if the approach used in SHaBE can also be applied to SystemC models which use the SystemC TLM and/or the SystemC Analog/Mixed-signal (AMS) extensions.

References

[1] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[2] Altera. White paper: Automated generation of hardware accelerators with direct memory access from ANSI/ISO standard C functions. http://www.altera.com/literature/wp/wp-aghrdwr.pdf, 2006.

[3] Jim Barnett, Rahul Akolkar, RJ Auburn, Michael Bodell, Daniel C. Burnett, Jerry Carter, Scott McGlashan, Torbjörn Lager, Mark Helbing, Rafah Hosn, T. V. Raman, and Klaus Reifenrath. State Chart XML (SCXML): State machine notation for control abstraction. W3C Working Draft, May 2010.

[4] K. Beck. Embracing change with extreme programming. *Computer*, 32(10):70 –77, October 1999.

[5] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[6] Y. Ben Asher and N. Rotem. Binary synthesis with multiple memory banks targeting array references. In *International Conference on Field Programmable Logic and Applications*, 31 2009.

[7] V. Berman. Standards: The P1685 IP-XACT IP metadata standard. *IEEE Design & Test of Computers*, 23(4):316–317, April 2006.

[8] David Berner, Hiren D. Patel, Deepak Mathaikutty, and Sandeep K. Shukla. Automated extraction of structural information from systemC-based IP for validation. In *Sixth International Workshop on Microprocessor Test and Verification, Common Challenges and Solutions, 3-4 November 2005, Austin, Texas, USA*, pages 99–104, 2005.

[9] David Berner, Jean-Pierre Talpin, Hiren Patel, Deepak Abraham Mathaikutty, and Eep Shukla. SystemCXML: An extensible SystemC front end using XML. In *In Proceedings of the Forum on specification and design languages*, 2005.

[10] Loïc Besnard, Thierry Gautier, Matthieu Moy, Jean-Pierre Talpin, Kenneth Johnson, and Florence Maraninchi. Automatic translation of C/C++ parallel code into synchronous formalism using an SSA intermediate form, 2009.

[11] Wilbert Bilderbeek, Harry Broeders, and Alex van Rooijen. The THRSim11 68HC11 simulator. *Dr. Dobb's Journal of Software Tools*, 24(3):54–55, March 1999.

[12] Paul V. Biron and Ashok Malhotra. XML schema part 2: Datatypes second edition. W3C Recommendation, October 2004.

[13] Nicolas Blanc, Daniel Kroening, and Natasha Sharygina. SCOOT: A tool for the analysis of SystemC models. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS*

2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings, volume 4963 of *Lecture Notes in Computer Science*, pages 467–470. Springer, 2008.

[14] Jonathan Borden. The resource directory description language (RDDL): What goes at the end of a namespace URI? In *Extreme Markup Languages*, 2001.

[15] Ulrik Brandes, Markus Eiglsperger, Ivan Herman, Michael Himsolt, and M. Scott Marshall. GraphML progress report: Structural layer proposal. In *Proc. 9th Intl. Symp. Graph Drawing (GD '01) LNCS 2265*, pages 501–512. Springer-Verlag, 2002.

[16] Harry Broeders. How to model a FIR filter in SystemC? Technical report, The Hague University, March 2010.

[17] Mihai Budiu. *Spatial Computation*. PhD thesis, Carnegie Mellon University, December 2003.

[18] Mihai Budiu and Seth Copen Goldstein. Compiling application-specific hardware. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 853–863, Montpellier (La Grande-Motte), France, September 2–4 2002.

[19] Mihai Budiu and Seth Copen Goldstein. Pegasus: An efficient intermediate representation. Technical Report CMU-CS-02-107, Carnegie Mellon University, 2002.

[20] Cadence. Technical paper: Cadence C-to-Silicon compiler delivers on the promise of high-level synthesis. http://www.cadence.com/rl/Resources/technical_papers/c_to_silicon_tp.pdf, 2008.

[21] Sean Callanan, Daniel J. Dean, and Erez Zadok. Extending GCC with modular GIMPLE optimizations. 2008.

[22] J. Castillo, P. Huerta, and J.I. Martinez. An open-source tool for SystemC to Verilog automatic translation. *Latin American Applied Research*, March 2007.

[23] D. Chen, J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang. xPilot: A platform-based behavioral synthesis system. In *Proceedings of SRC Techcon Conference*, 2005.

[24] Yan Chen, Xuan Du, Xuegong Zhou, and Chenglian Peng. An automatic coverage analysis for SystemC using UML and aspect-oriented technology. In Weiming Shen, Zongkai Lin, Jean-Paul Barths, and Tangqiu Li, editors, *Computer Supported Cooperative Work in Design*, volume 3168 of *Lecture Notes in Computer Science*, pages 398–405. Springer Berlin / Heidelberg, 2005.

[25] Yan Chen, Weidong Qiu, Bo Zhou, and Chenglian Peng. An automatic test coverage analysis for SystemC description using aspect-oriented programming. In *Computer Supported Cooperative Work in Design, 2004. Proceedings. The 8th International Conference on*, volume 2, pages 632 – 636 Vol.2, May 2004.

[26] Cristina Cifuentes and K. John Gough. Decompilation of binary programs. *Softw., Pract. Exper.*, 25(7):811–829, 1995.

[27] James Clark. XSL Transformations (XSLT) version 1.0. W3C Recommendation, November 1999.

[28] Philippe Coussy and Dominique Heller. GAUT - high-level synthesis tool from C to RTL. http://www-labsticc.univ-ubs.fr/www-gaut/.

[29] Philippe Coussy and Adam Morawiec, editors. *High-Level Synthesis - from Algorithm to Digital Circuit*. Springer, 2008.

[30] José Gabriel F. Coutinho, Jun Jiang, and Wayne Luk. Interleaving behavioral and cycle-accurate descriptions for reconfigurable hardware compilation. In *FCCM*, pages 245–254. IEEE Computer Society, 2005.

[31] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct 1991.

[32] E. N. Dolgova and A. V. Chernov. Automatic reconstruction of data types in the decompilation problem. *Programming and Computer Software*, 35(2):105–119, 2009.

[33] Rolf Drechsler, Görschwin Fey, Christian Genz, and Daniel Große. SyCE: An integrated environment for system design in SystemC. In *IEEE International Workshop on Rapid System Prototyping*, pages 258–260. IEEE Computer Society, 2005.

[34] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen North, and Gordon Woodhull. Graphviz open source graph drawing tools. In Petra Mutzel, Michael Jnger, and Sebastian Leipert, editors, *Graph Drawing*, volume 2265 of *Lecture Notes in Computer Science*, pages 594–597. Springer Berlin / Heidelberg, 2002.

[35] Van Emmerik. *Static Single Assignment for Decompilation*. PhD thesis, School of ITEE, University of Queensland, May 2007.

[36] Görschwin Fey, Daniel Große, Tim Cassens, Christian Genz, Tim Warode, and Rolf Drechsler. ParSyC: An efficient SystemC parser. In *In Workshop on Synthesis And System Integration of Mixed Information technologies*, pages 148–154, 2004.

[37] D Gajski. NISC: The ultimate reconfigurable component. Technical Report TR 03-28, Center for Embedded Computer Systems, October 2003.

[38] D.D. Gajski, S. Abdi, A. Gerstlauer, and G. Schirner. *Embedded System Design - Modeling, Synthesis and Verification*. Springer, 2009.

[39] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.

[40] Gansner and North. An open graph visualization system and its applications to software engineering. *SOFTPREX: Software–Practice and Experience*, 30, 2000.

[41] Thierry Gautier and Paul Le Guernic. SIGNAL: A declarative language for synchronous programming of real-time systems. In Gilles Kahn, editor, *FPCA*, volume 274 of *Lecture Notes in Computer Science*, pages 257–277. Springer, 1987.

[42] Christian Genz and Rolf Drechsler. Overcoming limitations of the SystemC data introspection. In *DATE*, pages 590–593. IEEE, 2009.

[43] Christian Genz, Rolf Drechsler, Gerhard Angst, and Lothar Linhard. Visualization of SystemC designs. In *ISCAS*, pages 413–416. IEEE, 2007.

[44] Bita Gorjiara, Mehrdad Reshadi, and Daniel Gajski. Aspect-oriented architecture description for retargetable compilation, simulation and synthesis of application-specific pipelined datapaths. In *ICCD*. IEEE, 2006.

[45] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. The Java Series. Addison-Wesley, Boston, Mass., 2005.

[46] Daniel Große and Rolf Drechsler. CheckSyC: an efficient property checker for

RTL SystemC designs. In *ISCAS (4)*, pages 4167–4170. IEEE, 2005.

[47] Daniel Große, Rolf Drechsler, Lothar Linhard, and Gerhard Angst. Efficient automatic visualization of SystemC designs. In *FDL*, pages 646–658. ECSI, 2003.

[48] Zhi Guo, Betul Buyukkurt, and Walid Najjar. Input data reuse in compiling window operations onto reconfigurable hardware. *ACM SIGPLAN Notices*, 39(7):249–256, July 2004.

[49] Sumit Gupta, Nikil D. Dutt, and Rajesh Gupta. *SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*. Springer, 2004.

[50] A. Habibi, H. Moinudeen, and S. Tahar. Generating finite state machines from SystemC. In *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, volume 2, pages 1–6, March 2006.

[51] Paula Herber, Joachim Fellmuth, and Sabine Glesner. Model checking SystemC designs using timed automata. In *Proceedings of the 6th International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2008, Atlanta, GA, USA, October 19-24, 2008*, pages 131–136. ACM, 2008.

[52] Michael Hind. Pointer analysis: haven't we solved this problem yet? In *PASTE*, pages 54–61. ACM, 2001.

[53] Glenn Holloway. The machine-SUIF static single assignment library, July 2002.

[54] Glenn Holloway and Allyn Dimock. The machine SUIF bit-vector data-flow-analysis library, July 2002.

[55] Glenn Holloway and Michael D. Smith. The machine-SUIF control flow analysis library, July 2002.

[56] Glenn Holloway and Michael D. Smith. The machine-SUIF control flow graph library, July 2002.

[57] Glenn Holloway and Michael D. Smith. The machine-SUIF machine library, 2002.

[58] IEEE. *Std 1666 - 2005 IEEE Standard SystemC Language Reference Manual*, 2006.

[59] C. Norris Ip and Stuart Swan. A tutorial introduction on the new SystemC verification standard. Presented at the 7th European SystemC Users Group Meeting at DATA, 2003.

[60] ISO. *ISO/IEC 14882:1998: Programming languages – C++*. International Organization for Standardization, September 1998.

[61] S. C. Johnson. Yacc: Yet another compiler compiler. Computer Science Technical Report 32, Bell Laboratories, Murray Hill, NJ, 1975.

[62] Simon Peyton Jones. *Haskell 98 language and libraries : the revised report*. Cambridge University Press, Cambridge U.K. New York, 2003.

[63] G. Kahn. The semantics of a simple language for parallel programming. In *Proc. IFIP 74*, pages 471–475, Amsterdam, 1974.

[64] Hamoudi Kalla, Jean-Pierre Talpin, David Berner, and Loïc Besnard. Automated translation of C/C++ models into a synchronous formalism. In *ECBS*, pages 426–436. IEEE Computer Society, 2006.

[65] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag,

Berlin, Heidelberg, and New York, 1997.

[66] Brad King. <GCC XML description="XML output for GCC"/>. http://www.gccxml.org/HTML/Index.html.

[67] Viswanathan Kodaganallur. Incorporating language processing into Java applications: A JavaCC tutorial. *IEEE Software*, 21(4):70–77, 2004.

[68] Chris Lattner. LLVM language reference manual. http://llvm.org/docs/LangRef.html.

[69] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE Computer Society, 2004.

[70] M. E. Lesk. Lex - A lexical analyzer generator. Technical Report CS-39, AT&T Bell Laboratories, Murray Hill , NJ , USA, 1975.

[71] Karthikeyan Manivannan. Design and development of a GCC based C front-end for the NISC compiler. Master's thesis, University of California, Irvine, 2007.

[72] Kevin Marquet and Matthieu Moy. PinaVM: a SystemC front-end based on an executable intermediate representation. In *International Conference on Embedded Software International Conference on Embedded Software*, page 79, Scottsdale, USA, 10 2010.

[73] Kevin Marquet, Matthieu Moy, and Bageshri Karkare. A theoretical and experimental review of SystemC front-ends. In *Forum for Design Languages (FDL)*, 2010.

[74] Grant Martin and Gary Smith. High-level synthesis: Past, present, and future. *IEEE Design and Test of Computers*, 26(4):18–25, 2009.

[75] Robert C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2002.

[76] Robert C. Martin. *Clean Code: A handbook of agile software craftsmanship*. Prentice Hall, 2009.

[77] D.A. Mathaikutty and S.K. Shukla. MCF: A metamodeling-based component composition frameworkcomposing SystemC IPs for executable system models. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(7):792–805, July 2008.

[78] Jason Merrill. GENERIC and GIMPLE: A new tree representation for entire functions. In Andrew J. Hutton, Stephanie Donovan, and C. Craig Ross, editors, *Proceedings of the GCC Developers Summit May 25–27, 2003, Ottawa, Ontario Canada*, pages 171–193, 2003.

[79] Matthieu Moy. *Techniques and Tools for the Verification of Systems-on-a-Chip at the Transaction Level*. PhD thesis, INPG, Grenoble, France, December 2005.

[80] Matthieu Moy, Florence Maraninchi, and Laurent Maillet-Contoz. LusSy: A toolbox for the analysis of systems-on-a-chip at the transactional level. In *International Conference on Application of Concurrency to System Design*, 2005.

[81] Matthieu Moy, Florence Maraninchi, and Laurent Maillet-Contoz. PINAPA: An extraction tool for SystemC descriptions of systems-on-a-chip. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 317–324, 2005.

[82] N. C. Myers. Traits: A new and useful template technique. *C++ Report*, June

1995.

[83] Ravi Namballa, N. Ranganathan, and Abdel Ejnioui. Control and data flow graph extraction for high-level synthesis. *VLSI, IEEE Computer Society Annual Symposium on*, 0:187, 2004.

[84] D. Novillo. Design and implementation of tree SSA. In *Proceedings of the 2004 GCC Summit*, Ottawa, Canada, June 2004.

[85] D. Novillo. GCC - An architectural overview, current status and future directions. In *Proceedings of the Linux Symposium*, volume Volume 2, pages 185–200, Ottawa, Ontario, Canada, July 2006.

[86] Object Management Group. Object Constraint Language version 2.2, February 2010.

[87] Object Management Group. UML 2.3 Superstructure, May 2010.

[88] OSCI. SystemC synthesizable subset draft 1.3. http://www.systemc.org/downloads/drafts_review/, August 2009.

[89] P. R. Panda. SystemC - a modeling platform supporting multiple design abstractions. In *Proc. 14th International Symposium on System Synthesis*, pages 75–80, 2001.

[90] Pavel Parfuntseu. KaSCPar Karlsruhe SystemC parser suite online documentation. http://kmir.de/downloads/sim/archives/kascpar-documentation.pdf, August 2006.

[91] Terence Parr. *The Definitive ANTLR Reference Guide: Building Domain-specific Languages*. Pragmatic Bookshelf, Raleigh, NC, 2007.

[92] Terence John Parr. *Language translation using PCCTS and C++: A reference guide*. Automata Publishing Company, San Jose, CA, USA, January 1997.

[93] David J. Pearce, Paul H. J. Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis for C. In *In ACM workshop on Program Analysis for Software Tools and Engineering*, pages 37–42. ACM Press, 2004.

[94] et al. Quinlan, D.J. ROSE compiler project. http://www.rosecompiler.org/.

[95] Elvinia Riccobene and Patrizia Scandurra. Model transformations in the UPES/UPSoC development process for embedded systems. *Innovations in Systems and Software Engineering*, 5:35–47, 2009.

[96] Vikram Singh Saun and Preeti Ranjan Panda. Extracting exact finite state machines from behavioral SystemC descriptions. *VLSI Design, International Conference on*, 0:280–285, 2005.

[97] Andreas Schallenberg, Wolfgang Nebel, Andreas Herrholz, Philipp A. Hartmann, and Frank Oppenheimer. OSSS+R: A framework for application level modelling and synthesis of reconfigurable systems. In *DATE*, pages 970–975. IEEE, 2009.

[98] Thorsten Schubert and Wolfgang Nebel. The Quiny SystemC front end: Self-synthesising designs. In *FDL*, pages 135–143. ECSI, 2006.

[99] Olaf Spinczyk, Andreas Gal, and Wolfgang Schrder-Preikschat. AspectC++: An aspect-oriented extension to the C++ programming language. In *CRPIT '02: Proceedings of the Fortieth International Conference on Tools Pacific*, pages 53–60, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[100] Richard M. Stallman and GCC DeveloperCommunity. *Using the GNU Compiler Collection (GCC)*. Free Software Foundation, 51 Franklin Street, Fifth Floor,

Boston, MA 02110-1301, USA, 2010.

[101] Richard M. Stallman, Roland Pesch, Stan Shebs, et al. *Debugging with GDB: The GNU Source-Level Debugger*. Free Software Foundation, 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA, ninth edition, 2010.

[102] Greg Stitt and Frank Vahid. New decompilation techniques for binary-level co-processor generation. In *ICCAD*, pages 547–554. IEEE Computer Society, 2005.

[103] Greg Stitt and Frank Vahid. Binary synthesis. *ACM Trans. Design Autom. Electr. Syst.*, 12(3), 2007.

[104] SystemCrafter. SystemCrafter SC user manual version 3.0.0. http://www.systemcrafter.com/downloads/User%20Manual.pdf.

[105] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. XML schema part 1: Structures second edition. World Wide Web Consortium, Recommendation REC-xmlschema-1-20041028, October 2004.

[106] Bas van den Aardweg. SystemC omzetten naar een diagram. Technical report, The Hague University, July 2010.

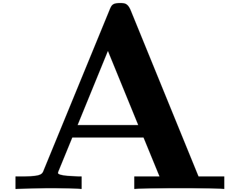
[107] T.G.R. van Leuken; A.C. de Graaf; H. Lincklaen-Arriens. A high-level design and implementation platform for IP prototyping on FPGA platforms. In *ProRISC IEEE 15th Annual Workshop on Circuits, Systems and Signal Processing*, 2004.

[108] Yiyin Wang. System design methodologies – High level synthesis and a VHDL implementation of a practical scheme for UWB communication. Master's thesis, Delft University of Technology, 2004.

[109] Trevor Wieman. CCIWG update video. http://media.systemc.org/CCIWGUpdate/player.html, July 2009.

[110] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, SteveW. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, December 1994.

C/C++/SystemC Synthesis Tools



Because the CaS group is especially interested in the synthesis of SystemC models we have also studied some C/C++/SystemC synthesis tools. In this appendix we briefly describe some commercial and academic High-Level Synthesis (HLS) tools. In these descriptions we focus on the way in which the input language is parsed and on the IR used by the tool. We are hoping to find a tool with a well-defined intermediate representation which we can use for the SystemC front-end that we want to develop.

A.1 Commercial Tools

There are quite a few commercial C/C++/SystemC synthesis tools available. A selection is given below.

Cascade from CriticalBlue is an automated co-processor synthesis solution. Using the profiling results of the application the specific tasks to be migrated to the co-processor are identified by the user. Cascade then generates the RTL code for the co-processor hardware. It also generates the interface hardware necessary to connect this co-processor with the CPU.

Catapult C Synthesis from Mentor Graphics takes an algorithm written in ANSI C++ and a set of user directives as input and generates an RTL implementation. The input specification is sequential and does not include any notion of time or explicit parallelism. It does not specify the interface nor the architecture of the design. The synthesizable subset of the SystemC integer and fixed-point data types are supported by Catapult. It also generates the required verification infrastructure in SystemC.

The Nios II C-to-Hardware Acceleration Compiler (C2H) from Altera transforms a function specified in C/C++ into a hardware accelerator which is mapped into the memory map of the Nios softcore. Some implementation details are revealed in [2]. The Nios II C2H compiler extracts instruction-level parallelism from the C/C++ code through construction and analysis of control and data flow graphs similar to those described in [30]. A number of compiler optimizations are performed including pointer analysis. The user can assist the compiler by using the ISO C99 restrict keyword which specifies that writes through this pointer will not effect the values read through other pointers available in the same context which are also declared as restricted. The scheduler is aware of the specific memory latencies and therefore Latency-aware scheduling and pipelining are used. The execution of successive loop iterations is pipelined. Operations in control blocks are executed speculatively and multiplexed using their condition expression at the merging of control paths if possible.

In the C2R compiler from Cebatech the hardware architecture is defined in the untimed C source by coding the datapath and parallelism needed using compiler directives. The compiler generates an RTL implementation.

C-to-Silicon Compiler from Cadence generates synthesizable RTL starting from an untimed C/C++/SystemC model. Some implementation details are revealed in [20]. The IR is saved in the Behavior Structure Timing (BST) database.

CyberWorkBench (CWB) from NEC is a C-based behavioral synthesis tool. CWB supports various C-based language including SystemC as an input description. The SystemC code is parsed into CWB's IR which is called tree-structured Control Flow Graph (tCFG). This tCFG is transformed into a unique CDFG. According to [29] all synthesis tasks are performed on those two data structures but we were not able to find any details about these structures.

Cynthesizer from Forte Design Systems generates an RTL description from a high-level SystemC TLM model. This model describes a pin-accurate and protocol-accurate model of the design. The protocol is specified by SystemC port `read`, port `write`, and `wait` statements. The behavioral SystemC code which specifies the computation is written without `wait` statements and is scheduled by Cynthesizer to satisfy latency, pipelining, and other constraints given by the designer.

The Impulse C software-to-hardware compiler can Compile C code into a processor-attached accelerator.

PICO Algorithmic Synthesis from Synfora creates application accelerators from untimed C. PICO is intended for applications that process data streams. This application is modeled as a Kahn process network [63], in which a set of sequential processes communicate via streams through unbounded FIFOs. PICO generates the RTL description for the hardware and its related software. It also produces a SystemC TLM model of the hardware at two levels of abstraction: an untimed programmer's view and a timed programmer's view.

SystemCrafter (SC) is a SystemC synthesis tool for Xilinx FPGAs. Not all off the OSCI SystemC synthesizable subset [88] is supported by SystemCrafter. For example: input-output ports (`sc_inout`), fixed point data types, `continue`, and `break` statements in loops, overloading, inheritance, and templates are not supported [104].

There is a lot of movement in the world of HLS tools. Take for example Handel-C, a rich subset of C with non-standard extensions to control hardware instantiation and parallelism which can easier be translated into RTL than standard C. It was originally developed at the Oxford University and commercialized by a company called Celoxica around 2000. In 2006 Celoxica's HLS business was acquired by Catalytic. Soon thereafter, Celoxica and Catalytic merged into a new company called Agility. In 2009, Mentor Graphics acquired Agility's C synthesis assets.

In a recent article [74] Mentor Graphics is qualified as the leader in the C to RTL SoC design market, and in the C to FPGA market. According to this article Forte leads in the SystemC to RTL area for SoC design.

A more in depth description of some of these tools (Catapult, PICO, Cynthesizer, AutoPilot, and CyberWorkBench) can be found in the book High-Level Synthesis - From Algorithm to Digital Circuit [29].

We can conclude from our survey of commercial available C/C++/SystemC HLS tools that most tools do not reveal their inner workings. This does not surprise us because these inner workings are valuable business assets. The list of customers on the website of the commercial EDG C++ front-end includes Mentor Graphics which

suggest that Catapult uses EDG as its front-end.

A.2 Academic Tools

HLS is a popular research topic in the academic world. Therefore many academic HLS tools are available. A selection is given below.

A.2.1 CASH

Compiler for Application Specific Hardware (CASH) [18] is part of the Phoenix project at Carnegie Mellon University. The Phoenix project explores the direct implementation of programs in (reconfigurable) hardware. CASH translates ANSI-C programs into asynchronous circuits. The CASH front-end translates the C code into a dataflow IR called Pegasus. This IR is expressed in SSA [31] form and is described in detail in [19]. The C front-end is based on Stanford University Intermediate Format (SUIF) 1 [110]. See also Section A.2.5. CASH and Pegasus are extensively described in the Ph.D. thesis of Mihai Budiu [17]. There seems to be no activity in the Phoenix project beyond June 2007. As far as we know, the CASH compiler is not publicly available.

A.2.2 Fossy

The tool called "Functional Oldenburg System Synthesizer" (Fossy) [97] is developed as part of the "Analysis and Design of run-time Reconfigurable, heterogeneous Systems" (ANDES) European project. Fossy is a tool for transforming system-level SystemC models to synthesizable VHDL. It is written in the pure functional programming language Haskell [62]. The commercial EDG C++ front-end parser is used and augmented with a thin layer converting the front-end specific IR to XML. Fossy is currently freely available for evaluation purposes in the form of an online tool. A SystemC module description can be entered in an HTML form and the resulting VHDL RTL description is returned by email. The Fossy source code is not publicly available.

A.2.3 GAUT

Universit de Bretagne Sud has developed GAUT [28] an HLS tool dedicated to digital signal processing applications. Starting from a C function, GAUT generates an RTL description in VHDL. It also generates a SystemC cycle-accurate simulation model for simulation-based validation. The compiler of GAUT derives GCC 4.2 to extract a Data Flow Graph (DFG) representation of the application. The GCC GIMPLE form [78] is translated into the GAUT IR. The DFG generated by the GAUT compiler can be visualized in the GAUT IDE. From this IR GAUT generates a potentially pipelined architecture. This architecture consists of a processing, a memory, and a communication unit.

A.2.4 NISC

The No-Instruction-Set-Computer (NISC) [37] is developed at the University of California, Irvine. It can be used as a C to RTL HLS tool. NISC can be evaluated online. First the design constraints must be specified. Then the C code can be paste into an HTML form and finally the output files including an RTL Verilog implementation can be downloaded. NISC can also be user to generate a control unit to execute the C code on a given datapath. This datapath can be defines using Generic Netlist Representation (GNR) [44] an XML format defined by the NISC developers to describe the target architecture. The NISC source code is not publicly available. The release notes of the latest release at the time of this writing say:

> "You can use the NiscToolset to synthesize other languages such as SystemC all you need to do is to generate the CDFG of the application in XML format for the NiscCompiler. The format is very straight forward and you can use the Msil2Nisc as guideline example."

This last advise is difficult to follow because the source code for the Msil2Nisc program is not available. The development of the C front-end for NISC was the subject of a master's thesis [71]. The front-end is implemented as an additional pass inside the GCC compiler version 4.3.0. This additional pass converts GCC's GIMPLE IR [78] to XML files. The XML files are read as input by the NISC compiler for its compilation process.

A.2.5 ROCCC

The Riverside Optimizing Configurable Computing Compiler (ROCCC) [48] is an open-source C-to-VHDL compiler infrastructure tool. Using profiling techniques ROCCC identifies the frequently executing code kernels in a given application. ROCCC then compiles these kernels to HDL code which is synthesized using commercial tools. The ROCCC system is built using SUIF 2 [110] and Machine-SUIF [57] platforms. Within the SUIF compiler infrastructure several C++ libraries are available to manage and analyze CFGs [56][55] and DFGs [54] and there is also a library available to convert a DFG into SSA form [53].

A.2.6 sc2v

SystemC to Verilog (sc2v) [22] is an open-source tool which translates SystemC RTL code into Verilog RTL code. The SystemC front-end was build using the well known lex [70] and yacc [61] parsing tools. It performs a relatively simple one-to-one translation and the supported SystemC subset is very small.

A.2.7 SPARK

SPARK is a C to VHDL HLS tool developed at the University of California, San Diego (UCSD). The IR format used by SPARK is described in detail in [49]. SPARK uses a CFG, a DFG, and a Hierarchical Task Graph (HTG). The HTG captures the

program structure in the input description. The nodes of these three graphs are interconnected and together form a 3-layered graph. These graphs can be visualized by using command-line options. When these command line options are used SPARK will produce output files in the DOT language. These files can be visualized using the Graph Visualization Software (Graphviz) [40] an open-source tool initiated by AT&T Research Labs. SPARK uses the commercial EDG C/C++ front end. The most recent version of SPARK available for download at the time of this writing is build almost 5 years ago. Around that same time Wang [108] investigated SPARK and she reported:

> "By doing several tests, we found out SPARK cannot support array access and SPARK cannot support while loop statement. The adder component and multiplier component generated automatically by SPARK are not correct; they have value out of range problem. There are code errors in the VHDL code generated by SPARK. The dataflow is not correct. In summary, SPARK is a lab tool."

A.2.8 xPilot

xPilot [23] is developed at the University of California, Los Angeles (UCLA). It can synthesize a behavioral SystemC model into an RTL implementation given the design constraints. The SystemC front-end is developed using the LLVM compiler infrastructure [69]. LLVM is a compiler infrastructure and a virtual instruction set [68]. The virtual instruction set is a low-level code representation that provides rich type and dataflow information. The DFG is in SSA form [31]. The LLVM compiler infrastructure includes a GCC-based C and C++ front-end. The IR representation produced by LLVM is converted by xPilot into a System-level Synthesis Data Model (SSDM). The SSDM consists of processes which describe the behavior of the SystemC modules and channels which describe the connections between the processes. The behavior of each process is captured in a CDFG. The xPilot system is being licensed by AutoESL Design Technologies, Inc. as the basis for a commercial Electronic System-Level (ESL) synthesis tool called AutoPilot.

A.3 Conclusions Drawn from the Survey of C/C++/SystemC Synthesis Tools

From our survey of C/C++/SystemC Synthesis Tools there are several conclusions that can be drawn. As far as we can determine all of these tools use a parser to extract the behavioral information from the C/C++/SystemC model. Some use the commercial EDG C/C++ front-end. Even two academic tools SPARK and Fossy are using this commercial front-end. Other academic front-ends use yacc (sc2v) or the compiler infrastructure LLVM (xPilot) or the SUIF compiler infrastructure (ROCCC and CASH). One academic tool (NISC) uses GCC (version 4) as a front-end.

We can also conclude that no standard IR is available. Most tools use some kind of AST in SSA form. If we want to connect our SystemC front-end to an RTL generating back-end we must design our IR in such a way that it can easily be converted to the

IR used in the back-end of our choice. The ROCCC and GAUT back-ends are the best candidates at the moment because these tools are actively being developed and both have a documented IR.

B
SystemC Metamodel

The SystemC metamodel which models a SystemC model at the end of the elaboration phase is described in Section 4.4.1. Some detailed diagrams are shown in this Appendix.

The primitive channel sc_signal and its derivatives are the only channels which are included in the synthesizable subset of SystemC. These channels are shown in Figure B.1. An sc_signal<T> is derived from the interface class sc_signal_inout_if<T> which is shown in Figure B.2.

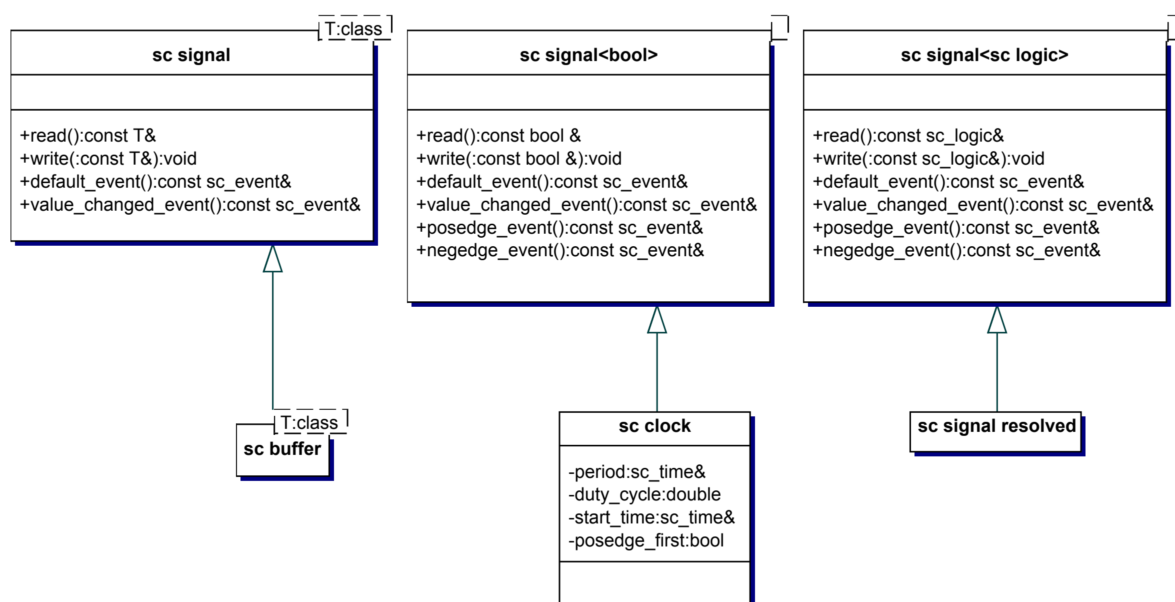


Figure B.1: The primitive channels which are defined in the synthesizable subset of SystemC.

An input port sc_in<T> requires to be bound to a channel which implements the sc_signal_in_if<T> for example an sc_signal<T>, see Figure B.3. Similar diagrams can be drawn for the other synthesizable ports.

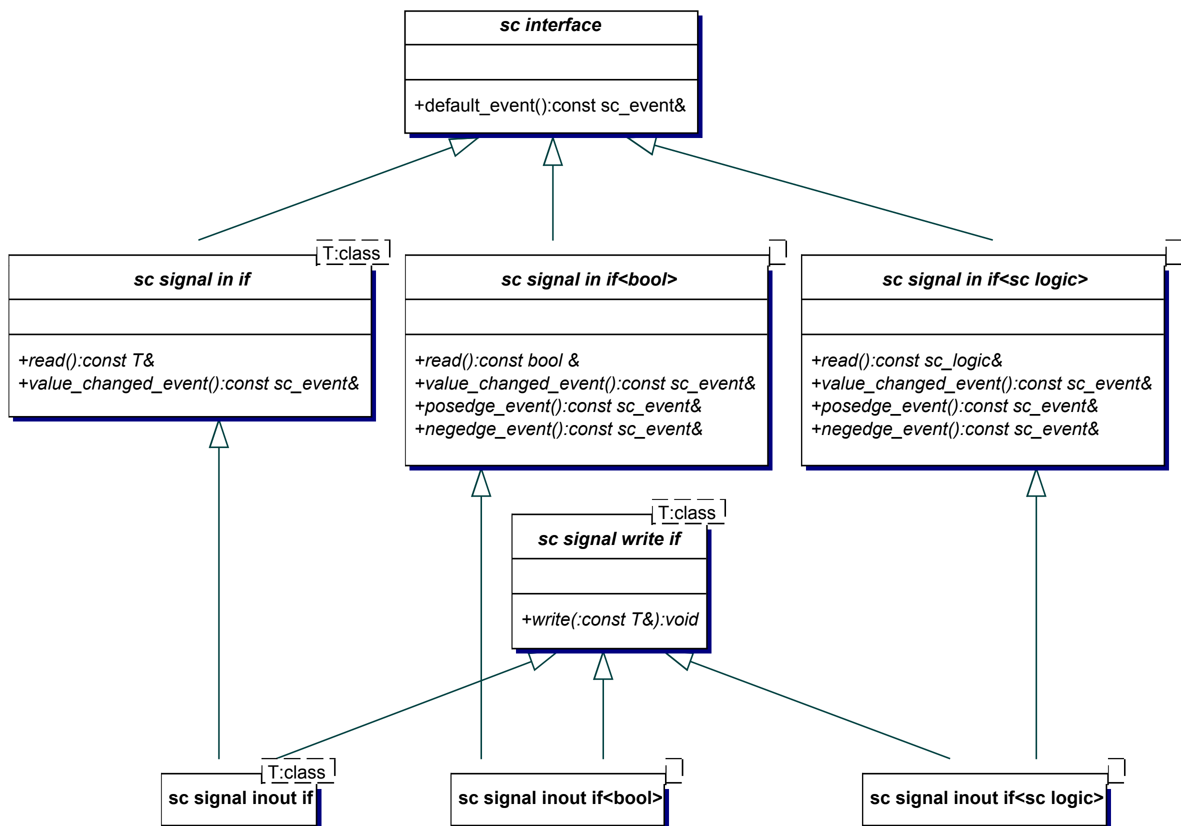


Figure B.2: The interfaces which are implemented by signals.

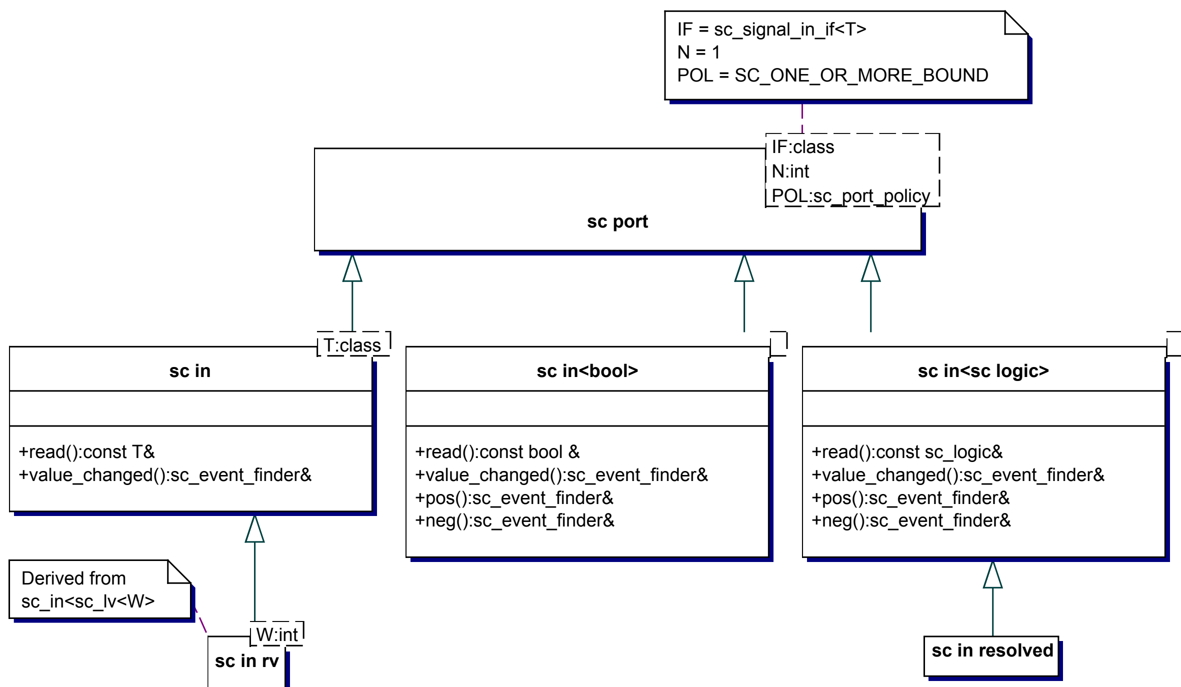
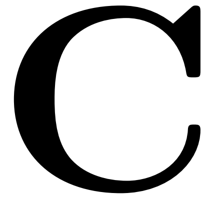


Figure B.3: The input ports which are defined in the SystemC standard.

Information which has to be Retrieved from a SystemC Model



The hierarchical information which has to be retrieved from a SystemC model at the end of the elaboration phase is identified in Section 4.4.1 and listed in this Appendix. The information items are sorted by priority. The priority values used are explained in Chapter 4.

Table C.1: The hierarchical information which must be retrieved from a SystemC model.

object	information	cardinality	priority
`sc_object`	SystemC name	1	1
`sc_object`	SystemC type	1	1
`sc_object`	C++ name	0..1	1
module	ports	0..*	1
module	exports	0..*	1
module	processes	0..*	1
module	primitive channels	0..*	1
module	submodules	0..*	1
hierarchy	top-level primitive channels	0..*	1
hierarchy	top-level modules	0..*	1
export	bound to export	0..1	1
export	bound to channel	0..1	1
port	bound to ports	0..N	1
port	bound to exports	0..N	1
port	bound to channels	0..N	1
process	sensitive to event from channels bound to ports	0..*	1
process	sensitive to event from channel bound to exports	0..*	1
process	sensitive to event from channels	0..*	1
process	name of the member function which describes the behavior	1	1
process	source file of the member function which describes the behavior	1	1
clocked thread process	reset signal or reset port	0..1	1
`sc_object`	C++ type	1	2
clock	period	1	2
clock	duty cycle	1	2
clock	start time	1	2
clock	posedge first	1	2
process	don't initialize	1	2
clocked thread process	reset active level	0..1	2
process	subprocess	0..*	3
fifo	size	1	3
semaphore	value	1	3
`sc_object`	address	1	4
process	address of the member function which describes the behavior	1	4
thread process	stack size	1	4

SystemC Model Description Language

D

A SystemC model can be described in XML using the SCMDL. This XML format is defined in the XML Schema shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:target="http://shabe.sourceforge.net/systemc-model"
  targetNamespace="http://shabe.sourceforge.net/systemc-model"
  elementFormDefault="qualified"
  version="1.0">
  <annotation>
    <documentation xml:lang="en">
      This XML Schema can be used to describes a SystemC model.
      Author: Harry Broeders mailto:j.z.m.broeders@hhs.nl
      Date: November 25, 2010
    </documentation>
  </annotation>
  <attributeGroup name="object-identifier">
    <attribute name="name" type="string" use="required"/>
    <attribute name="systemc-name" type="string" use="required">
      <annotation>
        <documentation xml:lang="en">
          This attribute stores the SystemC hierarchical name.
        </documentation>
      </annotation>
    </attribute>
  </attributeGroup>
  <complexType name="abstract-object-type" abstract="true">
    <attributeGroup ref="target:object-identifier"/>
    <attribute name="type" type="string" use="required"/>
    <attribute name="address" type="string" use="required"/>
  </complexType>
  <complexType name="primitive-channel-type">
    <complexContent>
      <extension base="target:abstract-object-type">
        <attribute name="systemc-type" use="required">
          <simpleType>
            <restriction base="string">
              <enumeration value="sc_prim_channel"/>
              <enumeration value="sc_buffer"/>
              <enumeration value="sc_clock"/>
              <enumeration value="sc_signal"/>
              <enumeration value="sc_signal_resolved"/>
              <enumeration value="sc_signal_rv"/>
              <enumeration value="sc_fifo"/>
              <enumeration value="sc_mutex"/>
```

```
            <enumeration value="sc_semaphore"/>
          </restriction>
        </simpleType>
      </attribute>
    </extension>
  </complexContent>
</complexType>
<complexType name="to-type" abstract="true">
  <attribute name="to" use="required">
    <simpleType>
      <restriction base="string">
        <enumeration value="primitive-channel"/>
        <enumeration value="port"/>
        <enumeration value="hierarchical-channel"/>
      </restriction>
    </simpleType>
  </attribute>
  <attributeGroup ref="target:object-identifier"/>
</complexType>
<complexType name="bound-to-type">
  <complexContent>
    <extension base="target:to-type">
    </extension>
  </complexContent>
</complexType>
<complexType name="sensitive-to-type">
  <complexContent>
    <extension base="target:to-type">
      <attribute name="event">
        <simpleType>
          <restriction base="string">
            <enumeration value="UNKNOWN"/>
            <enumeration value="default"/>
            <enumeration value="any-change-of-value"/>
            <enumeration value="positive-edge"/>
            <enumeration value="negative-edge"/>
          </restriction>
        </simpleType>
      </attribute>
    </extension>
  </complexContent>
</complexType>
<complexType name="port-type">
  <complexContent>
    <extension base="target:abstract-object-type">
      <sequence>
        <element name="bound-to" type="target:bound-to-type" minOccurs=
            "0" maxOccurs="unbounded"/>
      </sequence>
      <attribute name="systemc-type" use="required">
        <simpleType>
          <restriction base="string">
            <enumeration value="sc_port"/>
```

```
            <enumeration value="sc_in"/>
            <enumeration value="sc_out"/>
            <enumeration value="sc_inout"/>
            <enumeration value="sc_in_resolved"/>
            <enumeration value="sc_out_resolved"/>
            <enumeration value="sc_inout_resolved"/>
            <enumeration value="sc_in_rv"/>
            <enumeration value="sc_out_rv"/>
            <enumeration value="sc_inout_rv"/>
            <enumeration value="sc_fifo_in"/>
            <enumeration value="sc_fifo_out"/>
          </restriction>
        </simpleType>
      </attribute>
    </extension>
  </complexContent>
</complexType>
<complexType name="export-type">
  <complexContent>
    <extension base="target:abstract-object-type">
      <sequence>
        <element name="bound-to" type="target:bound-to-type" minOccurs=
           "0"/>
      </sequence>
      <attribute name="systemc-type" use="required" type="string" fixed
         ="sc_export"/>
    </extension>
  </complexContent>
</complexType>
<complexType name="process-type">
  <complexContent>
    <extension base="target:abstract-object-type">
      <sequence>
        <element name="sensitive-to" type="target:sensitive-to-type"
           minOccurs="0" maxOccurs="unbounded"/>
        <element name="reset-to" minOccurs="0">
          <complexType>
            <attribute name="to" use="required">
              <simpleType>
                <restriction base="string">
                  <enumeration value="primitive-channel"/>
                  <enumeration value="port"/>
                </restriction>
              </simpleType>
            </attribute>
            <attribute name="active-level" use="required">
              <simpleType>
                <restriction base="string">
                  <enumeration value="high"/>
                  <enumeration value="low"/>
                </restriction>
              </simpleType>
            </attribute>
```

```
          <attributeGroup ref="target:object-identifier"/>
        </complexType>
      </element>
    </sequence>
    <attribute name="systemc-type" use="required">
      <simpleType>
        <restriction base="string">
          <enumeration value="sc_method"/>
          <enumeration value="sc_thread"/>
          <enumeration value="sc_cthread"/>
        </restriction>
      </simpleType>
    </attribute>
    <attribute name="function" type="string" use="required"/>
  </extension>
 </complexContent>
</complexType>
<complexType name="module-type">
 <complexContent>
  <extension base="target:abstract-object-type">
    <sequence>
      <element name="port" type="target:port-type" minOccurs="0"
         maxOccurs="unbounded"/>
      <element name="export" type="target:export-type" minOccurs="0"
         maxOccurs="unbounded"/>
      <element name="process" type="target:process-type" minOccurs="0
         " maxOccurs="unbounded"/>
      <element name="primitive-channel" type="target:primitive-
         channel-type" minOccurs="0" maxOccurs="unbounded"/>
      <element name="module" type="target:module-type"  minOccurs="0"
          maxOccurs="unbounded"/>
    </sequence>
    <attribute name="systemc-type" use="required" type="string" fixed
       ="sc_module"/>
  </extension>
 </complexContent>
</complexType>
<complexType name="real-constant-node-type">
 <attribute name="value" type="string" use="required"/>
 <attribute name="precision" type="positiveInteger" use="required"/>
 <attribute name="is-signed" type="boolean" use="required"/>
</complexType>
<complexType name="integer-constant-node-type">
 <attribute name="value" type="decimal" use="required"/>
 <attribute name="precision" type="positiveInteger" use="required"/>
 <attribute name="is-signed" type="boolean" use="required"/>
</complexType>
<complexType name="variable-node-type">
 <sequence minOccurs="0">
  <choice>
    <!-- only needed for array-element -->
    <element name="integer-constant" type="target:integer-constant-
       node-type"/>
```

```
      <!-- only needed for array-element or for phi -->
      <element name="variable" type="target:variable-node-type"
          maxOccurs="unbounded"/>
    </choice>
  </sequence>
  <attribute name="name" type="string" use="required"/>
  <attribute name="type">
    <simpleType>
      <restriction base="string">
        <enumeration value="data-member"/>
        <enumeration value="ssa-name"/>
        <enumeration value="local"/>
        <enumeration value="temporally"/>
        <enumeration value="phi"/>
        <enumeration value="array-element"/>
      </restriction>
    </simpleType>
  </attribute>
</complexType>
<group name="operand">
  <choice>
    <element name="variable" type="target:variable-node-type"/>
    <element name="real-constant" type="target:real-constant-node-type"
        />
    <element name="integer-constant" type="target:integer-constant-node
        -type"/>
  </choice>
</group>
<complexType name="unary-expression-node-type">
  <sequence>
    <group ref="target:operand"></group>
  </sequence>
  <attribute name="type" use="required">
    <simpleType>
      <restriction base="string">
        <enumeration value="real-to-fix-trunc"/>
        <enumeration value="integer-to-real"/>
        <enumeration value="negate"/>
        <enumeration value="abs"/>
        <enumeration value="bit-not"/>
        <enumeration value="truth-not"/>
        <enumeration value="paren"/>
        <enumeration value="convert"/>
        <enumeration value="addr-space-convert"/>
        <enumeration value="fixed-convert"/>
        <enumeration value="nop"/>
        <enumeration value="non-lvalue"/>
        <enumeration value="view-convert"/>
        <enumeration value="compound-literal"/>
        <enumeration value="save"/>
        <enumeration value="addr"/>
        <enumeration value="conj"/>
        <enumeration value="va-arg"/>
```

```
      </restriction>
    </simpleType>
  </attribute>
</complexType>
<complexType name="binary-expression-node-type">
  <sequence>
    <group ref="target:operand"></group>
    <group ref="target:operand"></group>
  </sequence>
  <attribute name="type" use="required">
    <simpleType>
      <restriction base="string">
        <enumeration value="plus"/>
        <enumeration value="minus"/>
        <enumeration value="mult"/>
        <enumeration value="pointer-plus"/>
        <enumeration value="trunc-div"/>
        <enumeration value="ceil-div"/>
        <enumeration value="floor-div"/>
        <enumeration value="round-div"/>
        <enumeration value="trunc-mod"/>
        <enumeration value="ceil-mod"/>
        <enumeration value="floor-mod"/>
        <enumeration value="round-mod"/>
        <enumeration value="rdiv"/>
        <enumeration value="exact_div"/>
        <enumeration value="min"/>
        <enumeration value="max"/>
        <enumeration value="lshift"/>
        <enumeration value="rshift"/>
        <enumeration value="lrotate"/>
        <enumeration value="rrotate"/>
        <enumeration value="bit-ior"/>
        <enumeration value="bit-xor"/>
        <enumeration value="bit-and"/>
        <enumeration value="less-than"/>
        <enumeration value="less-than-or-equal"/>
        <enumeration value="greater-than"/>
        <enumeration value="greater-than-or-equal"/>
        <enumeration value="equal"/>
        <enumeration value="not-equal"/>
        <enumeration value="truth-andif"/>
        <enumeration value="truth-orif"/>
        <enumeration value="truth-and"/>
        <enumeration value="truth-or"/>
        <enumeration value="truth-xor"/>
        <enumeration value="unordered"/>
        <enumeration value="ordered"/>
        <enumeration value="unordered-less-than"/>
        <enumeration value="unordered-less-than-or-equal"/>
        <enumeration value="unordered-greater-than"/>
        <enumeration value="unordered-greater-than-or-equal"/>
        <enumeration value="unordered-equal"/>
```

```
        <enumeration value="less-than-greater-than"/>
        <enumeration value="range"/>
        <enumeration value="fdesc"/>
        <enumeration value="complex"/>
        <enumeration value="predecrement"/>
        <enumeration value="preincrement"/>
        <enumeration value="postdecrement"/>
        <enumeration value="postincrement"/>
      </restriction>
    </simpleType>
  </attribute>
</complexType>
<complexType name="port-node-type">
  <sequence>
    <element name="variable" type="target:variable-node-type"/>
  </sequence>
</complexType>
<complexType name="target-node-type">
  <sequence>
    <element name="variable" type="target:variable-node-type"/>
  </sequence>
</complexType>
<complexType name="source-node-type">
  <sequence>
    <choice>
      <group ref="target:operand"></group>
      <choice>
        <element name="unary-expression" type="target:unary-expression-
            node-type"/>
        <element name="binary-expression" type="target:binary-
            expression-node-type"/>
      </choice>
    </choice>
  </sequence>
</complexType>
<complexType name="function-type">
  <sequence minOccurs="0" maxOccurs="unbounded">
    <choice maxOccurs="unbounded">
      <element name="basic-block">
        <complexType>
          <sequence minOccurs="0" maxOccurs="unbounded">
            <choice>
              <element name="wait">
                <complexType>
                  <attribute name="cycles" type="nonNegativeInteger"
                      use="required"/>
                </complexType>
              </element>
              <element name="read">
                <complexType>
                  <sequence>
                    <element name="target" type="target:target-node-
                        type"/>
```

```
            <element name="port" type="target:port-node-type"/>
          </sequence>
        </complexType>
      </element>
      <element name="write">
        <complexType>
          <sequence>
            <element name="port" type="target:port-node-type"/>
            <element name="source" type="target:source-node-
                type"/>
          </sequence>
        </complexType>
      </element>
      <element name="assign">
        <complexType>
          <sequence>
            <element name="target" type="target:target-node-
                type"/>
            <element name="source" type="target:source-node-
                type"/>
          </sequence>
        </complexType>
      </element>
    </choice>
  </sequence>
  <attribute name="identifier" type="string" use="required"/>
</complexType>
</element>
<element name="unknown">
  <complexType>
    <attribute name="type" type="string" use="required"/>
  </complexType>
</element>
<element name="edge">
  <complexType>
    <attribute name="source" type="string" use="required"/>
    <attribute name="destination" type="string" use="required"/>
    <attribute name="condition" type="string"/>
    <attribute name="value" type="boolean"/>
  </complexType>
</element>
<element name="condition">
  <complexType>
    <attribute name="type" use="required">
      <simpleType>
        <restriction base="string">
          <enumeration value="less-than"/>
          <enumeration value="less-than-or-equal"/>
          <enumeration value="greater-than"/>
          <enumeration value="greater-than-or-equal"/>
          <enumeration value="equal"/>
          <enumeration value="not-equal"/>
        </restriction>
```

```
          </simpleType>
        </attribute>
        <attribute name="identifier" type="string" use="required"/>
      </complexType>
    </element>
   </choice>
  </sequence>
  <attribute name="name" type="string" use="required"/>
 </complexType>
 <complexType name="hierarchy-type">
  <sequence>
   <element name="primitive-channel" type="target:primitive-channel-
       type" minOccurs="0" maxOccurs="unbounded"/>
   <element name="module" type="target:module-type" minOccurs="0"
       maxOccurs="unbounded"/>
  </sequence>
 </complexType>
 <complexType name="behavior-type">
  <sequence>
   <element name="function" type="target:function-type" minOccurs="0"
       maxOccurs="unbounded"/>
  </sequence>
 </complexType>
 <element name="systemc-model">
  <complexType>
   <sequence>
    <element name="hierarchy" type="target:hierarchy-type"/>
    <element name="behavior" type="target:behavior-type"/>
   </sequence>
   <attribute name="name" use="required"/>
  </complexType>
 </element>
</schema>
```

Resource Directory for SCMDL

A Resource Directory Description Language (RDDL) document, which is called a resource directory, is used to describe the XML namespace http://shabe.sourceforge.net/systemc-model and refers to the XML Schema for the SCMDL. This RDDL document can be found at http://shabe.sourceforge.net/systemc-model and is shown below.

```
<!DOCTYPE html PUBLIC "-//XML-DEV//DTD XHTML RDDL 1.0//EN"
"http://www.rddl.org/rddl-xhtml.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns:rddl="http://www.rddl.org/">
  <head>
    <title>
      Resource Directory Description Language (RDDL) Document
      for SystemC Model Description Language (SCMDL)
    </title>
  </head>
  <body>
    <h1>
      Resource Directory Description Language (RDDL) Document
      for SystemC Model Description Language (SCMDL).
    </h1>
    <p>
      This document describes an XML format to describe SystemC Models.
    </p>
    <rddl:resource
      xlink:role="http://www.w3.org/2000/10/XMLSchema"
      xlink:arcrole="http://www.rddl.org/purposes#schema-validation"
      xlink:href="http://shabe.sourceforge.net/systemc-model/systemc-
          model.xsd"
      xlink:title="The SystemC Model Description Language">
      <p>
        The XML Schema for SCMDL documents is <a href="systemc-model.xsd"
            > available from here</a>.
        The documentation for this schema can be found <a href="systemc-
            model.html">here</a>.
      </p>
    </rddl:resource>
    <hr />
    Copyright 2010 <a href="http://bd.eduweb.hhs.nl/">Harry Broeders</a><
        br />
    Last Modified November 25, 2010
  </body>
</html>
```

Test Cases

F

A selection of test cases is presented in Chapter 5. Some other test cases are presented in this appendix and all others can be found at http://shabe.sourceforge.net/test_programs.

A useful implementation for a static array of ports is given in Figure F.1: a generic $N : 1$ multiplexer. The data type of the signals which must be multiplexed and the number of inputs (N) are declared as template parameters of this module. The number of bits needed for the select input is calculated at compile time by using the function `Log2` which is not shown here. This function is implemented using a technique called template metaprogramming [1] and calculates $\lceil log_2 N \rceil$.

```
template <typename T, size_t N>
SC_MODULE(Mux) {
    sc_in<T> in[N];
    sc_in<sc_uint<Log2<N>::result> > select;
    sc_out<T> out;
    SC_CTOR(Mux) {
        SC_METHOD(run);
        for (size_t i(0); i<N; ++i)
            sensitive << in[i];
        sensitive << select;
    }
private:
    void run() {
        out.write(
            select.read()<N ?
            in[select.read()].read() :
            in[N-1].read()
        );
    }
};
```

Figure F.1: A generic $N : 1$ multiplexer.

Figure F.2 shows how this generic multiplexer can be used to instantiate a $3 : 1$ multiplexer for `int`s. The expected output in SCMDL for this instantiation is also shown.

It is also possible to create ports in the base classes of a module. This can be a useful coding idiom. Figure F.3 shows a test case in which a module `ResetableDff` is derived from a base module `SynchronousModule` and from a mixin class `ResetPin`. Both base classes declare a port which is inherited by the `ResettableDFF` class. Using the base module `SynchronousModule` as a base class for all synchronous modules in a

```
Mux<int, 3> mux("mux");
```

```
<module name="mux" type="Mux&lt;int, 3ul&gt;" systemc-name="mux"
    systemc-type="sc_module">
    <port name="in[0]" type="sc_core::sc_in&lt;int&gt;" systemc-
        name="mux.port_0" systemc-type="sc_in" />
    <port name="in[1]" type="sc_core::sc_in&lt;int&gt;" systemc-
        name="mux.port_1" systemc-type="sc_in" />
    <port name="in[2]" type="sc_core::sc_in&lt;int&gt;" systemc-
        name="mux.port_2" systemc-type="sc_in" />
    <port name="select" type="sc_core::sc_in&lt;sc_dt::sc_uint&lt
        ;2&gt; &gt;" systemc-name="mux.port_3" systemc-type="sc_in
        " />
    <port name="out" type="sc_core::sc_out&lt;int&gt;" systemc-
        name="mux.port_4" systemc-type="sc_out" />
    <process name="run" type="Mux&lt;int, 3ul&gt;::run()" systemc
        -name="mux.run" systemc-type="sc_method" function="
        Mux::run">
         <sensitive-to to="port" name="in[0]" systemc-name="mux.
             port_0" event="default" />
         <sensitive-to to="port" name="in[1]" systemc-name="mux.
             port_1" event="default" />
         <sensitive-to to="port" name="in[2]" systemc-name="mux.
             port_2" event="default" />
         <sensitive-to to="port" name="select" systemc-name="mux.
             port_3" event="default" />
    </process>
</module>
```

Figure F.2: A submodule can be created and stored in a base module.

system has the advantage that the code which declares the clock input, the process, and its sensitivity only has to be declared once. If we want the system to be clocked on the negative edge instead of the positive edge of the clock then we only have to change a single line of code. Using the mixin base class `ResetPin` as a base class for all resettable modules in a system has the advantage that the code which declares the reset input only has to be declared once. If we want the reset input to be a `sc_login` input instead of a `bool` input then we only have to change a single line of code. Multiple inheritance from the classes derived `sc_object` is not permitted in SystemC [58] p. 93. Therefore we must declare the mixin class as a separate class which does not inherits from `sc_module`.

The UML diagram, Figure F.4, shows the classes declared in Figure F.3 and their inheritance relationships.

Figure F.5 an instantiation of a `ResetableDff`. The expected output in SCMDL for this instantiation is also shown.

```
SC_MODULE(SynchronousModule) {
    sc_in_clk clk;
    SC_CTOR(SynchronousModule) {
        SC_CTHREAD(run, clk.pos());
    }
private:
    virtual void run() = 0;
};
struct ResetPin {
    sc_in<bool> reset;
};
template <typename T>
struct ResettableDff: public SynchronousModule, public ResetPin {
    sc_in<T> din;
    sc_out<T> dout;
    ResettableDff(const sc_module_name& nm): SynchronousModule(nm) {
    }
private:
    virtual void run();
};
```

Figure F.3: A module which inherits ports from two different base classes.

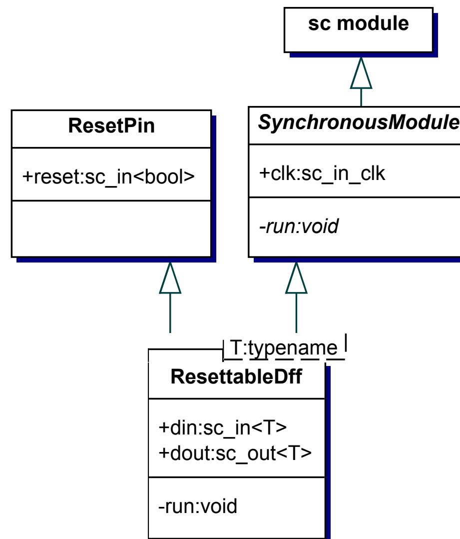


Figure F.4: The UML diagram for the test case shown in Figure F.3.

```
ResettableDff<int> r("r");
```

```
<module name="r" type="ResettableDff&lt;int&gt;" systemc-name="r"
     systemc-type="sc_module">
    <port name="clk" type="sc_core::sc_in&lt;bool&gt;" systemc-
       name="r.port_0" systemc-type="sc_in" />
    <port name="reset" type="sc_core::sc_in&lt;bool&gt;" systemc-
       name="r.port_1" systemc-type="sc_in" />
    <port name="din" type="sc_core::sc_in&lt;int&gt;" systemc-
       name="r.port_2" systemc-type="sc_in" />
    <port name="dout" type="sc_core::sc_out&lt;int&gt;" systemc-
       name="r.port_3" systemc-type="sc_out" />
        <process name="run" type="ResettableDff&lt;int&gt;::run()
           " systemc-name="resettableDff.run" systemc-type="
           sc_cthread" function="ResettableDff::run">
            <sensitive-to to="port" name="clk" systemc-name="r.
               port_0" event="positive-edge" />
        </process>
</module>
```

Figure F.5: An instantiation of the module `ResettableDff` which is declared in Figure F.3.

Parser for GDB/MI Output Records

G

The UML class diagram of the parser for GDB/MI output records is shown in Figure G.1. All private member functions and data members are hidden to prevent clutter. The composite design pattern [39] is used to create `ListOfResults` which can contain `ListOfResults`.

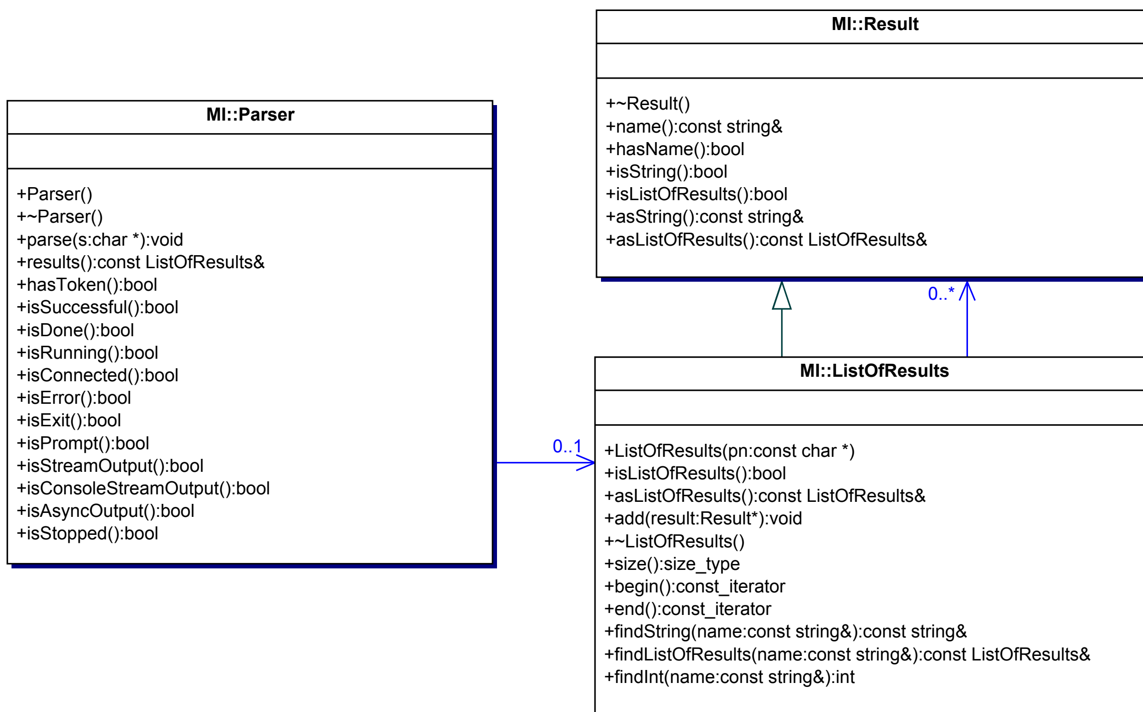


Figure G.1: The primitive channels which are defined in the synthesizable subset of SystemC.

Transformation of SCMDL documents using XSLT

As an example, an eXtensible Stylesheet Language Transformations (XSLT) stylesheet is written which can transform the hierarchical module structure described in SystemC Model Description Language (SCMDL) into GraphML. The SCMDL document for the FIR filter which is instantiated in Figure 2.7 can be produced by SHaBE. Using the XSLT stylesheet presented below this SCMDL document can be transformed into an GraphML document which can be viewed in the graph editor yEd[1]. This GraphML document is shown in Figure H.1. We have used the Saxon XSLT processor[2] to execute the transformation.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns="http://graphml.graphdrawing.org/xmlns"
  xmlns:y="http://www.yworks.com/xml/graphml">
  <xsl:output method="xml" indent="yes"/>
  <xsl:strip-space elements="*"/>
  <xsl:template match="/">
    <graphml xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns http:
            //www.yworks.com/xml/schema/graphml/1.1/ygraphml.xsd">
      <key for="node" id="d0" yfiles.type="nodegraphics"/>
      <xsl:element xmlns:scm="http://shabe.sourceforge.net/systemc-model"
          name="graph">
        <xsl:attribute name="edgedefault">
          <xsl:value-of>undirected</xsl:value-of>
        </xsl:attribute>
        <xsl:attribute name="id">
          <xsl:copy-of select="/scm:systemc-model/@name"/>
        </xsl:attribute>
        <xsl:apply-templates select="/scm:systemc-model/scm:hierarchy/
           scm:module"/>
      </xsl:element>
    </graphml>
  </xsl:template>
  <xsl:template xmlns:scm="http://shabe.sourceforge.net/systemc-model"
     match="scm:module">
    <xsl:param name="basename"/>
    <xsl:element name="node">
      <xsl:variable name="id" select="concat($basename, 'n', position()
         -1)"/>
      <xsl:attribute name="id">
```

[1] http://www.yworks.com/en/products_yed_about.html

[2] http://saxon.sourceforge.net/

```
  <xsl:value-of select="$id"/>
</xsl:attribute>
<data key="d0">
  <y:ProxyAutoBoundsNode>
    <y:Realizers active="0">
      <y:GroupNode>
        <xsl:if test="not(scm:module)">
          <!-- module is a leaf -->
          <xsl:element name="y:Geometry">
            <xsl:attribute name="height">
              <xsl:value-of>80</xsl:value-of>
            </xsl:attribute>
            <xsl:attribute name="width">
              <xsl:value-of>100</xsl:value-of>
            </xsl:attribute>
            <xsl:attribute name="x">
              <!-- pos must be a simple counter -->
              <xsl:variable name="pos">
                <xsl:number level="any"/>
              </xsl:variable>
              <!--<xsl:value-of select="$pos*200"/>-->
              <xsl:value-of select="count(preceding::scm:module[not
                  (scm:module)])*200"/>
            </xsl:attribute>
            <xsl:attribute name="y">
              <xsl:value-of>0</xsl:value-of>
            </xsl:attribute>
          </xsl:element>
        </xsl:if>
        <y:Fill color="#F8ECC9" transparent="false"/>
        <y:NodeLabel alignment="right" autoSizePolicy="node_width"
            backgroundColor="#404040" fontFamily="Dialog" fontSize="
            16" fontStyle="plain" hasLineColor="false" modelName="
            internal" modelPosition="t" textColor="#FFFFFF" visible=
            "true">
          <xsl:value-of select="@name"/>
        </y:NodeLabel>
        <y:State closed="false" innerGraphDisplayEnabled="false"/>
      </y:GroupNode>
      <y:GroupNode>
        <y:Fill color="#F8ECC9" transparent="false"/>
        <y:NodeLabel alignment="right" autoSizePolicy="node_width"
            backgroundColor="#404040" fontFamily="Dialog" fontSize="
            16" fontStyle="plain" hasLineColor="false" modelName="
            internal" modelPosition="t" textColor="#FFFFFF" visible=
            "true">
          <xsl:value-of select="@name"/>
        </y:NodeLabel>
        <y:State closed="true" innerGraphDisplayEnabled="false"/>
      </y:GroupNode>
    </y:Realizers>
  </y:ProxyAutoBoundsNode>
</data>
```

```
        <xsl:element name="graph">
          <xsl:attribute name="edgedefault">
            <xsl:value-of>undirected</xsl:value-of>
          </xsl:attribute>
          <xsl:attribute name="id">
            <xsl:value-of select="concat($id, ':')"/>
          </xsl:attribute>
          <xsl:apply-templates select="scm:module">
            <xsl:with-param name="basename" select="concat($id, '::')"/>
          </xsl:apply-templates>
        </xsl:element>
      </xsl:element>
    </xsl:template>
</xsl:stylesheet>
```

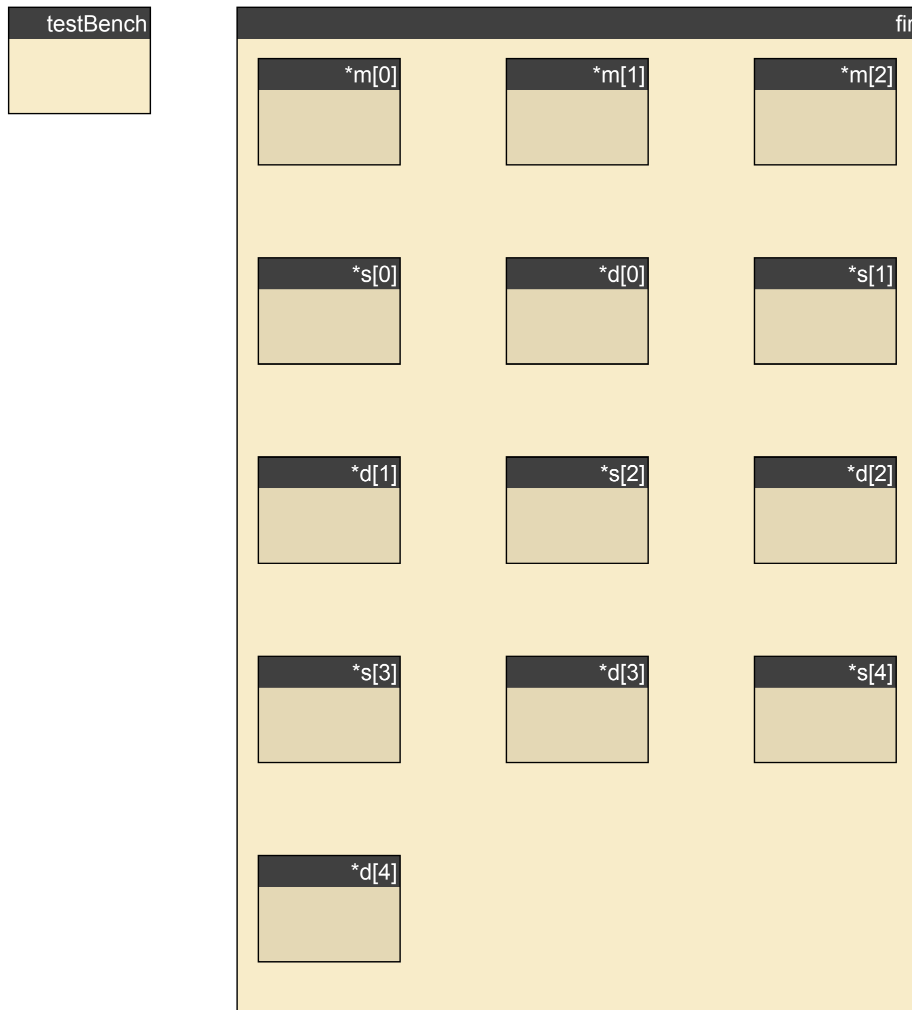
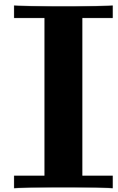


Figure H.1: The module hierarchy of the FIR filter which is instantiated in Figure 2.7 as shown by yEd.

An SCMDL document produced by SHaBE.



In this appendix a SystemC model of a simple FIR filter is shown and the SCMDL documents produced for this model is presented. The model, shown below, consists of the module `Test_FIR`, the module `FIR` and the function `sc_main`. The module `Test_FIR` models the test bench. It produces a sample and reads in the result on every negative edge of the clock signal. The behavior of the test bench is described in an `SC_THREAD` process. The member function `behavior` which is associated with this process can, for example, generate an impulse or a step function and check the responses. The implementation of this function is not shown here to prevent clutter. The module `FIR` models the `FIR` filter, the delay elements are implemented as `sc_buffer` member variables. In the `sc_main` function a `sc_clock` instance of each module is created

```
#include <systemc>
using namespace sc_core;

SC_MODULE(Test_FIR) {
    sc_in_clk clk;
    sc_out<double> sample;
    sc_in<double> result;
    SC_CTOR(Test_FIR) {
        SC_THREAD(behavior);
        sensitive << clk.neg();
    }
private:
    void behavior();
};

SC_MODULE(FIR) {
    sc_in_clk clk;
    sc_in<double> sample;
    sc_out<double> result;
    SC_CTOR(FIR) {
        SC_METHOD(behavior);
        sensitive << clk.neg();
    }
private:
    sc_buffer<double> i1, i2, i3, i4, i5;
    void behavior() {
        result.write(
            -0.07556556070608 * sample.read() +
             0.09129209297815 * i1.read() +
             0.47697917208036 * i2.read() +
             0.47697917208036 * i3.read() +
             0.09129209297815 * i4.read() +
            -0.07556556070608 * i5.read()
```

```
        );
        i1.write(sample.read());
        i2.write(i1.read());
        i3.write(i2.read());
        i4.write(i3.read());
        i5.write(i4.read());
    }
};

int sc_main (int argc , char *argv[]) {
    sc_clock clock("clock", 10, SC_NS);
    sc_signal<double> sample;
    sc_signal<double> result;

    Test_FIR testBench("testBench");
    testBench.clk(clock.signal());
    testBench.sample(sample);
    testBench.result(result);

    FIR fir("fir");
    fir.clk(clock.signal());
    fir.sample(sample);
    fir.result(result);

    sc_start();
    return 0;
}
```

This SystemC program must be compiled and linked with the debug version of the OSCI SystemC library version 2.2.0. If this executable is called `fir_example1` then the hierarchical and behavioral information from the model can be extracted by using the command `shabe fir_example1`. The SCMDL document produced by SHaBE is shown below.

```
<?xml version="1.0" encoding="UTF-8" ?>
<systemc-model xmlns="http://shabe.sourceforge.net/systemc-model"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://shabe.sourceforge.net/systemc-model http://
    shabe.sourceforge.net/systemc-model/systemc-model.xsd" name="fir_2">
  <hierarchy>
    <primitive-channel name="clock" type="sc_core::sc_clock" systemc-name
        ="clock" systemc-type="sc_clock" address="0x7fffffffde70" />
    <primitive-channel name="sample" type="sc_core::sc_signal&lt;double&
        gt;" systemc-name="signal_0" systemc-type="sc_signal" address="0
        x7fffffffe0a0" />
    <primitive-channel name="result" type="sc_core::sc_signal&lt;double&
        gt;" systemc-name="signal_1" systemc-type="sc_signal" address="0
        x7fffffffe020" />
    <module name="testBench" type="Test_FIR" systemc-name="testBench"
        systemc-type="sc_module" address="0x7fffffffdc80">
      <port name="clk" type="sc_core::sc_in&lt;bool&gt;" systemc-name="
          testBench.port_0" systemc-type="sc_in" address="0x7fffffffdd38">
```

```
      <bound−to to="primitive-channel" name="clock" systemc−name="clock
         " />
    </port>
    <port name="sample" type="sc_core::sc_out&lt;double&gt;" systemc−
       name="testBench.port_1" systemc−type="sc_out" address="0
       x7fffffffdda8">
      <bound−to to="primitive-channel" name="sample" systemc−name="
         signal_0" />
    </port>
    <port name="result" type="sc_core::sc_in&lt;double&gt;" systemc−
       name="testBench.port_2" systemc−type="sc_in" address="0
       x7fffffffde10">
      <bound−to to="primitive-channel" name="result" systemc−name="
         signal_1" />
    </port>
    <process name="behavior" type="Test_FIR::behavior()" systemc−name="
       testBench.behavior" systemc−type="sc_thread" address="0x706950"
       function="Test_FIR::behavior">
      <sensitive−to to="port" name="clk" systemc−name="testBench.port_0
         " event="negative-edge" />
    </process>
  </module>
  <module name="fir" type="FIR" systemc−name="fir" systemc−type="
     sc_module" address="0x7fffffffd830">
    <port name="clk" type="sc_core::sc_in&lt;bool&gt;" systemc−name="
       fir.port_0" systemc−type="sc_in" address="0x7fffffffd8e8">
      <bound−to to="primitive-channel" name="clock" systemc−name="clock
         " />
    </port>
    <port name="sample" type="sc_core::sc_in&lt;double&gt;" systemc−
       name="fir.port_1" systemc−type="sc_in" address="0x7fffffffd958">
      <bound−to to="primitive-channel" name="sample" systemc−name="
         signal_0" />
    </port>
    <port name="result" type="sc_core::sc_out&lt;double&gt;" systemc−
       name="fir.port_2" systemc−type="sc_out" address="0x7fffffffd9b8"
       >
      <bound−to to="primitive-channel" name="result" systemc−name="
         signal_1" />
    </port>
    <process name="behavior" type="FIR::behavior()" systemc−name="fir.
       behavior" systemc−type="sc_method" address="0x709160" function="
       FIR::behavior">
      <sensitive−to to="port" name="clk" systemc−name="fir.port_0"
         event="negative-edge" />
    </process>
    <primitive−channel name="i1" type="sc_core::sc_buffer&lt;double&gt;
       " systemc−name="fir.buffer_0" systemc−type="sc_buffer" address="
       0x7fffffffda20" />
    <primitive−channel name="i2" type="sc_core::sc_buffer&lt;double&gt;
       " systemc−name="fir.buffer_1" systemc−type="sc_buffer" address="
       0x7fffffffda98" />
```

```
    <primitive-channel name="i3" type="sc_core::sc_buffer&lt;double&gt;
        " systemc-name="fir.buffer_2" systemc-type="sc_buffer" address="
        0x7fffffffdb10" />
    <primitive-channel name="i4" type="sc_core::sc_buffer&lt;double&gt;
        " systemc-name="fir.buffer_3" systemc-type="sc_buffer" address="
        0x7fffffffdb88" />
    <primitive-channel name="i5" type="sc_core::sc_buffer&lt;double&gt;
        " systemc-name="fir.buffer_4" systemc-type="sc_buffer" address="
        0x7fffffffdc00" />
  </module>
</hierarchy>
<behavior>
  <function name="FIR::behavior">
    <basic-block identifier="0x7f710ae43680">
      <read>
        <target>
          <variable type="ssa-name" name="3" />
        </target>
        <port>
          <variable type="data-member" name="sample" />
        </port>
      </read>
      <assign>
        <target>
          <variable type="ssa-name" name="6" />
        </target>
        <source>
          <binary-expression type="mult">
            <variable type="ssa-name" name="3" />
            <real-constant value="-7.556556070608e-2" precision="64" is
                -signed="true" />
          </binary-expression>
        </source>
      </assign>
      <read>
        <target>
          <variable type="ssa-name" name="8" />
        </target>
        <port>
          <variable type="data-member" name="i1" />
        </port>
      </read>
      <assign>
        <target>
          <variable type="ssa-name" name="10" />
        </target>
        <source>
          <binary-expression type="mult">
            <variable type="ssa-name" name="8" />
            <real-constant value="9.129209297815e-2" precision="64" is-
                signed="true" />
          </binary-expression>
        </source>
```

```
</assign>
<assign>
  <target>
    <variable type="ssa-name" name="11" />
  </target>
  <source>
    <binary-expression type="plus">
      <variable type="ssa-name" name="6" />
      <variable type="ssa-name" name="10" />
    </binary-expression>
  </source>
</assign>
<read>
  <target>
    <variable type="ssa-name" name="13" />
  </target>
  <port>
    <variable type="data-member" name="i2" />
  </port>
</read>
<assign>
  <target>
    <variable type="ssa-name" name="15" />
  </target>
  <source>
    <binary-expression type="mult">
      <variable type="ssa-name" name="13" />
      <real-constant value="4.7697917208036e-1" precision="64" is
         -signed="true" />
    </binary-expression>
  </source>
</assign>
<assign>
  <target>
    <variable type="ssa-name" name="16" />
  </target>
  <source>
    <binary-expression type="plus">
      <variable type="ssa-name" name="11" />
      <variable type="ssa-name" name="15" />
    </binary-expression>
  </source>
</assign>
<read>
  <target>
    <variable type="ssa-name" name="18" />
  </target>
  <port>
    <variable type="data-member" name="i3" />
  </port>
</read>
<assign>
  <target>
```

```
      <variable type="ssa-name" name="20" />
    </target>
    <source>
      <binary-expression type="mult">
        <variable type="ssa-name" name="18" />
        <real-constant value="4.7697917208036e-1" precision="64" is
            -signed="true" />
      </binary-expression>
    </source>
  </assign>
  <assign>
    <target>
      <variable type="ssa-name" name="21" />
    </target>
    <source>
      <binary-expression type="plus">
        <variable type="ssa-name" name="16" />
        <variable type="ssa-name" name="20" />
      </binary-expression>
    </source>
  </assign>
  <read>
    <target>
      <variable type="ssa-name" name="23" />
    </target>
    <port>
      <variable type="data-member" name="i4" />
    </port>
  </read>
  <assign>
    <target>
      <variable type="ssa-name" name="25" />
    </target>
    <source>
      <binary-expression type="mult">
        <variable type="ssa-name" name="23" />
        <real-constant value="9.129209297815e-2" precision="64" is-
            signed="true" />
      </binary-expression>
    </source>
  </assign>
  <assign>
    <target>
      <variable type="ssa-name" name="26" />
    </target>
    <source>
      <binary-expression type="plus">
        <variable type="ssa-name" name="21" />
        <variable type="ssa-name" name="25" />
      </binary-expression>
    </source>
  </assign>
  <read>
```

```
    <target>
      <variable type="ssa-name" name="28" />
    </target>
    <port>
      <variable type="data-member" name="i5" />
    </port>
  </read>
  <assign>
    <target>
      <variable type="ssa-name" name="30" />
    </target>
    <source>
      <binary-expression type="mult">
        <variable type="ssa-name" name="28" />
        <real-constant value="-7.556556070608e-2" precision="64" is
            -signed="true" />
      </binary-expression>
    </source>
  </assign>
  <assign>
    <target>
      <variable type="ssa-name" name="31" />
    </target>
    <source>
      <binary-expression type="plus">
        <variable type="ssa-name" name="26" />
        <variable type="ssa-name" name="30" />
      </binary-expression>
    </source>
  </assign>
  <write>
    <port>
      <variable type="data-member" name="result" />
    </port>
    <source>
      <variable type="ssa-name" name="31" />
    </source>
  </write>
  <read>
    <target>
      <variable type="ssa-name" name="34" />
    </target>
    <port>
      <variable type="data-member" name="sample" />
    </port>
  </read>
  <write>
    <port>
      <variable type="data-member" name="i1" />
    </port>
    <source>
      <variable type="ssa-name" name="34" />
    </source>
```

```
</write>
<read>
  <target>
    <variable type="ssa-name" name="38" />
  </target>
  <port>
    <variable type="data-member" name="i1" />
  </port>
</read>
<write>
  <port>
    <variable type="data-member" name="i2" />
  </port>
  <source>
    <variable type="ssa-name" name="38" />
  </source>
</write>
<read>
  <target>
    <variable type="ssa-name" name="41" />
  </target>
  <port>
    <variable type="data-member" name="i2" />
  </port>
</read>
<write>
  <port>
    <variable type="data-member" name="i3" />
  </port>
  <source>
    <variable type="ssa-name" name="41" />
  </source>
</write>
<read>
  <target>
    <variable type="ssa-name" name="44" />
  </target>
  <port>
    <variable type="data-member" name="i3" />
  </port>
</read>
<write>
  <port>
    <variable type="data-member" name="i4" />
  </port>
  <source>
    <variable type="ssa-name" name="44" />
  </source>
</write>
<read>
  <target>
    <variable type="ssa-name" name="47" />
  </target>
```

```
        <port>
          <variable type="data-member" name="i4" />
        </port>
      </read>
      <write>
        <port>
          <variable type="data-member" name="i5" />
        </port>
        <source>
          <variable type="ssa-name" name="47" />
        </source>
      </write>
    </basic-block>
    <edge source="0x7f710ae43680" destination="0x7f710ae43618" />
  </function>
  <function name="Test_FIR::behavior">
    <!-- REMOVED -->
  </function>
</behavior>
</systemc-model>
```

As can be seen above all hierarchical and behavioral information is retrieved from the model. Using this SCMDL document the structure of the model and the AST which describes its behavior can be drawn as shown in Figure I.1 respectively Figure I.2. The AST is only partially drawn.

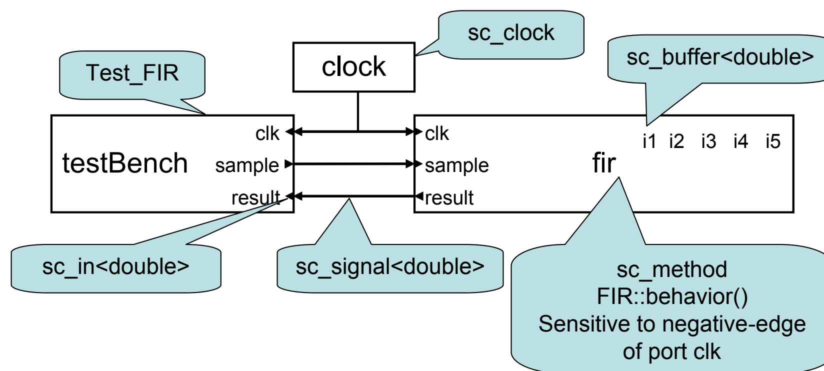


Figure I.1: The module structure used to test the FIR filter.

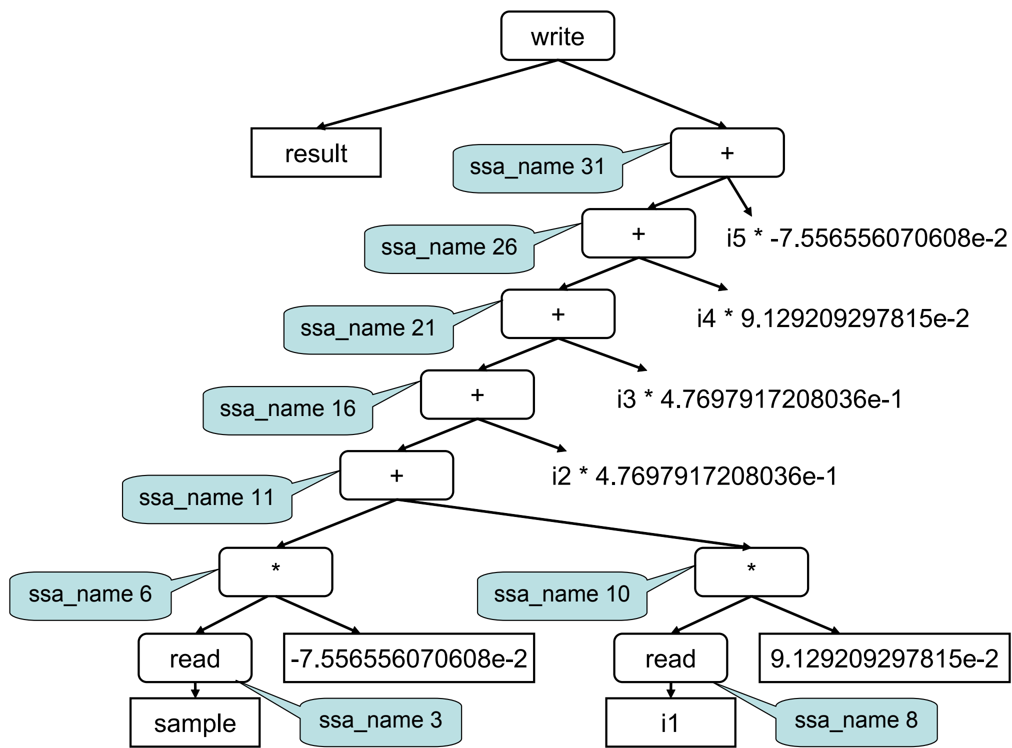


Figure I.2: Part of the AST of the behavior function of the FIR filter.

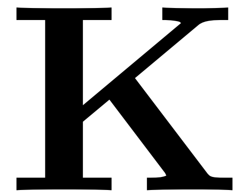
Execution Time of SHaBE J

Table J.1 shows the execution time of SHaBE for the FIR filter presented in Figure 2.6 for different values of the template parameter `ORDER` which specifies the order of the filter. All execution times were measured on a PC equipped with 2 GB of RAM and an AMD Athlon 64 processor running at 2 GHz by using the Unix command `time`. The execution time was measured 5 times for each value of ORDER and the average value and the sample standard derivation s were calculated.

Table J.1: The execution time of SHaBE for different orders of the FIR filter presented in Figure 2.6.

`ORDER`	# SystemC objects	Execution time (s)						s
		1	2	3	4	5	average	
10	168	8.19	8.03	7.96	8.04	7.98	8.04	0.09
20	323	9.82	9.79	9.79	9.79	9.78	9.79	0.02
30	478	11.67	11.64	11.62	11.64	11.61	11.64	0.02
40	633	13.49	13.42	13.49	13.45	13.49	13.46	0.03
50	788	15.21	15.19	15.26	15.18	15.24	15.22	0.03
60	943	17.12	17.42	17.03	17.20	17.09	17.17	0.15
70	1098	19.02	18.98	18.71	19.01	18.86	18.91	0.13
80	1253	20.71	20.68	20.53	20.59	20.75	20.65	0.09
90	1408	22.40	22.51	22.53	22.51	22.58	22.51	0.07
100	1563	24.36	24.35	24.43	24.54	24.37	24.41	0.08
110	1718	26.26	26.23	26.07	25.89	25.90	26.07	0.17
120	1873	27.95	27.98	27.89	27.93	27.87	27.92	0.04
130	2028	29.78	29.95	29.89	29.85	29.74	29.84	0.08
140	2183	31.79	31.85	31.79	31.81	31.61	31.77	0.09
150	2338	33.48	33.49	33.49	33.50	33.70	33.53	0.09
160	2493	35.20	35.20	35.25	35.16	35.28	35.22	0.05
170	2648	37.11	37.10	36.94	36.99	37.20	37.07	0.10
180	2803	39.12	39.09	38.86	38.96	38.76	38.96	0.15
190	2958	40.69	40.76	40.84	40.68	40.67	40.73	0.07
200	3113	42.83	42.85	42.44	42.98	42.60	42.74	0.22

SHaBE versus PinaVM



During the time we were developing SHaBE an other SystemC front-end called PinaVM was introduced, see Section 3.1.3.3. The developers of PinaVM and we have, independently from each other, concluded that the hybrid approach, see Section 3.1.3, is the only viable approach if we do not want to restrict the SystemC code which can be used during the elaboration phase. PinaVM and SHaBE are the only freely available open-source SystemC front-end which take this hybrid approach. There are some more similarities between them.

PinaVM and SHaBE both:

- describe the behavior of the model in an AST in SSA form.
- recognize the calls into the SystemC library and analyze the arguments used in these calls.

The differences between PinaVM and SHaBE are:

- PinaVM uses LLVM-GCC to retrieve the ASTs, while SHaBE uses its own GCC plug-in called SHaBEPlugIn.
- PinaVM stores the information retrieved from the model in LLVM IR, while SHaBE stores the information in an SCMDL document which is based on XML. The XML format will be much easier to use for a tool based on the front-end because it is must closer to the SystemC vocabulary than LLVM IR which are instructions for a virtual machine. In addition, SCMDL can much easier be read by humans.
- Because SHaBE uses the debug version of the SystemC library, it can identify the calls into the SystemC library by their qualified names. PinaVM uses LLVM to extract the ASTs which describe the behavior and therefore it must find the calls into the SystemC library in the LLVM IR. Because this is compiled code, the names used in the LLVM IR are mangled. This means that the types of the arguments are encoded in the function names. For example, PinaVM can recognize a call to the `read` member function of a `sc_signal<int>` by looking for a call to the function with the mangled name `_ZNK7sc_core5sc_inIiE4readEv` [72]. But, to recognize a call to the `read` member function of a `sc_signal<double>` it has to look for a different mangled name. In fact it is impossible for PinaVM to recognize a call to the `read` member function of a `sc_signal<user_defined_type>` in this way, because the mangled name of this type can not be predicted in advance. The authors of PinaVM propose a solution to this problem in [72] which is not implemented yet as far as we know. SHaBE can recognize a call to the `read` member function of any class in the SystemC library (not only `sc_signal`s but

also `sc_ports`, etc) by simply checking if the fully qualified function name start with `sc_core::` and ends with `::read`.

- To link the hierarchical information with the behavioral information PinaVM needs to determine the value of certain arguments which are used in the calls into the SystemC library. It uses the novel approach of using just-in-time compilation to execute the code fragments which determine these values. This approach is, according to the authors of PinaVM, limited to models in which the ports which are being used in the behavioral description can be determined statically. SHaBE on the other hand has extracted the C++ names of the object which can be used as arguments for these calls into the SystemC library during the execution of the elaboration phase. Therefore it can simply link the behavioral information with the hierarchical information by using the fully qualified C++ names of these arguments.
- PinaVM needs a slightly modified version of the SystemC library. SHaBE can use the SystemC library without modifications.
- PinaVM can be used with any (slightly modified) implementation of the SystemC standard, while SHaBE can only be used with the OSCI implementation version 2.2.0.