THESIS ON INFORMATICS AND SYSTEM ENGINEERING C111

Comprehensive Abstraction of VHDL RTL Cores to ESL SystemC

SYED SAIF ABRAR

TUT PRESS

TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology
Department of Computer Engineering

This dissertation was accepted for the defence of the degree of Doctor of Philosophy in Computer and Systems Engineering on March 15, 2016.

Supervisors: Dr. Maksim Jenihhin, Prof. Jaan Raik
Department of Computer Engineering
Tallinn University of Technology, Tallinn, Estonia

Opponents: Prof. Zainalabedin Navabi,
School of Electrical and Computer Engineering
University of Tehran, Tehran, Iran
Department of Electrical and Computer Engineering
Worcester Polytechnic Institute, USA

Dr. Alexander Kamkin
Institute for System Programming of the Russian Academy of Sciences, Moscow, Russia

Defence of the thesis: April 20, 2016

Declaration:
Hereby I declare that this doctoral thesis, my original investigation and achievement, submitted for the doctoral degree at Tallinn University of Technology has not been submitted for any academic degree.

Copyright: Syed, Saif Abrar, 2016
ISSN 1406-4731
ISBN 978-9949-23-913-9 (publication)
ISBN 978-9949-23-914-6 (PDF)

INFORMAATIKA JA SÜSTEEMITEHNIKA C111

Register-siirde taseme VHDL kirjelduste kompleksne abstraheerimine süsteemitaseme SystemC mudeliteks

SYED SAIF ABRAR

TTÜ
KIRJASTUS

TABLE OF CONTENTS

LIST OF PUBLICATIONS

The work of this thesis, with author's contributions, is based on the following publications

[A] Syed, Saif Abrar; Jenihhin, Maksim; Raik, Jaan. "FSMD RTL Design Manipulation for Clock Interface Abstraction." IEEE International Conference on Advances in Computing, Communications and Informatics (ICACCI), Kochi, India, 2015. pp 1-6.

The author identified the need for clock interface abstraction and worked on manipulation strategy of Finite State Machine with Datapath (FSMD) designs to Algorithmic State Machine (ASM) charts. Once formalized, the author applied these steps on various FSMD designs to arrive at their ASM representations. Then the author defined the rules for implementing the ASM charts in SystemC such that simulation speed is optimized. Various benchmarks in ASM were then implemented in SystemC by the author. Based on these SystemC implementations, the author performed experiments to measure and compare the simulation speeds of FSMD and clock-abstracted designs. The author wrote a paper describing the approach as well as its advantages, and then presented it in the IEEE International Conference on Advances in Computing, Communications and Informatics (2015).

[B] Syed, Saif Abrar; Jenihhin, Maksim; Raik, Jaan. "SystemC-Based Loose Models for Simulation Speed-Up by Abstraction of RTL IP Cores." IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems, Belgrade, Serbia, 2015. pp 1-4.

The author initiated the discussion and transformations that led to the novel concept of SCLM. Various RTL IP cores were transformed from FSMD to SCLM by the author, preserving the functionality even at a higher level of abstraction. Author re-run the RTL testbench on the SCLM models to verify the functional accuracy. Further experiments were conducted by the author to measure the improvement in simulation speed of RTL to SCLM models. The author wrote the paper describing the SCLM concept and the results comparing the simulation speed of the FSMD and SCLM models.

[C] Syed, Saif Abrar; Jenihhin, Maksim; Raik, Jaan. "Abstraction of clock interface for conversion of RTL VHDL to SystemC." IEEE International Advance Computing Conference, Gurgaon, India, 2014. pp 50-55.

The author came up with the idea of clock interface abstraction to improve the simulation speed. Then the author formalized the mechanisms for the

minimization of delta-cycles and events in clock-based SystemC models. Based on these concepts, the author abstracted the clock interface of the existing SystemC models. Then the author conducted various experiments to measure the effectiveness of the proposed methodology, and compared the simulation speeds of original and clock-abstracted SystemC models. The author wrote the paper describing the simulation speed-up obtained by the clock interface abstraction methodology and presented it in the IEEE International Advance Computing Conference (2014).

[D] Syed, Saif Abrar; Shyam, K.; Jenihhin, Maksim; Raik, Jaan; Babu, C.. "Performance Analysis of Cosimulating Processor Core in VHDL and SystemC." IEEE International Conference on Advances in Computing, Communications & Informatics, Mysore, India, 2013. pp 1-6.

The author analyzed the usage of the ESL models by the product-architects while transitioning from complete VHDL-based environment towards SystemC. As a result, the author made a setup for the co-simulation of Plasma core using VHDL and SystemC models, within ModelSim simulator. The author profiled the loading of the simulator by the VHDL and SystemC models within this setup. Then the author conducted various experiments by alternating the VHDL and SystemC implementation of the components within Plasmsa-core. Based on these experiments, the author proposed the best practices to replace a RTL VHDL by its SystemC model. The author wrote the paper describing the performance analysis of VHDL-SystemC co-simulation of a processor core.

[E] Syed, Saif Abrar; Jenihhin, Maksim; Raik, Jaan. "Optimization Methodologies for Cycle-Accurate SystemC Models Converted from RTL VHDL." IP-SoC 2013 (IP Based Electronic System Conference), Grenoble, France, 2013. pp 1-6.

The author identified the scope for the optimization of SystemC models translated from RTL VHDL in order to improve the simulation speed. Then the author analyzed the characteristics of the cycle-accurate SystemC models and developed the optimization methodology. Various experiments were conducted by the author to measure the effect of the optimization methodology. The author wrote the paper describing these optimization methodologies and presented it in IP-SoC (2013).

[F] Syed, Saif Abrar; Jenihhin, Maksim; Raik, Jaan. "Extensible Open-Source Framework for Translating RTL VHDL IP Cores to SystemC." IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems, Karlovy Vary, Czech Republic, 2013. pp 1-4.

The author formulated the rules for the translation of RTL VHDL to SystemC. These rules were then implemented in the open-source

framework zamiaCAD by the author so that any VHDL design parsed by zamiaCAD is automatically translated to SystemC. Then the author performed the experiments on various benchmarks for comparison with other translation tools. The author wrote the paper about these VHDL to SystemC translation rules.

The author has been pro-active in publishing the results of this research and is the first author of all the above mentioned papers.

OTHER RELATED PUBLICATIONS

[1] Syed, Saif Abrar; Jenihhin, Maksim; Raik, Jaan. "SystemC-Based Loose Models: RTL Abstraction for Design Understanding." DUHDE – 2nd Workshop on Design Automation for Understanding Hardware Designs – Friday Workshop at DATE, Grenoble, France, 2015. pp 1-1.

[2] Syed, Saif Abrar, Valentin Tihhomirov, Maksim Jenihhin and Jaan Raik. "VHDL To SystemC Translation And Abstraction: SystemC Manipulation Framework: From RTL VHDL To Optimized TLM SystemC." University Booth at DATE, Grenoble, France, 2015. pp 1-1.

[3] Jenihhin, Maksim; Tihhomirov, Valentin; Syed, Saif Abrar; Raik, Jaan; Bartsch, Günter. "zamiaCAD: Understand, Develop and Debug Hardware Designs." 1st Workshop on Design Automation for Understanding Hardware Designs (DUHDE), Friday Workshop at DATE, Dresden, Germany, 2014. pp 1-6.

[4] Syed, Saif Abrar; Jenihhin, M.; Raik, Jaan. "Open-source Framework and Practical Considerations for Translating RTL VHDL to SystemC." IP-SoC 2012 (IP Based Electronic System Conference), Grenoble, France, 2012. pp 1-6.

[5] Tihhomirov, V.; Tšepurov, A.; Syed, Saif Abrar; Jenihhin, Maksim; Raik, Jaan. "VHDL Design Debug Framework Based on zamiaCAD." University Booth at DATE, Grenoble, France, 2013. pp 1-1.

[6] Tšepurov, Anton; Tihhomirov, Valentin; Syed, Saif Abrar; Jenihhin, Maksim; Raik, Jaan. "Applications of the Open Source HW Design Framework zamiaCAD." University Booth at DATE, Dresden, Germany, 2012. pp 1-1.

ABBREVIATIONS

ASM Algorithmic State Machine
BCA Bus Cycle Accurate
BS-ESL Bluespec ESL
CA Cycle Accurate
CP Communicating Process
DES Discrete Event Simulation
EDA Electronic Design Automation
ESL Electronic System Level
FIFO First In First Out
FPGA Field-Programmable Gate Array
FSM Finite State Machine
FSMD Finite State Machine with Datapath
GCD Greatest Common Divisor
GUI Graphical User Interface
HDL Hardware Description Language
HLS High Level Synthesis
HW Hardware
IP Intellectual Property
ISA Instruction Set Architecture
ITRS International Technology Roadmap for Semiconductors
LE Load Elements
LRM Language Reference Manual
LT Loosely Timed
OSCI Open SystemC Initiative
PCA Pin Cycle Accurate
PV Programmers View
PVT Programmer's View with Timing
RISC Reduced Instruction Set Computing

RTL	Register Transfer Level
SA	Static Analysis
SCLM	SystemC-based Loose Model
SCR	Semantic Consistency Rules
SiP	System in Package
SLD	System Level Design
SoC	System-on-Chip
SW	Software
TB	Testbench
TF	Timed Functional
TLM	Transaction Level Modeling
UT	Un-Timed
UTF	Un-Timed Functional
VCD	Value Change Dump
VHDL	Very high speed IC Hardware Description Language
W3C	World Wide Web Consortium
WIP	Work In Progress
XML	eXtensible Markup Language

1 INTRODUCTION

This thesis addresses several approaches for the abstraction of Register Transfer Level (RTL) IPs. The ultimate goal is *comprehensive* abstraction of SystemC (SystemC, 2011) models, such that various interfaces as well as functionality of a design are abstracted.

This introductory chapter first presents the motivation leading to this research, followed by a more detailed problem formulation. This is followed by a summary of the main contributions and an overview of the thesis structure.

1.1 Motivation

The International Technology Roadmap for Semiconductors (ITRS, Online) has identified a dual trend in the consumer electronic products: miniaturization of the digital functions ("More Moore") and functional diversification ("More-than-Moore"). This is shown in the Figure 1.1 published by ITRS (Arden, 2011), where the terms are defined as follows:

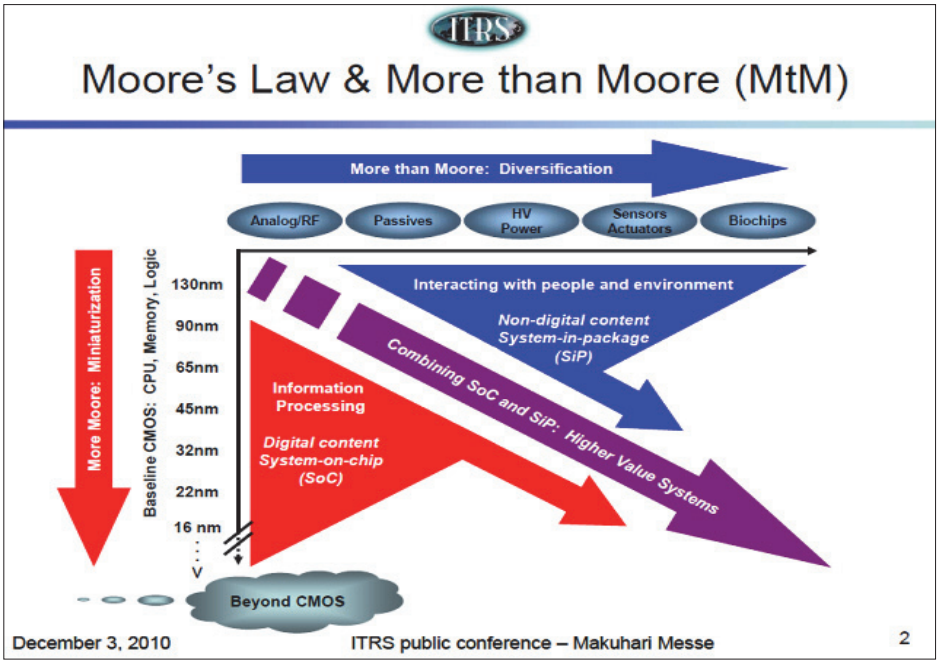


Figure 1.1 Miniaturization and diversification in SoC's, ITRS (Arden, 2011)

- "More Moore" (Scaling): Continued shrinking of physical feature sizes of the digital functionalities (logic and memory storage) in order to improve density (cost per function) and performance (speed, power).
- "More-than Moore" (Functional diversification): Incorporation into devices of functionalities that do not necessarily scale according to "Moore's Law", but provide additional value in different ways. The "More-than Moore" approach allows for the non-digital functionalities to migrate from the system board-level into the package (SiP) or onto the chip (SoC).

In order to enable these new trends, traditional design approaches at RTL are no longer suitable. The rapid increase of System-on-Chip (SoC) design complexity has emphasized the importance of high-performance simulation models and solutions for design realization. This has resulted in emergence of design methodologies at higher level of abstraction that enable unified development of the SoC designs. This level has been called system level, behavioral level, C-level, algorithmic level, Electronic System Level (ESL) (Bailey et al., 2012), or just a higher level (i.e. higher than RTL).

Realization of the ESL design methodology has been made practical with the introduction of TLM (Frank, 2006), that has gained wide acceptance within industry and academia for modeling the modern complex SoCs. TLM abstracts the pin-level details and cycle-level communication-protocol for modeling the memory-mapped interfaces of the IPs. Communication between simulation models is achieved via pre-defined function calls, resulting in easy modeling of interfaces, high simulation speeds and early availability of simulation platform. This enables the usage of TLM models for a range of design tasks, e.g. early software development, performance analysis, architecture exploration, hardware/software partitioning, etc. TLM enables the system level performance evaluation to select proper architecture, as well as software development and debug much before the availability of the physical hardware.

In order to enable the ESL and TLM methodologies, various languages and tools have been proposed and tried out to focus on different aspects of the system-level design. Few of these languages are behavioral Verilog (Verilog, 2005) or VHDL (VHDL, 2008), C (C11, 2011), Java, C++ class libraries (C++, 2014), C derivatives like SpecC and HandelC, SystemVerilog and special-purpose languages like Bluespec and Esterel. As of 2009, the SystemVerilog and Verilog language standards were merged into SystemVerilog 2009 (SV, 2009). Among these, SystemC has emerged as the dominant language that has been standardized as IEEE Standard 1666-2011 (SystemC, 2011). This standard also defines the TLM-2.0 interfaces for developing compliant models and tools.

SystemC has gained wide acceptance in the design of new digital IPs. However, there are numerous IPs already designed in VHDL or Verilog. With the

advances in SystemC ecosystem, like IEEE standardization, TLM-2.0 standard, SystemC high-level synthesis, many IP design houses are interested in developing SystemC models of their portfolio RTL IPs.

As a result of the above discussed advances in the SoC design practices, product-development companies are focusing their methodologies around ESL, TLM-2.0 and SystemC. SoC architects, hardware designers, software developers, verification teams, etc. have started using ESL for their project tasks. Either complete or partial TLM-based systems are now being modeled in SystemC, proliferating its usage and acceptance. In practice, system architects and system integrators often have access to a library of legacy RTL IP cores or obtain new ones from IP design houses. To address architectural exploration, early prototyping and simulation performance, such RTL IP cores are manually recreated at more abstract levels, which implies significant effort and is an inherently error-prone process.

Nowadays, product-development companies are increasingly demanding ESL-models in SystemC, along with VHDL/Verilog RTL IPs. With the acceptance of TLM methodology for system level design, IP-customers need ever increasing support for the TLM-2.0 models. These factors have put tremendous pressure on IP design houses, in terms of cost and effort, to provide high-level models. IP providers are deeply pressed to make available the SystemC models of their legacy VHDL or Verilog IPs to remain competitive in today's market.

1.2 Problem Formulation

As discussed in the previous section, IP design houses are facing a huge challenge of providing the abstract SystemC models of their RTL IPs. Manual abstraction of RTL has a potential of mismatch between different abstraction levels and is impractical in terms of time and effort. A preferred way is to automate the VHDL/Verilog translation and abstraction to SystemC, as this approach is faster and maintains consistency between source and SystemC implementations.

Abstraction of RTL IP that is acceptable and beneficial to the SoC industry must focus on the following salient features:

- SystemC is the target language of the high-level model.
- Generated code is readable and comprehensible for easy maintenance.
- Code-level optimizations are applied to improve the simulation speed.
- Memory-mapped inputs (e.g. address, data) are abstracted to TLM-2.0.
- Rest of the input signals (e.g. interrupts, reset, etc.) remain functional.
- Dependence on clock signal is minimized for higher simulation speed.

- Functional behavior is appropriately abstracted from RTL coding style.
- Complete methodology is available via an accessible EDA framework.

Purpose of this research is to develop a methodology to automatically abstract RTL IPs into higher-level models, with zero or minimal manual-interaction. Methodology for translation of VHDL RTL to abstract SystemC, adhering to the above desired features, is the result of this research. The described methodology has been implemented in zamiaCAD (zamiacad, Online), an open-source RTL entry and analysis framework. The target SystemC model can be generated for both cycle-accurate as well as untimed Programmer's View (PV) abstractions.

Results of this research are extremely relevant for the SystemC model providers, ESL/EDA tool-vendors and to the wider SystemC user community.

1.3 Main Contributions

This thesis explores various methodologies that can be used to abstract RTL IPs. Main contributions of this thesis are:

1. **A methodology for translation of VHDL RTL to SystemC:** Various rules are proposed to automatically generate SystemC from input VHDL RTL. These rules are generic in nature and can be applied to any VHDL RTL design.
2. **An optimization approach for simulation speed-up of translated SystemC:** Various optimizations techniques are introduced for the VHDL to SystemC translated code. These optimizations allow higher simulation speeds as compared to the directly translated SystemC-models.
3. **Optimization techniques for co-simulation:** While moving from complete system simulation in RTL towards SystemC implementation, co-simulation of VHDL-SystemC is often employed. Different strategies are discussed for selecting the modules to replace from VHDL to SystemC so that faster co-simulation speed is realized.
4. **An approach for abstraction of I/O interface:** Interface abstraction deals with the conversion of signal-level and cycle-accurate protocol to the standard TLM-2.0 protocol in SystemC. The introduced methodology can be used to abstract any arbitrary signal-level protocol to TLM-2.0. Only side-band signals like clock and interrupt are not converted to TLM-2.0 protocol.
5. **An approach for clock signal abstraction:** Clock abstraction aims at removing the clock signal, either partially or completely, preserving the behavior of the system design. Minimization of delta-cycles and events in a system level model is the core idea behind the clock signal

abstraction. Another proposal is to transform VHDL Finite State Machine with Datapath (FSMD) design to an equivalent Algorithmic State Machine (ASM) representation in SystemC that enables event-based triggering of ASM states.

6. **A methodology for functional abstraction:** Its focus is on abstracting the behavioral implementation of a module. Notion of SystemC-based Loose Modeling (SCLM) is introduced to functionally abstract a design implemented as VHDL FSMD. SCLM provides for an instrument to neglect design model parts irrelevant for particular manipulation step of the abstraction process, thus simplifying the abstraction flow.

Abstraction of memory-mapped signal-level interface to TLM-2.0 standard is accomplished by a novel and highly innovative work as a part of this research. It is decided to explore filing a *patent* for this work and hence has been omitted in this thesis.

1.4 Thesis Organization

This thesis consists of 6 main chapters. The rest of it is organized as follows.

Chapter 2 provides background information required for discussion of the various methodologies proposed in this thesis. First, new design methodology termed as Electronic System Level (ESL) design is discussed. It is followed by an introduction of abstraction levels, that form the basis of ESL. Then the SystemC language is introduced that is a set of C++ classes and macros approved by the IEEE Standards Association as IEEE Standard 1666-2011 in the SystemC Language Reference Manual (LRM). This is followed by a discussion of Transaction Level Modeling in which a transaction between two IP models is via a function call as opposed to multiple signal transitions that occur in RTL for the same transaction. Finally, zamiaCAD is introduced that is a modular and extensible open source framework supporting multiple use-cases, like advance hardware design and debug, analysis and research, along with translation and abstraction of VHDL to SystemC.

Chapter 3 focusses on a methodology to translate RTL VHDL IP cores to cycle-accurate SystemC designs. The SystemC output is emphasized to be human-readable and providing for clear correspondence to the source VHDL code, thus allowing further manual code changes and debug. The described methodology has been implemented in zamiaCAD and has been successfully applied to translate various VHDL benchmark designs. Availability of cycle-accurate SystemC model, instead of VHDL, has advantages across the product development, like (1) Architects integrate and update their C/C++ algorithms directly in the hardware model (2) Hardware designers, transitioning from VHDL to SystemC, learn relationship between VHDL and SystemC (3) Verification engineers directly use SystemC testbenches and can avoid

expensive co-simulation tools (4) Firmware developers working with assembly/C/C++ find familiar development environment for their tasks.

Chapter 4 explores various optimizations methodologies for the translated SystemC models as well as for models used in co-simulation. In addition to the plain VHDL-to-SystemC translation, there are possibilities of alternate implementations for a SystemC model. This chapter explores these alternate scenarios to get better simulating SystemC models. Another discussion in this chapter is about optimization scenarios that affect the co-simulation performance, as the VHDL and SystemC models are frequently co-simulated by architects as well as verification teams. The optimization methodologies discussed in this chapter are relevant to architects, designers, verification teams, and IP design houses that need to provide high-speed simulation models, and can be used for optimizing co-simulation tools as well system level models.

Chapter 5 discusses about the abstraction of clock interface that is an integral part of synchronous designs. The chapter begins with an introduction of various modes of simulation as well the SystemC simulation mechanism. It is important to understand these concepts in order to abstract the clock interface in the SystemC models. The chapter proposes a manipulation approach that transforms an FSMD RTL design in VHDL to an equivalent ASM representation in SystemC with explicit separation of design functionality by states. Finally, the clock interface is abstracted up to optimize the simulation performance.

Chapter 6 proposes an approach for automated abstraction of the computational part of cycle-accurate RTL IP cores to untimed TLM using a novel concept of SystemC-based Loose Models (SCLM). SCLMs provide for an instrument to neglect design model parts irrelevant for particular manipulation step of the abstraction process, thus simplifying the abstraction flow. As a result, the computational complexity of the abstraction process is reduced, thus increasing the overall scalability.

Chapter 7 finally draws the conclusions for this thesis and discusses possible directions for future work.

2 BACKGROUND

The purpose of this chapter is to introduce various concepts that form the basis of the current research documented in this thesis. Technology roadmaps predicts that the technology scaling for semiconductor chips continues at least until the year 2020 (ITRS, 2012). All major semiconductor roadmaps have emphasized the higher levels of abstraction in order to increase the design productivity. The most recent approach in this direction is the Electronic System Level (ESL) design methodology. ITRS lists the use of various abstraction levels for tasks ranging from system specification till implementation and verification. Various levels of abstractions and their use-cases are further discussed in this chapter that are conceptualized to handle various design tasks appropriately. Among the recently proposed high-level design languages, SystemC has emerged as an IEEE Standard 1666-2011 that is introduced in this chapter. Finally, the chapter discusses the zamiaCAD that is an open-source EDA tool for design entry, analysis, translation and simulation.

2.1 Electronic System Level Design

Modern SoCs are complex systems requiring advances in design methodologies as compared to the traditional approaches. Main challenges faced by SoC designers are:

a) increased functionality,

b) huge design space,

c) reduced time to market,

d) increased verification complexity,

e) specifications at higher levels,

f) highly complex software development,

g) enabling concurrent design of hardware/software

These challenges demand the entire SoC (hardware, software, interconnects, etc.) to be developed and verified as early and as parallel as possible.

Electronic System Level design is an electronic design methodology focusing on the higher abstraction levels. The term 'Electronic System Level' design was first coined by Gartner Dataquest. The book on ESL Design and Verification (Bailey et al., 2007) defines it as: “the utilization of appropriate abstractions in order to increase comprehension about a system, and to enhance the probability of a successful implementation of functionality in a cost-effective manner.”

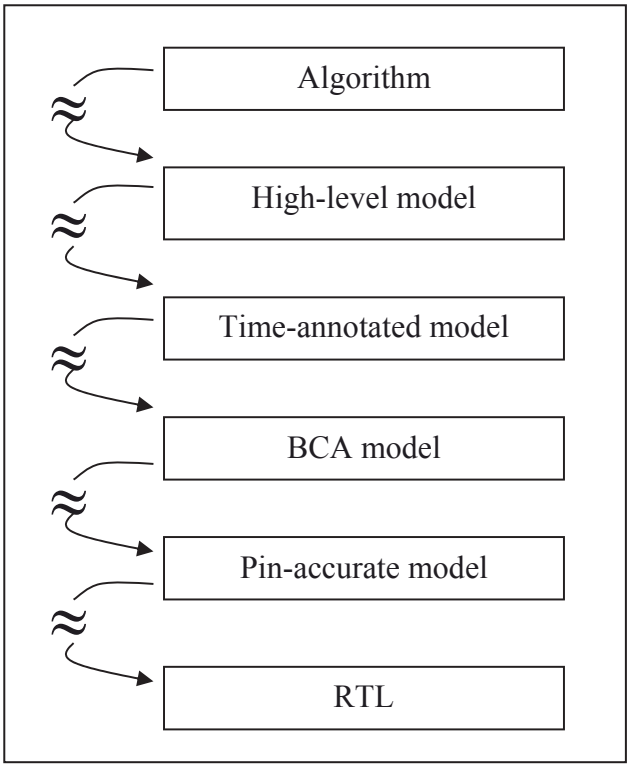


Figure 2.1 ESL Design Methodology

Most of the SoC design companies use the ESL methodology for system design, resulting in shorter development time with reduced effort and budget. ESL started as an algorithm modeling methodology, without any links to implementation. It has now evolved into a set of complementary methodologies that enable embedded system design, verification, and debugging. ESL has been successfully used for hardware and software implementation of custom SoC, system-on-FPGA (Field-Programmable Gate Array), system-on-board, and entire multi-board systems.

Figure 2.1 shows the common ESL design steps ranging from algorithmic level down to the RTL implementation. ESL methodology starts with an algorithmic representation of the desired functionality. This is accomplished using C, MATLAB, Excel, etc. to explore and finalize the algorithmic details. Once the algorithmic details have been finalized, the next level is to model the SoC at a higher level of abstraction using C, SystemC, etc. Aim of this level is to explore the system level details like memory-map, embedded application, functional validation, etc. This is commonly referred to as TLM and results in a golden prototype of the complete system. At the TLM-level, IPs are modeled at a functional level and the system bus is modeled independent of any particular

bus protocol. Once the high-level platform is validated for functionality, it is refined for architecture exploration by annotating timing information in the functional model. Till now, the external interfaces of the models are the transaction interfaces that are further refined for pin-accuracy as per the specifications of the IPs. This pin-accurate view of the system enables the verification of the communication protocol and interrupts. Final refinement is towards the RTL of the design that is then synthesized and fabricated.

As discussed earlier, ESL ranges the completed spectrum of hardware (HW) and software (SW) design cycles in an SoC ranging from requirements validation down till implementation. Table 2.1 depicts the suitability and usage of various languages used in the industry for the ESL tasks.

Behavior of the entire system is modeled in ESL using high-level languages like C, C++ or MATLAB or using graphical "model-based" design tools like VisualSim or Simulink. Newer languages are emerging that enable the creation of higher level models, like SysML, SMDL and SSDL, as well as system design automation products, like Incisive Verification Platform (Incisive, Online) and Cocentric System Studio (Synopsys, Online). EDA tools can be employed to automate the rapid and correct-by-construction implementation of the system.

Table 2.1 ESL tasks and suitable languages

	Verilog / VHDL	***Bluespec***	***System Verilog***	***e***	***SystemC***	***MATLAB***
Requirements						√
Architecture		√			√	√
HW / SW					√	√
Behavior	√	√	√		√	
Verification		√	√	√	√	
Testbench	√	√	√	√	√	
RTL	√				√	
Gates	√					

The focus of this thesis is on VHDL and SystemC design languages, so that the VHDL RTL IP is comprehensively abstracted to high-level SystemC model.

2.2 Abstraction Levels

Levels of abstraction are the basic premise in the ESL methodology. The most abstract level is defined by Communicating Processes (CP), as shown in Figure 2.2, that can be progressively refined to the Cycle Accurate (CA) level. Each level is suited for a particular design task and there is a conscious trade-off between accuracy and speed at each level. These abstraction levels can be described as follows:

1. **Communicating Processes (CP):** The CP-level is free of any architectural information. The entire system is described at the level of a set of processes communicating via (unbounded) FIFOs. The processes focus on the actual task (e.g. frame processing) without consideration for whether the task would be finally realized as a piece of silicon or as a software algorithm executing on a DSP.

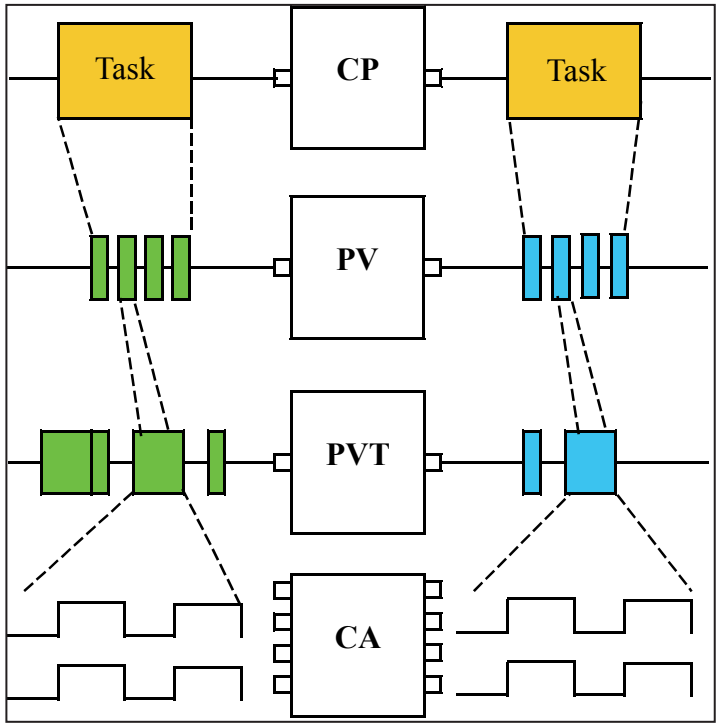


Figure 2.2 Abstraction levels in ESL

2. **Programmers View (PV):** The CP network is refined with architectural details to get the PV level. A PV-model has complete functionality of the hardware-IP but no sense of timing. This allows for an early distribution of compute tasks and a partial ordering of their

response, allowing application developers a software view of the system.

3. **Programmers View with Timing (PVT):** The PV-model is annotated with timing-details to arrive at the PVT-model. A PVT-model does not model all signal transitions. Instead a transaction interface will be used to complete an entire set of transitions. However, timing may be annotated into the model at known intervals in order to preserve and guarantee a certain ordering of events and to get an approximate view of cycle count for transactions.
4. **Cycle Accurate (CA):** The lowest level of abstraction is represented by a CA model that describes timing sequence as in actual hardware. Accuracy at this level is at its highest but speed at its lowest, as the model represents all the signals of the hardware IP.

These abstraction levels are used for different steps in the SoC design cycle, such as software development, architecture exploration, performance analysis and system verification. Usage of abstraction levels is as follows:

A) **Software Development:** Typically, software development takes place, in the embedded context, when the hardware is available either as an emulator or a chip. This places immense pressure on HW-SW integration and final system validation. It is often the case that the integration and validation do not go as anticipated and either hardware or software has to re-spin. This can be avoided by beginning software development early in the design lifecycle and by carrying out HW-SW pre-integration tests on the functional models of the processor(s), memories and other peripherals. Hence, the PV level of abstraction is best suited for this task. These PV models can provide the same tool suite interface as with hardware reference designs but with better visibility, control and flexibility, and execute in the range of millions of instructions per second. Note that it is possible to use the PVT or CA level also for software development but is not practical due to drastic reduction in simulation speed.

B) **Architecture Exploration, Performance Analysis:** Architecture exploration of an SoC requires the models to have complete functionality (as in PV) along with timing details. The timing information in a model can be dynamic/implicit or static/explicit. The dynamic/implicit timing depends on the actual events occurring in the system and is expressed as a function of system events. The static/implicit timing is embedded within the model itself and is made independent of the system. Timing details necessitate the use of PVT-level of abstraction for architecture exploration and performance

analysis. Note that it is possible to use the CA level also for these tasks but is not practical due to reduction in simulation speed.

C) **Verification:** Both hardware verification and HW/SW co-verification require a level of accuracy predictive of real chip timing behavior. Also, pin-accurate RTL testbenches/models may be used for verification tasks. Hence, CA level of abstraction is employed for verification purposes. The CA-models simulate 5-10 times faster than RTL models and hence can still execute the embedded software to detect design errors. This leads to a tremendous speedup of verification, one to two orders of magnitude larger than the Bus Functional Model and Instruction Set Simulator based verification approaches.

Table 2.2 summarizes the characteristics and uses of each abstraction level, as discussed earlier in this section.

Table 2.2 Characteristics and uses of abstraction levels

Level	***Characteristics***	***Use-cases***
CP, PV	Very high simulation speed, no timing information	Functional correctness, Software development
PVT	Moderate simulation speed, timing details	Architecture exploration, HW/SW partitioning
CA	High accuracy, low simulation speed	Co-verification, Device-driver

2.3 SystemC

As the ESL methodology was adopted by the SoC companies, there was a rise in development of proprietary solutions. Although this worked well for each individual SoC company, it created a barrier for exchange of high-level models across the semiconductor industry. This demanded some standard high-level language(s) to be defined and used across the industry. Among these high-level languages, SystemC, System-Verilog, and SpecC represent the initial efforts that were most widely used across the industry. With the passage of time, SystemC gained more popularity and acceptance in the industry due to its applicability to system level modeling, architectural exploration, software development, functional verification, and high-level synthesis.

SystemC is a set of C++ classes and macros that has been approved as IEEE 1666 Standard SystemC Language Reference Manual (LRM) (SystemC, 2011).

The LRM provides the definitive statement of the semantics of the SystemC. SystemC traces its origins to work on Scenic programming language (Liao et al., 1997). SystemC provides an event-driven simulation kernel in C++, together with signals, events, and synchronization primitives. These facilities enable a designer to simulate concurrent processes, each described using plain C++ syntax. SystemC processes can communicate in a simulated real-time environment, using signals of all the datatypes offered by C++, some additional ones offered by the SystemC library, as well as those defined by the users. SystemC models can be developed at various abstraction levels like Un-timed (UT), Loosely Timed (LT), Cycle Accurate (CA) etc. to suite various SoC requirements like software development, architectural exploration, synthesis, etc.

Features and advantages of SystemC over RTL languages are:

- Design of an SoC at a higher level of abstraction.
- Development of both hardware and software components.
- Focus on the functionality of the system, rather than its implementation.
- Effective evaluation of different architecture alternatives.
- Hardware and software partitioning of the system functionality.

Building-blocks of a SystemC model are:

- **Modules:** Modules are the basic building blocks of a SystemC design hierarchy. A SystemC model usually consists of several modules which communicate via ports. The modules can be thought of as a building block of SystemC.
- **Ports:** Ports allow communication from inside a module to the outside (usually to other modules) via channels.
- **Exports:** Exports incorporate channels and allow communication from inside a module to the outside (usually to other modules).
- **Processes:** Processes are the main computation elements. They are concurrent.
- **Channels:** Channels are the communication elements of SystemC. They can be either simple wires or complex communication mechanisms like FIFOs or bus channels. Elementary channels can be of type signal (wire), buffer, fifo, mutex, semaphore.
- **Interfaces:** Ports use interfaces to communicate with channels.
- **Events:** Events allow synchronization between processes and must be defined during initialization.

- **Data types:** SystemC introduces several data types which support the modeling of hardware.
  - Extended standard types:
    - sc_int<n> n-bit signed integer
    - sc_uint<n> n-bit unsigned integer
    - sc_bigint<n> n-bit signed integer for n > 64
    - sc_biguint<n> n-bit unsigned integer for n > 64
  - Logic types:
    - sc_bit 2-valued single bit
    - sc_logic 4-valued single bit
    - sc_bv<n> vector of length n of sc_bit
    - sc_lv<n> vector of length n of sc_logic
  - Fixed point types:
    - sc_fixed<> templated signed fixed point
    - sc_ufixed<> templated unsigned fixed point
    - sc_fix untemplated signed fixed point
    - sc_ufix untemplated unsigned fixed point

Ecosystem of SystemC consists of varying offerings and options, like:

- Proof-of-concept simulator, freely available from http://accellera.org/
- Tool providers, e.g. Cadence
- Model providers, e.g. Synopsis
- Service providers, e.g. Circuitsutra
- Academic institutes, e.g. OFFIS (Germany)
- Professional consulting, e.g. Gary Smith EDA
- Papers and journals, e.g. IEEE
- Users Group meetings, like
  - ESCUG - European SystemC Users Group
  - NASCUG - North American SystemC User's Group
  - LASCUG - Latin American SystemC User's Group
  - ISCUG - Indian SystemC User's Group

The author is an active member of the ISCUG as well as manages the LinkedIn group related to Open SystemC Initiative (OSCI).

History of SystemC dates back to the formation and announcement of OSCI alliance in 1999, launching the proof-of-concept SystemC kernel in 2000. SystemC 2.1 LRM and TLM 1.0 standard was released in 2005, the same year IEEE approved the IEEE 1666–2005 standard. Analog extension to SystemC, AMS, was introduced in 2010 and the IEEE approved the IEEE 1666–2011 standard for SystemC in 2011.

2.4 zamiaCAD

Several research contributions presented in this thesis are implemented using the zamiaCAD's graphical user interface (GUI) for code entry and exploits the internal data-structures (abstract syntax tree and instantiation graph) for semantic analysis of VHDL for translation to SystemC.

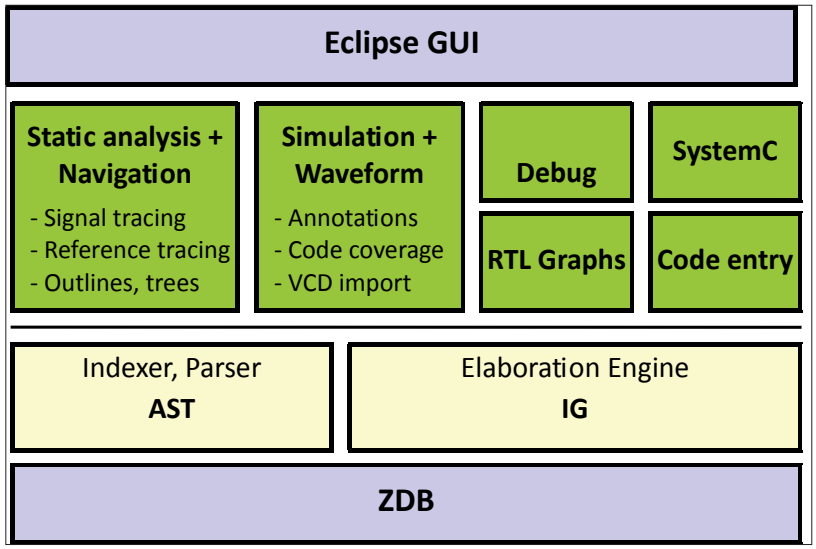


Figure 2.3 Structure of zamiaCAD (Jenihhin et al., 2014)

zamiaCAD (Jenihhin et al., 2014) is a modular and extensible open source framework supporting multiple use-cases, like advance hardware design and debug, analysis and research, along with translation of VHDL to SystemC. Currently, zamiaCAD front-end supports IEEE VHDL 2008 and the IEEE 1364-2005 Verilog support is a work in progress.

Figure 2.3 shows the structure and applications of zamiaCAD. It can handle large industrial designs, such as an SoC of 3500 Leon3MP-s (Leon3, Online), and provides all facilities through Eclipse-based (Eclipse, Online) GUI front-end. This framework is developed to provide the research community with a robust open-source platform equipped with powerful debugging functionality.

zamiaCAD consists of three basic building blocks:

1. Frontend for Hardware Description Language (HDL) parsing: full VHDL 2008 parser, syntax tree, elaboration (Verilog 2005 parser and syntax tree, no elaboration yet), persistent and scalable syntax tree storage.
2. Core for intermediate design representation and analysis: based on a powerful, persistent and scalable design database, fully elaborated design model, full source back-annotation, static analysis, interpreter for quick expression evaluation, built-in simulator for validation.

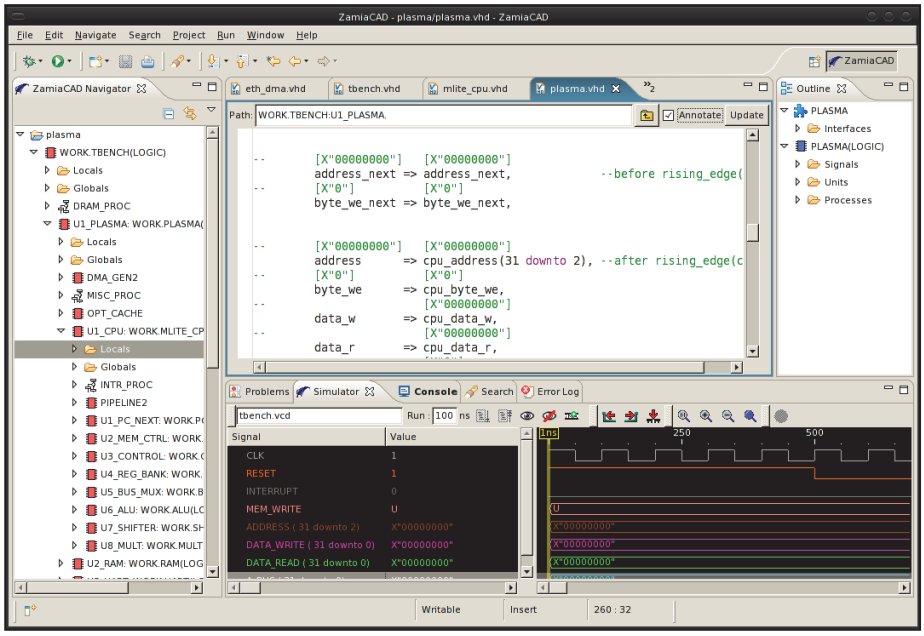


Figure 2.4 zamiaCAD GUI

3. GUI based on Eclipse IDE Plugin: graphical viewers and editors, automatic model builder, as shown in Figure 2.4.

The frontend consists of an elaboration engine and a parser. Currently, Verilog has only a parser, whereas VHDL has a complete frontend. zamiaCAD has a complete VHDL compiler with project build facilities. Applications like a simulator and an Eclipse based GUI are built on top of the IG and potentially language dependent structures like the abstract syntax tree (AST).

A simplified flow of the zamiaCAD framework is shown in Figure 2.5. An object database ZDB (zamiaCAD Data Base), which has been custom-designed and highly optimized for scalability and performance, is used for zamiaCAD applications. Full elaboration in zamiaCAD semantically resolves the Abstract Syntax Tree generated by the parser and results in a set of scalable Instantiation Graph data structures, stored in ZDB. IG allows clients to perform all kinds of

tasks. IG is a densely connected graph of semantically resolved objects representing elements of the hardware design. Design database is also automatically and efficiently persisted to disk to save time on later elaboration.

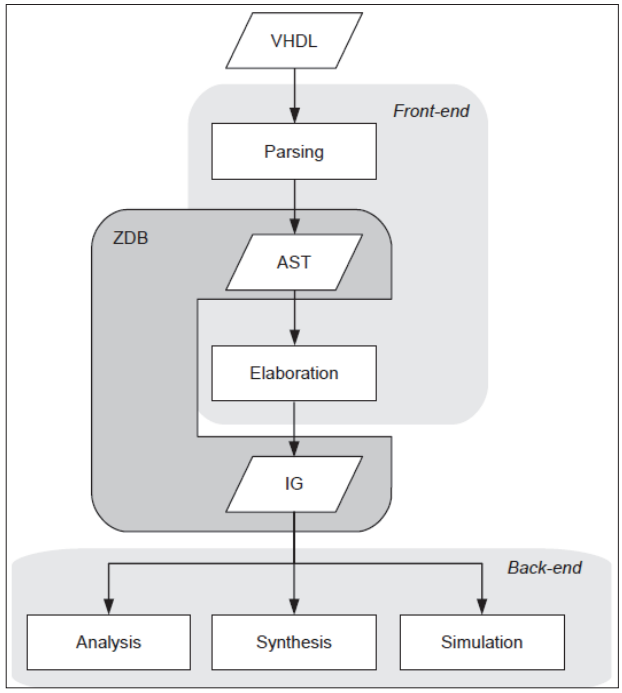


Figure 2.5 Flow of the zamiaCAD framework (Tšepurov, 2012)

Parsing all VHDL sources that are accessible through the directories configured in a project's setting can be very time consuming. Since a significant portion of the design units defined in those files might never actually get instantiated or used in the project (this is especially true for technology/simulation libraries), it is often not necessary. Therefore, zamiaCAD comes with a very high-speed VHDL indexer, which will extract only the information about which design units are declared in which files. The actual VHDL parser is then used in an on-demand manner only on local source files and files that define design units that are actually needed during elaboration.

The zamiaCAD framework addresses mainly advanced HW RTL design, verification and analysis, and offers the following functionality:

1. Code entry features comprise syntax highlight, code entry, content assist (identifier auto-completion), extensible set of HDL templates and an incremental IG model builder. zamiaCAD's HDL code editor is based on Eclipse's editor.

2. Static analysis (SA) tasks and navigation are feasible due to the fully elaborated IG design model, where all identifiers (including types) are resolved. SA tasks include tracing of parts of signals, precise global signal tracing, precise matching of overloaded subprograms, tracing through generate-statements, advanced signal value annotations (e.g. annotating only one bit of a vector), computing expressions on the fly, source-less and sink-less signal detection, Finite State Machine (FSM) recognition (WIP), code outline and code hierarchy view, declaration search.
3. VHDL simulator, implemented in accordance with the IEEE Standard VHDL Language Reference Manual. Simulator also provides code coverage measurements, value/timing source back-annotations and importing of waveform files in VCD format.
4. Debugging features include an experimental algorithm for automatic design error localization, which brings together SA and simulator to narrow down the search space where the design bugs are to be localized.
5. SystemC generation from VHDL. As part of this research, zamiaCAD is able to generate a cycle-accurate, pin-accurate SystemC model out of RTL VHDL, thus preserving the level of abstraction.

As a framework, zamiaCAD offers a scripting interface, implemented in JPython, for controlling external tools, such as ghdl, ModelSim etc. JPython scripts also make zamiaCAD itself easier to use from the command line.

2.5 Chapter Summary

This chapter has provided necessary information required for understanding the research documented in this thesis. The first part of this chapter discussed the need and practice of the ESL methodology. The second section introduced the abstraction levels that form the underlying concept in the ESL methodology. The third part discussed the SystemC modeling language that has emerged as an IEEE standard for implementing ESL and is the basis of the current research. Finally, zamiaCAD was introduced that is a modular and extensible open source framework supporting multiple use-cases, like advance hardware design and debug, analysis and research, along with translation of VHDL to SystemC and subsequent abstractions.

3 VHDL-TO-SYSTEMC TRANSLATION

SystemC has gained wide acceptance in the design of VLSI SoCs. At the same time there exists a large number of legacy IP cores described in VHDL whose reuse and integration into SystemC ecosystem is highly demanded. However, there is a lack of any standard approach in this regard.

This chapter describes a methodology to translate RTL VHDL IP cores to cycle-accurate SystemC designs. The described framework has been implemented in zamiaCAD platform and successfully applied to translate various VHDL benchmark designs.

The translated SystemC output is emphasized to have:

• *Human readability:* This is an important consideration about the usage of the generated SystemC code: whether the SystemC code is only to be fed to a compiler or is it going to be maintained by human developers. Many VHDL to SystemC translation-tools lack this feature and generate an obscure code which is not fit for human use. It goes a long way to decide the human relationship with the generated SystemC code.

• *Correspondence of the translated SystemC to VHDL:* If a team of designers needs to maintain both the VHDL and SystemC code-bases, then it is appropriate to have a consistent view between the two. The translation mechanism must decide about this early on and take care of. Usage of the same module names, variables, constructs, etc. must be adhered too, unless SystemC does not support a feature inherently.

The chapter is divided into following sections: Section 3.1 presents the related work in this domain. Methodology for VHDL-to-SystemC translation is discussed in Section 3.2. Results of applying the translation to various benchmarks are presented in Section 3.3 and finally Section 3.4 summarizes the chapter with pros and cons of the presented methodology and discussion on future scope of work.

3.1 Related Work

Code translation tools speed up the model development time as well as guaranty consistency between original and translated models. Both top-down approaches and VHDL IP-reuse approaches are used for deriving SystemC models. High-level synthesis tools, like (Cynthesizer, Online), (Catapult, Online) and (C-to-Si, Online) are an example of top-down approach. Bottom-up approaches, like (Bombieri et al., 2010) and (SAVANT, Online) are examples of IP reuse. However, readability and efficiency of the translated code are two major problems of the current code translators.

Among the commercial solutions there is Carbon Model Studio (Carbon, Online) that allows creating configurable SystemC models from RTL VHDL or Verilog descriptions. It is targeted mainly at simulation speedup and does not intend to create human-readable output. HIFSuite (HIFSuite, Online), (Bombieri et al., 2010) is a design and verification framework addressing manipulation and integration of heterogeneous design parts. Similarly, it allows dumping out RTL VHDL descriptions into SystemC. Before the dump out the design models can be manipulated on the internal HIF model. The output result also does not consider human-readability and correspondence to the source VHDL. The approach supports equivalence checking to prove the correctness of the result. However, the major advantage of this tool is ability to raise the level of abstraction of the design from RTL to TLM. Only a demo version of the tool was available for the experiments.

Another option is to use non-commercial and free solutions to generate SystemC from VDHL. VHDLParser by University of Tuebingen (Tuebingen, Online) and VH2SC by HT-Lab (VH2SC, Online). Both approaches consider mapping of the source VHDL to a limited set of SystemC constructs. These tools have demonstrated significant limitations, do not guarantee equivalence and they are not maintained anymore. VHDLParser dates to 2001 and addresses SystemC 1.0. Closed sources of the tools do not allow engineers to extend them to their needs.

The most relevant approach is published by OFFIS in (Görgen et al., 2012). It assumes creation of readable SystemC representations from VHDL that are targeted to be wrapped and simulated in the Simulink environment. The approach is claimed to support industrial designs, however only an illustrative example details are available (VHDL2SC, 2012). This practical work also does not provide for equivalence checking mechanisms or results. There are known approaches for creating SystemC models from Verilog (Verilator, Online) and tools targeting creation of other C++ subsets like FreeHDL (FreeHDL, Online) and VHDLC (Ostatic, Online).

Different from the existing works, this chapter addresses an open-source extensible framework for fully automated generation of standard SystemC descriptions from legacy RTL VHDL IP cores targeting their reuse in industrial SystemC environments. The framework assumes future extension of the current implementation to support automated RTL to TLM abstraction.

3.2 Translation Methodology

The translation methodology is highlighted by the rules described in this section. The proposed methodology has been implemented in zamiaCAD, using the abstract syntax tree (AST) generated by the platform's front-end while assuming usage of the semantically resolved scalable instantiation graph (IG) data structures (Tšepurov, 2012) for complex design constructs.

3.2.1 SystemC module constructor on-the-fly

SystemC module needs a constructor, unlike a VHDL model. Some information, like sensitivity-list, is part of the SystemC constructor that is only available in VHDL process declaration, as shown in Table 3.1. Hence, SystemC constructor can only be written once complete VHDL code has been parsed. This demands that the information for the SystemC constructor be gathered while parsing the VHDL module, accumulated on-the-fly and finally written to the SystemC file.

Table 3.1 SystemC module constructor on-the-fly

VHDL Syntax	SystemC Syntax
P1 : process(A,B) begin ... end process P1;	class module : public sc module {... void P1(void); …}; module::module() {... SC METHOD(P1); sensitive << A << B; ...}

Table 3.2 SystemC model naming

VHDL Syntax	SystemC Syntax
entity E is . . . end E; architecture behav of E is begin . . . end behav;	class E behav : public sc module { . . . };
architecture RTL of E is begin . . . end RTL;	class E RTL : public sc module { . . . };

3.2.2 Multiple architecture definitions

VHDL enables definition of multiple architectures for a single entity. However, SystemC is limited in this regards that there is only a single SystemC class available for each representation. A practical approach in this regard is to derive the name of each SystemC module from a combination of the VHDL entity and architecture names, e.g. by concatenating VHDL entity and architecture names. Table 3.2 shows this approach.

3.2.3 Using constructors rather than SC_CTOR

VHDL allows model parameterization using generics. In order to achieve the same in SystemC, it is recommended to use the class constructors instead of SC_CTOR. This enables the VHDL generics to be used as constructor parameters, as shown in Table 3.3.

Table 3.3 SystemC module constructor instead of SC_CTOR

VHDL Syntax	SystemC Syntax
`entity E is` `  generic (G: integer := 10);` `  . . .` `end E;`	`class E_behav : public sc_module` `{ . . .` `  E behav(sc module name n, int G=10)` `  : sc module(n), my G(G)` `  . . .` `};`

3.2.4 Virtual destructor

As it is also discussed in the guideline 7 in Effective C++ (Meyers, 2005), if the SystemC model has any virtual function then it should have a virtual destructor. Additionally, the classes not designed to be base classes or not designed to be used polymorphically should not declare virtual destructors.

3.2.5 Native C++ data-types for faster simulation

Using lesser state types might suffice instead of higher state types. Take an example of VHDL std_logic type. If only values '0'/'1' are relevant, and not the 'X'/'Z' states, then C++ bool type is recommended instead of exact replacement by SystemC sc_logic type. However, this approach must be used with care. The entire code needs to be analyzed to make sure that the left-out values (e.g. 'X'/'Z' in this case) are not used anywhere in the code. Also, this restricts the SystemC model for later design updates when the dropped values might be used.

3.2.6 Operator precedence differences between VHDL and SystemC

This consideration is extremely important to keep up with. VHDL and SystemC have different precedence for certain operators. As is common, use parenthesis for clarity and overwriting precedence.

3.2.7 Using port-methods for clarity

SystemC uses the same operator '=' for reading the variables as well as the ports. To remove ambiguity, use methods for read/write on ports and to reserve '=' for variable assignments, as shown in Table 3.4.

Table 3.4 SystemC port method

VHDL Syntax	SystemC Syntax
`entity E is` `  port (A: in std logic);` `end E;` `architecture behav of E is` `  signal X, Y: std_logic;` `begin` `  X <= A;` `  Y <= X;` `end behav;`	`class E behav : public sc module` `{` `  sc in<sc logic> A;` `  sc logic X, Y;` `  . . .` `  X = A.read();` `  Y = X;` `  . . .` `};`

3.2.8 VHDL and SystemC port types

Both VHDL and SystemC have similar types of ports and can be mapped uniquely from VHDL to SystemC. Table 3.5 shows the similar port mapping between VHDL and SystemC. Data-type of the VHDL ports can be mapped to similar port data-types in SystemC, e.g. `std_logic` port in VHDL is mapped to `sc_logic` port in SystemC. VHDL has a unique 'buffer' port type, that indicates that the port is out type, but its value can be read inside the entity. For all practical purposes this can be mapped to SystemC 'sc_inout' port.

Table 3.5 VHDL and SystemC port types

VHDL Syntax	SystemC Syntax
port (A: in std_logic);	sc_in<sc_logic> A;
port (B: out std_logic);	sc_out<sc_logic> B;
port (C: inout std_logic);	sc_inout<sc_logic> C;
port (D: buffer std logic);	sc inout<sc logic> D;

3.2.9 Translating VHDL process

VHDL process is used to implement sequential behavior. SystemC allows either SC_METHOD or SC_THREAD for such purpose. Sensitivity of the VHDL-process becomes the sensitivity of SystemC translation. Whether to choose SC_METHOD or SC_THREAD depends on the usage of wait within VHDL process. Since an SC_METHOD cannot have wait statement, any VHDL process with a wait statement should be implemented as SC_THREAD, otherwise SC_METHOD might be preferred. Table 3.6 shows this approach.

Table 3.6 SystemC process declarations

VHDL Syntax	SystemC Syntax
P1: process (A,B) begin . . . end P1; P2: process begin . . . wait . . .; . . . end P2;	SC METHOD(P1); sensitive << A << B; void P1(void){. . .} SC THREAD(P2); void P2(void) { . . . wait(. . .); . . . }

3.2.10 Concurrent statements in VHDL

Sequential statements within a VHDL process goes inside a similar SC_METHOD or SC_THREAD in SystemC. But what about the concurrent statements outside of any VHDL process? There are 2 ways to handle such statements in SystemC.

a) Use single SC_METHOD for all concurrent statements, and sensitive to all the source (right-hand-side) variables in these statements, as shown in Table

3.7. This approach is easy to implement but has a drawback that the SC_METHOD executes whenever any variable changes. This affects the performance.

b) Separate SC_METHOD for each statement, being sensitive to only this statement's source variable, as shown in Table 3.8. The approach adds to the translation effort as well as increases the code-size but only single statement is executed each time, reducing the simulation overhead.

3.2.11 Chain of SystemC library calls

Sometimes it is necessary in practice to employ a chain of SystemC library calls to achieve the desired behavior, e.g. comparing a single-bit value from a port.

Table 3.7 Single SC_METHOD

VHDL Syntax	SystemC Syntax
architecture behav of E is begin X <= A and B; Y <= not A; Z <= X or Y; end behav;	SC METHOD(P1); sensitive << A << B << X << Y; void P1(void) { X = A && B; Y = !A; Z = X || Y; }

Table 3.8 Separate SC_METHODs

VHDL Syntax	SystemC Syntax
architecture behav of E is begin X <= A and B; Y <= not A; Z <= X or Y; end behav;	SC_METHOD(P_X); sensitive << A << B; void P X(void) { X = A && B; } SC METHOD(P Y); sensitive << A; void P_Y(void) { Y = !A; } SC METHOD(P Z); sensitive << X << Y; void P Z(void) { Z = X || Y; }

Table 3.9 SystemC switch-case

VHDL Syntax	SystemC Syntax
entity E is port(SEL: in std_logic_vector(2 downto 1)); end E; architecture behav of E is begin case SEL is when "00" => . . . ; when "01" => . . . ; . . . end case; end behav;	class E behav : public sc module { sc_in<sc_lv<2> > SEL; }; void p(void) { int p SEL = SEL.read().to uint(); switch(p_SEL) { case 0:. . . ; break; case 1:. . . ; break; . . . } }

Table 3.10 SystemC if-then-else

VHDL Syntax	SystemC Syntax
entity E is port(VAL: in integer range 0 to 100); end E; architecture behav of E is begin case VAL is when 0 to 10 => . . . ; when 11 to 25 => . . . ; . . . end case; end behav;	class E_behav : public sc_module { sc in<int> VAL; }; void p(void) { int p VAL = VAL.read(); if ((0 <= p VAL) && (p VAL <= 10)) {. . .} else if((11 <= p VAL) && (p VAL <= 25)) {. . .} . . . }

3.2.12 Translating switch-cases

VHDL allows variables, logic-types as well as ports in the switch-case construct, whereas SystemC allows only integers. Hence, VHDL switch literal must be converted to an integer, as shown in Table 3.9. Another approach is to use if-then-else constructs, as in Table 3.10, where the VHDL switch literal

a) uses a range of values

b) cannot be converted to an integer, e.g. string types.

3.2.13 Handling clock-edge sensitivity

VHDL and SystemC uses varying notations to describe clock-edge sensitivity. Table 3.11 shows the translation for positive-edge clock sensitivity. But the process P1 is invoked on both the edges of clock. Efficient and recommended, but elaborate, approach is to analyze the VHDL implementation of the process P1 to determine the clock-sensitivity of interest, and then use it in SystemC, as shown in Table 3.12. Such scheme is better for an event-based simulator.

Table 3.11 Double clock-edge sensitivity

VHDL Syntax	SystemC Syntax
P1: process (CLOCK) begin . . . if(CLOCK'event and CLOCK = '1') then . . . end if; . . . end process;	SC METHOD(P1); sensitive << CLOCK; void P1(void) { . . . if (CLOCK.posedge()){ . . . } . . . }

Table 3.12 Single clock-edge sensitivity

VHDL Syntax	SystemC Syntax
P1: process (CLOCK) begin . . . if(CLOCK'event and CLOCK = '1') then . . . end if; . . . end process;	SC_METHOD(P1); sensitive << CLOCK.pos(); void P1(void) { . . . //no need of this: if (CLOCK.posedge()){ . . . } . . . }

3.2.14 SystemC process at start-up

SystemC scheduler has an interesting feature that each process (SC_METHOD or SC_THREAD) is always invoked once at the start-of-simulation. This happens even in absence of any event! Use dont_initialize() to stop this default invocation, as shown in Table 3.13.

Table 3.13 SystemC process at start-up

VHDL Syntax	SystemC Syntax
P1: process (CLOCK) begin . . . end process;	SC METHOD(P1); sensitive << CLOCK; dont_initialize();

3.2.15 Executing the SystemC model constructor

SystemC has a unique requirement to execute the constructor of each module instance. This is unlike VHDL as there is no constructor for a VHDL design. Table 3.14 shows the structure of a 4-bit adder from 4 instances of 1-bit adder.

3.3 Results

The above mentioned translation methodology natively parses the IG graph to generate a SystemC model with the following features:

- Cycle-accurate, pin-accurate, bit-accurate generated code
- Human-readable generated code
- SystemC module-name made from entity and architecture names
- Variable-names same as in VHDL
- Sensitivity-list same as in VHDL
- Process-names same as VHDL, else named as process_<line-number>
- Separate header and source-code files
- Proper indentation of the generated code
- Successful compilation with C++ compiler

The current implementation of the proposed framework based on zamiaCAD is able to successfully and automatically translate most of the exercised designs into a compilable SystemC output.

Table 3.14 Executing SystemC model constructor

VHDL Syntax	SystemC Syntax
entity Bit1 Adder is port(a,b,c_in: in bit; sum, c_out: out bit); end Bit1_Adder; entity Bit4_Adder is port(a,b: in std logic vector(3 downto 0); c in: in bit; sum: out std logic vector(3 downto 0); c_out: out bit); end Bit1_Adder; architecture struct 4 of Bit4 Adder is component Bit1 Adder . . . end component; begin Bit1 Adder A: Bit1 Adder port map . . . ; Bit1 Adder B: Bit1 Adder port map. . . ; Bit1_Adder_C: Bit1_Adder port map. . . ; Bit1 Adder D: Bit1 Adder port map. . . ; . . . end struct_4;	class Bit1 Adder : public sc module { sc_in<bool> a,b,c_in; sc out<bool> sum, c out; }; class Bit4 Adder : public sc module { sc in<sc uint<4> > a,b; sc_in<bool> c_in; sc_out<sc_uint<4> > sum; sc out<bool> c out; Bit1_Adder Bit1_Adder_A; **//instantiate** Bit1_Adder Bit1_Adder_B; **//instantiate** Bit1_Adder Bit1_Adder_C; **//instantiate** Bit1_Adder Bit1_Adder_D; **//instantiate** }; Bit4 Adder::Bit4 Adder(sc module name n): sc module(n) **, Bit1_Adder_A("Bit1_Adder_A") //CTOR** **, Bit1_Adder_B("Bit1_Adder_B") //CTOR** **, Bit1_Adder_C("Bit1_Adder_C") //CTOR** **, Bit1_Adder_D("Bit1_Adder_D") //CTOR** { . . . }

As a separate case-study, RTL VHDL *ram* core of the OpenCores.org *plasma* processor design (Plasma, Online) was also translated successfully. The resulted cycle-accurate SystemC module was successfully co-simulated in the Mentor Graphics ModelSim environment with the original testbench producing an equivalent system behavior.

The discussed methodology is applied to a set of benchmarks and compared against the available tools described in Section 3.1. Table 3.15 demonstrates the results of translating a set of ITC99 benchmarks (ITC99, 1999) and a greatest common divisor implementation *gcd*.

Table 3.15 Comparison of translation tools on various benchmarks

Design	*gcd*	*b01*	*b02*	*b03*	*b04*	*b05*	*b06*	*b07*	*b08*	*b09*	*b10*	*b11*
EKUT	●	-	-	-	-	-	-	-	-	-	-	-
HT-Lab	●	○	○	○	-	-	○	-	○	○	○	-
HIFSuite[1]	N/A	m	m	m	m	m	m	m	m	m	m	m
zamiaCAD	●	●	●	●	●	-	●	-	-	●	-	●

- – the design was not translated

○ – the design was translated but not compiled

m – according to the tool manual data

● – the design was translated and compiled

[1] Access to demo version of the tool that implies very strict design size limits.

The EKUT's tool (V2SC-EKUT, Online) was able to translate *gcd* and the translation results were successfully compiled by a Microsoft Visual C++ based setup of SystemC. However, translation of the ITC99 benchmarks contained errors. The HT-Lab's tool (VH2SC, Online) was able to translate most of the designs but the compilation of them did not succeed. In this research, access was only to the demo version of the HIFSuite v2012.10 tool (HIF, Online) that implies very strict design size limits. The largest design allowed by this version that successfully translated was a 3-bit adder implementation. However, the user manual of this tool reports successful translation of all the ITC99 benchmarks listed in the table.

3.4 Chapter Conclusions

This chapter presented a framework and methodology to translate a VHDL model to cycle-accurate SystemC. The methodology has been verified by translating various benchmark VHDL cores, as well as partially translating and co-simulating with a processor implementation in VHDL. The SystemC output is emphasized to be human-readable and providing for clear correspondence to the source VHDL code, thus allowing further manual code changes and debug.

Future scope of work might include supporting hierarchical RTL designs, port maps, native translation of RTL switch-case to C++ switch-case, more regressive testing in addition to currently tested benchmarks like gcd1, ITC99 b01, b04, b06, and b09, and translation of Verilog designs to SystemC.

4 OPTIMIZATIONS FOR TRANSLATION AND CO-SIMULATION

SystemC model available from the previous chapter is obtained by translating the VHDL RTL on a line-by-line basis. This plain translation approach does not take into account the simulation characteristics as well as implementation style of the SystemC model. These considerations enable optimizing the translated SystemC models for higher simulation speed.

The current chapter explores various optimizations methodologies for the translated models as well as for models used in co-simulation. In addition to the plain VHDL-to-SystemC translation, there are possibilities of alternate implementations for a SystemC model. This chapter explores these alternate scenarios to get 25% better simulation speed. Additionally, VHDL and SystemC models are frequently co-simulated by architects as well as verification teams. This chapter explores optimization scenarios that affect the co-simulation performance, resulting in 20% faster co-simulation. The optimization methodologies in this chapter are relevant to IP design houses that need to provide high-speed simulation models and can be used for optimizing co-simulation tools as well system level models.

Section 4.1 presents the related work and Section 4.2 discusses the optimizations for the SystemC model translated from VHDL. Section 4.3 introduces the optimizations for the co-simulation of VHDL-SystemC model. Experiments and results are presented in 4.4 and section 4.5 summarizes this chapter.

4.1 Related Work

OSCI has provided a proof-of-concept simulator for SystemC LRM. Many attempts have been made to optimize the SystemC kernel, models, tools, etc. A number of techniques to enhance SystemC simulation time are suggested in (Alemzadeh et al., 2010). Optimizing memory partitioning, memory hierarchy, memory characteristics and allocation of data structures is proposed in (Brandenburg, Stabernack, 2013). Optimized simulation of SystemC RTL models on many-core architectures is discussed in (Roth et al., 2014), that is based on a configurable parallel SystemC kernel that preserves the partial order defined by the SystemC delta cycles while avoiding global synchronization as far as possible. Adaptive algorithms are able to intelligently adjust their behavior in light of the changing situation to achieve the best promising results. Optimized Network-on-Chip (NoC) architecture is presented in (Abba, Jeong-a, 2013). Technique for optimizing the simulation speed of a QEMU and SystemC-based virtual platform is presented in (Tse-Chen et al., 2010). An approach targeted to aid design exploration, early decision making in model refinement, optimization and tradeoffs is proposed in (Zaidi et al., 2011). SystemC-based simulation approach for fast performance analysis of parallel

software components, using source code annotated with low-level timing properties is discussed in (Stattelmann et al., 2011).

Co-simulation plays an important role in SoC system level design (SLD). Architects initialize system-design at a higher abstraction level, e.g. MATLAB, C, etc., then refine each component in a step-wise manner, while co-simulating with the rest of the system still at higher level. HW designers co-simulate their designs under development in a low level language e.g. VHDL/Verilog while the rest of the test environment, e.g. CPU, memory, bus, etc., is still in higher level language. SW developers typically co-simulate the instruction-set-simulator (ISS) at a higher level while the HW for which driver is being developed is simulated at a lower level. Hence co-simulation is important all over the SoC design cycle.

A co-simulation framework which enables rapid elaboration, architectural exploration and verification of virtual platforms made up of SystemC and Simulink components is presented in (Mendoza et al, 2011). A SystemC-based virtual prototype of a distributed controller implementation combined with high-level models of the plants specified in MATLAB/Simulink is proposed in (Glass et al., 2012). Approximate timed co-simulation has been proposed as a fast solution for system modeling at early design steps. This co-simulation technique allows simulating systems at speed close to functional execution, while considering timing effects. (Posadas et al., 2010) proposes a new embedded system modeling solution considering dual RTOS/GPOS systems.

(Xuexiang et al, 2012) present a uniform SystemC co-simulation methodology to describe the whole chip entirely with the same language. The processor model divides every instruction into a number of atomic operations, which makes it possible to accomplish fully cycle-accurate simulation. Meanwhile, the transaction-level-modeling communication model enables each hardware block to be built at different abstraction levels. (Bouhadiba et al, 2013) present a methodology that allows a coupled simulation of a SystemC/TLM model with a power and temperature solver. (Ming et al., 2011) present a fast cycle-accurate instruction set simulator (CA-ISS) for system-on-chip development based on co-simulation of QEMU and SystemC. Results show that the combination of QEMU and SystemC can make the co-simulation at the CA level much faster than the conventional RTL simulation, even with a full-fledged operating system up and running. (Kirchner et al., 2010) shows co-simulation of SystemC and Saber platforms using a proxy module that interfaces to the SystemC simulation and relays signals to Saber. (Cucchetto et al., 2014) presents a common co-simulation approach that works for integrating SystemC components with both QEMU and Open Virtual Platform (OVP).

4.2 Optimizations for Translation

In addition to the plain VHDL-to-SystemC translation, there are possibilities of alternate implementations for a SystemC model. This section explores various

implementation aspects to improve the simulation speed and discover potential optimization possibilities.

4.2.1 Combinational-statements

Combinational statements in VHDL are implemented outside of any VHDL process and are executed concurrently, in parallel. SystemC does not have this feature of concurrent statements, as all statements are executed within a SystemC process, either SC_METHOD or SC_THREAD. Since a VHDL combinational method does not have a 'wait' feature, only SC_METHOD is to be considered for translating combinational statements to SystemC.

Multiple combinational VHDL statements can be translated to SystemC SC_METHOD in the following 2 alternatives:

a) Single SC_METHOD for all combinational statements

This approach implements all VHDL combinational statements within a single SC_METHOD. The single SC_METHOD is made sensitive to all the source right-hand-side (RHS) elements in these statements. Table 4.1. (a) shows an example of this approach. Advantage of this implementation is its simplicity and ease of implementation. Drawback is that all the statements in the SC_METHOD are executed even if only 1 RHS element changes.

Table 4.1 Translation of combinational-statements

VHDL syntax	(a) Single SC_METHOD	(b) Multiple SC_METHOD
architecture behav of E is begin X <= A and B; Y <= not A; Z <= X or Y; end behav;	SC_METHOD(P1); sensitive << A << B << X << Y; void P1(void) { X = A && B; Y = !A; Z = X || Y; }	SC_METHOD(P_X); sensitive << A << B; void P_X(void) { X = A && B; } SC_METHOD(P_Y); sensitive << A; void P_Y(void) { Y = !A; } SC_METHOD(P_Z); sensitive << X << Y; void P_Z(void) { Z = X || Y; }

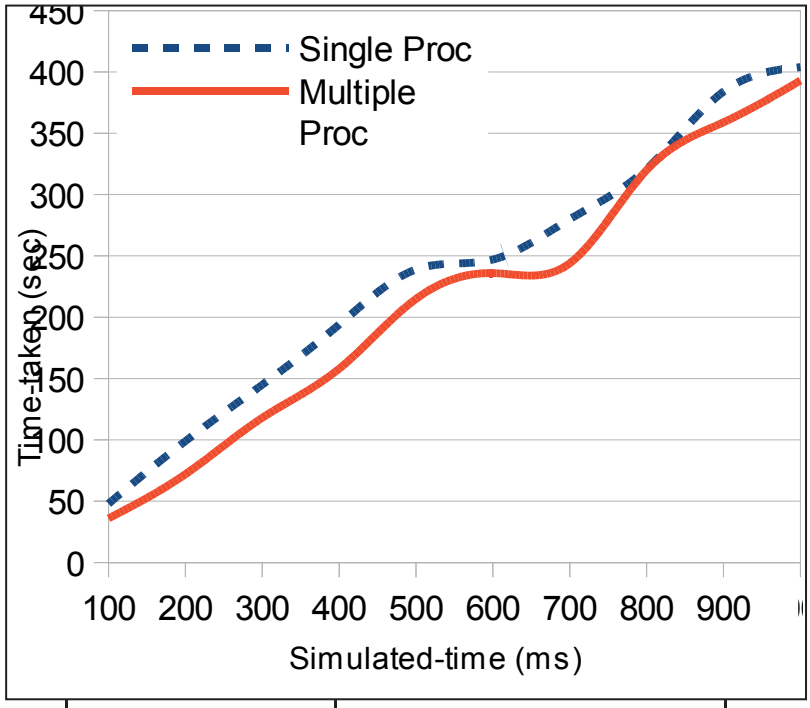
b) Separate SC_METHOD for each combinational statement

In this approach, each combinational statement is implemented in a separate SC_METHOD. This SC_METHOD is made sensitive only to the RHS elements appearing in source VHDL statement. As seen in an example in Table 4.1(b),

advantage of this approach is that only the relevant SystemC statements are executed, within a particular SC_METHOD. Limitation is the increase in the source-code size due to an SC_METHOD for each combinational statement.

4.2.2 Events

Events are used in SystemC to synchronize actions among processes. A SystemC process can wait for multiple events, suspending its task while in wait-state. Another process triggers the event at a later time during simulation. The waiting process wakes-up when it receives the notification of event-trigger and resumes its task. A process can be made to wait on a single-event or on multiple-events, as shown in Table 4.2. On the other hand, the event notification can be done immediately or in the next delta-cycle, as shown in Table 4.3.




```
. . .
ev_do.notify();
. . .
}
```

```
. . .
ev_do.notify(SC_ZERO_TIME);
. . .
}
```

Figure 4.1 Simulation-time for single and multiple processes

4.2.3 Results

To experiment the 2 alternative methods of translating the combinational-statements, the 'shifter' module in the Plasma core is selected. The 'shifter' module has 12 combinational-statements, making it a good candidate to observe the differences in simulation-time. Figure 4.1 shows the simulated-time taken by a model against the time-taken by a wall clock. The multiple process approach takes about 25% lesser simulation-time, resulting in faster simulation speed. This can be attributed to the fact that in multiple-process implementation, only a single statement is executed, resulting in optimized simulation model.

For analysis of events optimization, the modules ALU, DDR, DMA and MUX are implemented in alternate styles; e.g. ALU implementation is referred to as:

- ALU SE: Single-event waiting, immediate notification
- ALU ME: Multiple-event waiting, immediate notification
- ALU ME-Z: Multiple-event waiting, delta-cycle notification

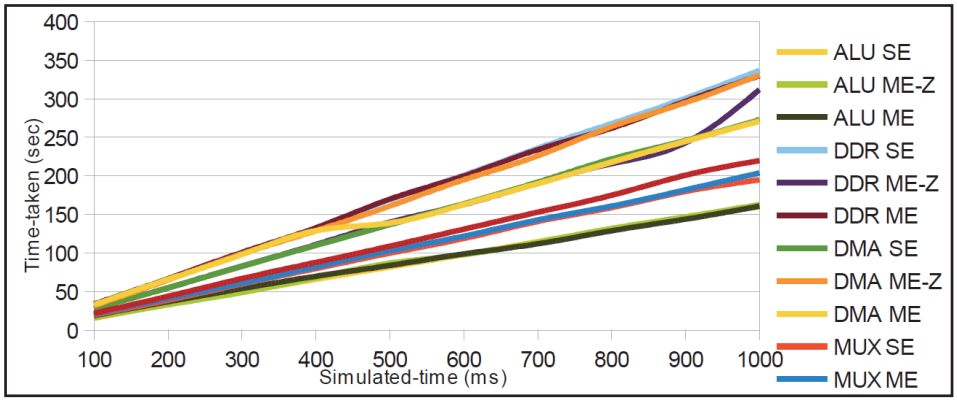


Figure 4.2 Simulation-time for various events implementation

Figure 4.2 shows the simulation-performance for ALU, DDR, DMA and MUX modules, for the 3 alternatives. As seen in Figure 4.2, there is no significant variation in the simulation-performance for a given module.

The single-event implementation is simpler in terms of coding, understanding and debugging. Hence, this might be a preferred implementation alternative, though based on coding aspect rather than the simulation speed.

4.3 Optimizations for Co-simulation

This section analyzes the performance of co-simulating VHDL and SystemC models. The aim of this analysis is to find possible optimizations in VHDL and SystemC models, as well as to recommend an effective co-simulation approach.

In order to validate the translation methodology described in the previous section, (Plasma, Online) RISC-core has been manually translated from VHDL to SystemC. Plasma is a MIPS-ISA compatible, 32-bit synthesizable RISC core, implemented in VHDL and available as open-source. Using translation rules described earlier, Plasma core has been successfully translated manually to SystemC. The translated cycle-accurate SystemC module was successfully co-simulated in the Mentor Graphics ModelSim environment with the original testbench accompanying Plasma source-code. The co-simulation produced an equivalent system behavior, proving that the VHDL-to-SystemC translation methodology is applicable even to complex design, like a processor-core.

Table 4.4 Plasma VHDL simulation-profile

Module	Simulation time (%)	Remarks
ALU	31.4	Highest simulation time
Register-bank	0.9	Lowest simulation time

Simulation profiling: Co-simulation performance analysis begins with profiling the VHDL simulation using the original testbench accompanying the Plasma source-code, to get the simulation-time and its %-age taken by each VHDL module. Table 4.4 shows that the ALU takes the highest (31.4%) whereas register-bank takes the lowest (0.9%) simulation-time.

Base performance analysis: VHDL-only simulation is taken as the base for performance analysis. The term VHDL-only implies the original Plasma core in VHDL, without any module in SystemC for co-simulation. The base performance will be used to compare against the co-simulation performance. As shown in Figure 4.3, the time-taken varies linearly with the simulation-time, ensuring reliable and consistent results for varying simulation times.

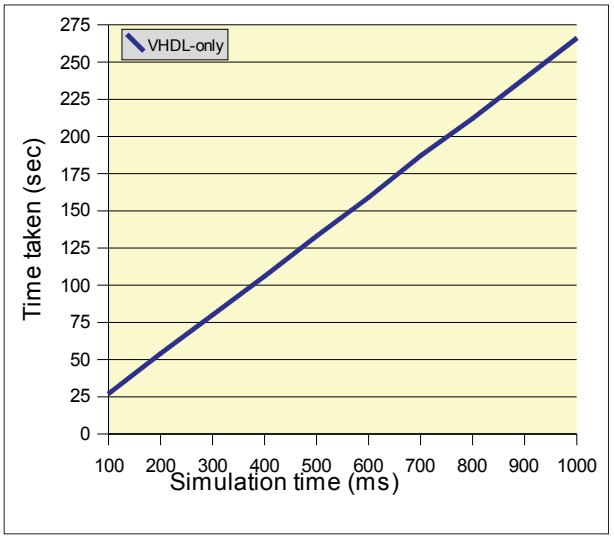


Figure 4.3 Plasma VHDL-only simulation

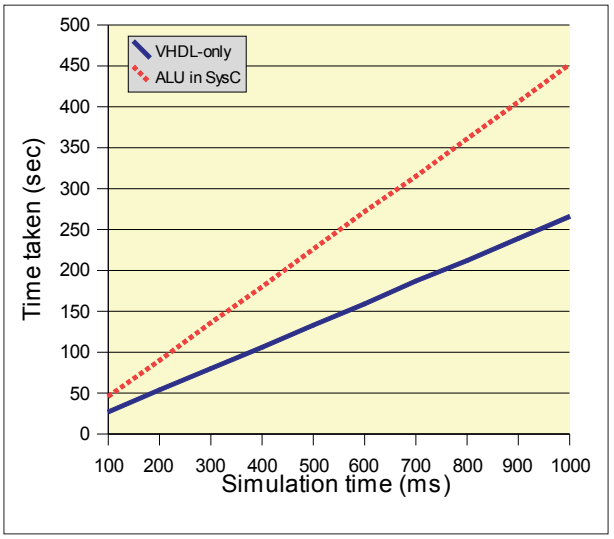


Figure 4.4 Co-simulation analysis for ALU

4.3.1 Co-simulating ALU in SystemC

As the ALU module is profiled in Table 4.4 taking the highest %-age of simulation-time, it is a good candidate for co-simulation performance analysis.

The ALU module is translated to SystemC and co-simulated with the rest of the Plasma core still in VHDL. Figure 4.4 shows the actual time-taken against the simulation-time. As expected, the variation of time-taken is linear. The co-simulation with ALU is taking more time than the VHDL-only simulation. The reason for lower performance of co-simulation is that both VHDL and SystemC simulation engines are now being invoked to perform the co-simulation, increasing the simulation time.

4.3.2 Co-simulating register-bank in SystemC

The least %-age of simulation-time is consumed by the register-bank. Hence, it is considered next for co-simulation performance-analysis. As earlier, the co-simulation setup consisted of the register-bank translated to SystemC while the rest of the Plasma design is in VHDL. The ALU module is also reverted back to VHDL from SystemC. Figure 4.5 shows the actual time-taken against the simulation-time, for both register-bank in SystemC as well as in VHDL.

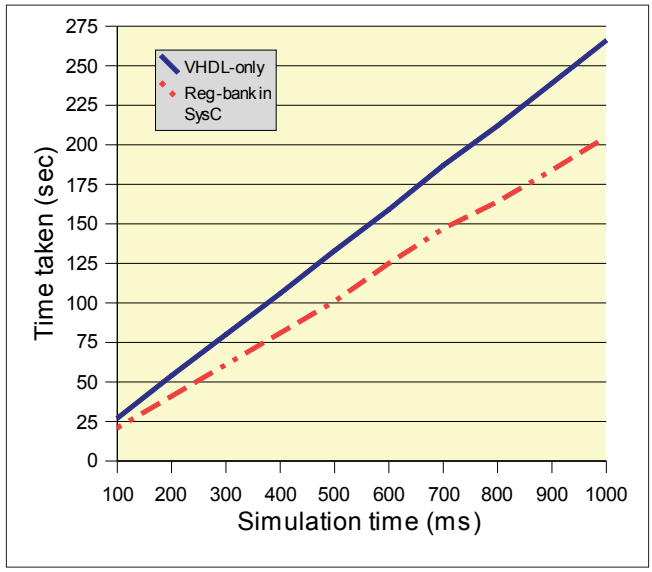


Figure 4.5 Co-simulation analysis for register-bank

4.3.3 Co-simulation Results

Significant and interesting observation in this case is that the register-bank in SystemC has better performance than the VHDL-only base performance. This is against the earlier ALU case which had lower performance than the base performance. The reason for better co-simulation performance of register-bank

in SystemC is due to the event-based design of the SystemC kernel. Event-based design invokes the SystemC kernel only when there is an event to execute the register-bank functionality. As earlier shown in Table 4.4, the register-bank takes negligible simulation-time, implying that there are almost no events for the register-bank module. Hence the SystemC kernel is infrequently invoked in this case, leading to a better performance than the VHDL-only scenario.

4.4 Chapter Conclusions

This chapter discussed about various optimization methodologies for the SystemC models translated from VHDL RTL. (1) Translation of VHDL combinational statements to multiple SystemC-processes takes about 25% less simulation-time than implementation with a single SystemC process. (2) SystemC synchronization strategies using single and multiple events take similar simulation time, although single-event based implementation is easier to develop and maintain. Hence, it is preferred to develop a model with single-event based synchronization. (3) Performance analysis of co-simulating VHDL with SystemC module taking the highest simulation time lowers the simulation performance. On the other hand, co-simulation with the module taking least simulation time is 20% faster than VHDL-only simulation. The optimization results obtained in this chapter are relevant to a wide SystemC community, including architects, designers, as well as IP design houses and EDA-vendors.

5 ABSTRACTION OF CLOCK INTERFACE

Clock interface is an essential part of a synchronous design. Various IP blocks in the design are fed by a single or multiple clock signals to make the design synchronous. Such clock signals are driven by a clock-generator at the system level. Maintaining the cycle-accuracy and updating each model based on its clock-sensitivity puts an enormous burden on the simulation kernel, resulting in slower simulation speed.

The current chapter addresses this problem by proposing 2 approaches based on minimization of delta-cycles and Finite State Machine with Datapath (FSMD) RTL abstraction to Algorithmic State Machine (ASM). SystemC scheduler employs delta-cycles and events for proper scheduling of various ports, channels, signals, interfaces, etc. Minimization of these delta-cycles and events in a SystemC description is the former idea behind abstraction of clock interface. Another methodology transforms an FSMD RTL design in VHDL to an equivalent ASM representation in SystemC with explicit separation of design functionality by states. Finally, the clock interface is abstracted up to optimize the simulation performance.

Section 5.1 discusses the related work and Section 5.2 introduces various modes of simulation. SystemC simulation mechanism is discussed in Section 5.3, as it is important to understand these concepts in order to abstract the clock interface. Section 5.4 introduces clock interface abstraction by minimization of delta-cycles. Abstraction based on ASM is proposed in Section 5.5. The manipulation details are demonstrated on a case study design and the first experimental results show simulation speed-up and prove feasibility of the proposed approach. Finally, Section 5.6 concludes this chapter.

5.1 Related Work

Discrete Event Simulation (DES) models the behavior of a complex system as an ordered sequence of well-defined events. Due to its generality, DES has been used is various domains. (Sumaryo et al, 2013) presents an improved DES model of traffic light control on a single intersection, developed using Simulink / SimEvent toolbox provided by MATLAB. (Smith et al, 1994) describes an application of DES for shop floor control for a flexible manufacturing system. (Chik et al., 2014) investigates simulation modeling methods for semiconductor fabrication factories from various publications since past 10 years. Primary simulation techniques reviewed in this analysis are discrete event, petri-net, gaming, virtual, intelligent, monte carlo and hybrid to understand individual strength and common usage in the market. The research concluded DES is the most suitable technique for simulating FAB operations. (Wong et al., 2012) explores various queuing methods in DES such as linked list, heap, splay tree and calendar based queue. Due to the design characteristic of discrete events

and single sequential queue in DES, these queuing methods are unable to fully utilize the computing power of multi-core processors. This paper proposes Multi-Dimensional Queue Mechanism (MDQM) to counter the stated problem and the experimental results show improvement of simulation run time by using the proposed mechanism when running DES on multi-core processors environments.

FSMD (Gajski, et al., 1992) representation is employed in various design tasks. Transformation on FSMDs for formal verification of sequential circuits is proposed in (Acosta et al., 2015), in which every conditional statement produce multiple sequences of states, being based on variable dependency. An automated framework for verification of low power transformations in RTL designs is presented in (Karfa et al., 2011). Both the input RTL and the transformed RTL are analyzed by a rewriting based method to obtain the FSMDs, followed by FSMD based equivalence checking method to establish equivalence between the RTLs. (Ahuja et al., 2009) presents a methodology to create abstract statistical power models for hardware co-processors and its utilization at system level for power estimation. The co-processors are realized as FSMD and co-simulated with simulation model of ARM processor.

There exists a number of flowchart-like models for functional design representation, e.g. flowchart (Samuel, 1976), FSM (Baranov, 2008), HGS (Sklyarov, 1999), BFSM (Takach, 1995). Hierarchical Finite State Machines (HFSM) find application in various domains, like implementation of network applications (Kramer et al., 2013), detection of human activities in retail surveillance (Trinh et al, 2011), control of Unnamed Aerial Vehicle (UAV) (Vitor et al, 2014), hardware implementation and optimization of recursive sequential and parallel algorithms (Mihhailov et al, 2010), etc.

This chapter uses the formalism defined in (Pong, 2006) as Algorithmic State Machine (ASM) chart, and goes beyond the existing state-of-the-art in the following aspects:

1. Develops a readable SystemC abstraction for FSMD designs.
2. Develops fully automated clock interface abstraction for RTL FSMD.
3. Transforms clock-edge triggering to static clock period usage, resulting in higher simulation speed.
4. Separates the design functionality into ASM blocks that enables functional abstraction of the model to cycle-inaccurate levels.

5.2 Modes of Simulation

This section introduces few models of digital simulation-kernel design and then elaborates on the DES that is the basis of SystemC kernel.

Discrete Event Simulation (DES) models a system in terms of sequence of discrete events. These events describe a change in the state of the system and hence, marks a time instant with respect to progress of system state, as in Figure 5.1. System is assumed to be in the same state between two consecutive events.

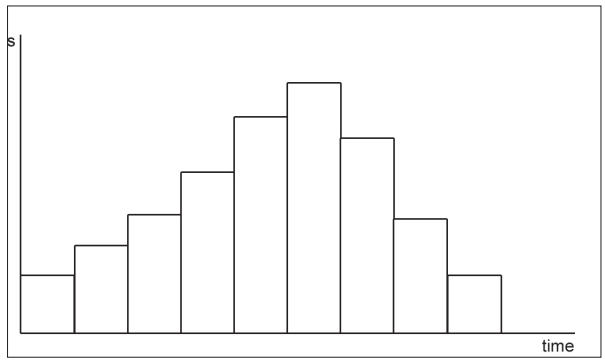


Figure 5.1 Discrete-event simulation model

Continuous Simulation models a system by continuously keeping track of the system activities and responses, as in Figure 5.2. Differential equations are typically used to model such responses in continuous domain. The time is broken in terms of time-slices so that system's activity in each time-slice can be simulated correctly. Due to keeping track of the activities in each time-slice, a continuous simulation is much slower than the discrete-event simulation.

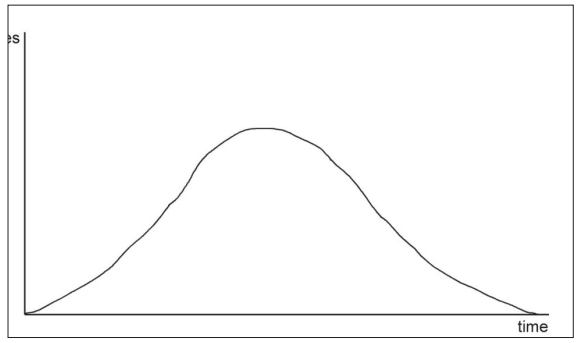


Figure 5.2 Continuous-event simulation model

Process-based Simulation represents a system as a set of processes. A process represents an activity in the system and can be simulated by a thread in the simulation-kernel. A thread can be in states like sleep, wake, resume, update, and cause other threads to change the states accordingly.

Monte-Carlo simulations are used to model the probability of different outcomes in a process that cannot easily be predicted due to the intervention of random variables. The technique was first developed by Stanislaw Ulam, a mathematician who worked on the Manhattan Project. He mentioned this to John von Neumann, and the two collaborated to develop the Monte Carlo simulation. In Monte Carlo simulation, the entire system is simulated a large number (e.g., 1000) of times. Each simulation is equally likely, referred to as a realization of the system. For each realization, all of the uncertain parameters are sampled. The system is then simulated through time (given the particular set of input parameters) such that the performance of the system can be computed. This results in a large number of separate and independent results, each representing a possible "future" for the system (i.e., one possible path the system may follow through time). The results of the independent system realizations are assembled into probability distributions of possible outcomes. As a result, the outputs are not single values, but probability distributions. Monte-Carlo simulation provides a number of advantages over deterministic, or "single-point estimate" analysis, such as probabilistic results, graphical results, sensitivity analysis, scenario analysis and correlation of inputs.

SystemC simulation-kernel is based on the DES in which the state of a system is represented by a set of variables that represent the properties of the system that are of interest to the user. A change in the state is represented by a change in any of the variables in the state-set. Time is modeled in a 'suitable' time-unit to represent the passage of time and each event represents the progress of time. Queue is used to maintain the simulation events. This queue typically consists of the events that have been generated by the earlier simulated event, but are yet to be simulated themselves. The queue also maintains the time at which the event was generated and may also mention the future-time in which event has to be simulated. Some events can be marked as instantaneous, in which case they are scheduled in the next trigger of the queue. The queue implements the priority scheduling, sorted by event time. This means that the events are picked in a chronological order, irrespective of the order in which they were placed in the queue. Scheduling of the discrete-events can be implemented in single-threaded or multi-threaded mode. Single-threaded simulation has just one 'current' event to simulate at a given time. Multi-threaded simulation might have many events marked as 'current' to be simulated simultaneously. Synchronization between events is a major design challenge in multi-threaded simulations. DES does not guarantee any particular order of simulation but can be re-played back consistently due to usage of pseudo-random numbers.

5.3 SystemC Scheduler

SystemC implements event-based scheduling for simulation. This allows SystemC to be used for varied scenarios, such as

- Untimed Functional (UTF): No notion of time while modeling functionality and interfaces. Both execution and data transport occurs in 0 time.
- Timed Functional (TF): Some notion of time is used for functionality and interfaces by modeling the latencies. Data transport takes positive time in such scenario.
- Bus Cycle Accurate (BCA): Interfaces are modeled cycle accurately, but not the functionality. Pin-level details are not yet modeled and transactions are used for data transport.
- Pin Cycle Accurate (PCA): Interfaces are modeled cycle accurately and the pin-level details are also accurate, functionality is not cycle-accurate.
- Register Transfer Level (RTL) Accurate: Everything is fully timed and cycle-accurate, by completely modeling the details of each clock-cycle and pin.

Due to event-based simulation, SystemC processes are executed based on their sensitivity to events and their outputs updated based on events. The SystemC kernel maintains an event-queue to describe the processes to execute at a given time. The evaluation order of runnable processes is implementation-defined in SystemC LRM (SystemC, 2011). A two-phase approach is used to evaluate and update the state of the simulation kernel, as described here.

After initialization, SystemC scheduler advances time to the earliest time step where there is a scheduled event notification. It adds all processes that are sensitive to that event (or other events at the same time step) into a list of "runnable" processes. It then resumes execution of every process in the list of runnable processes (one at a time) in a non-deterministic order. This is known as the evaluation phase. An immediate event notification can add processes to the list of runnable processes within the current time step, provided such processes are currently waiting for that event. Once the list of runnable processes is empty, the scheduler proceeds to the update phase where primitive channels (such as sc_signal) that have been written to, get updated. These primitive channels may be in the sensitivity list of other processes, in which case there is another iteration of the evaluate and update phases known as a delta cycle. This continues until there are no more events to process at the current time step, when the scheduler advances to the next time step.

SystemC simulator kernel employs Evaluate-Update mechanism, and functions as follows:

1. *Initialization: execute all processes in an unspecified order*
2. ***Evaluate**: Select a ready to run process and resume its execution. This may result in more processes ready for execution due to Immediate Notification.*
3. *Step-2 is repeated until no more processes to run.*
4. ***Update**: Execute all pending calls to update() method.*
5. *If 2 or 4 resulted in delta event notifications, go back to 2*
6. *Simulation is finished for current time if there are no timed events.*
7. *Advance to next simulation time that has pending events.*
8. *If no pending events then exit the simulation else go back to step 2*

SystemC uses a 64-bit unsigned integer 'sc_time' to represent the simulation time values. The resolution of time is set to default value of nanoseconds (SC_NS) and can be set by user to other values such as SC_FS, SC_PS, SC_US, SC_MS and SC_SEC.

The SystemC simulation kernel supports the concept of delta cycles. A delta cycle consists of an evaluation phase and an update phase. This is typically used for modeling constructs that cannot change instantaneously (e.g. primitive channels such as sc_signal). By separating the two phases of evaluation and update, it is possible to guarantee deterministic behavior. However, SystemC can model software, and in that case it is useful to be able to cause a process to run without a delta cycle (i.e. without executing the update phase). This requires events to be notified immediately.

When an SC_THREAD is declared, the threading library (pthreads or quickthreads) keeps track of a local stack pointer and local stack variables for that thread. When the SC_THREAD is suspended by calling wait(), it pushes the state of the processor onto the stack at that point. The process of storing the state so that another process can run is called a context switch, which does indeed take time. So SC_THREADs are slower than SC_METHODs.

Table 5.1 shows a sample RTL VHDL description and the equivalent SystemC description. As shown, SystemC description consists of a clock in-port (CLOCK), SC_THREAD named P1() that is sensitive to the clock in-port. The sc_main() routine, instantiates a clock-generator and connects it to the clock in-port of the module.

Table 5.1 Clock-based designs in VHDL and SystemC

VHDL	SystemC
entity b09 is port (clock: in bit;); end b09; architecture BEHAV of b09 is begin P1: process(clock) <snip> end process; end BEHAV;	class sc_WORK_B09_BEHAV : public sc_module { public: sc_in < bool > CLOCK; private: void P1(void); }; sc_WORK_B09_BEHAV :: sc_WORK_B09_BEHAV(sc_module_name mn_) : sc_module(mn_) { SC_THREAD(P1); sensitive << CLOCK; } #define SC_CLK_PERIOD 10 int sc_main(int argc, char **argv) { sc_clock my_sc_clk("my_sc_clk", SC_CLK_PERIOD, SC_NS); sc_WORK_B09_BEHAV my_mod("my_mod"); my_mod.CLOCK(my_sc_clk); sc_start(); return 0; } //end sc-main

5.4 Minimization of Delta-Cycles

Abstraction of clock interface while generating SystemC from VHDL is based on the concept of minimization of delta-cycles and events, as these affect the performance of the SystemC scheduler profoundly.

5.4.1 Recognizing clock interface

It is necessary to recognize the clock interface in a design in order to eliminate it for abstraction. A simple heuristic for recognizing the clock interface in an RTL VHDL is based on the following general observations:

1. Clock is 1-bit input port
2. Appears in the sensitivity list of a process
3. Appears with a 'event qualifier
4. Does not appear in any other control/data statement

5.4.2 Waiting on clock-period (OPT-A)

A fundamental property of a clock signal is its clock-period, i.e. interval between clock-pulses. Simple designs have static clock-periods, whereas a

complex design might employ varying clock-periods to achieve power savings. As shown in Table 5.1, an SC_THREAD waits on the arrival of clock-edge (positive, negative or both). This waiting requires the SystemC-scheduler to keep track of the changes in the clock signal and trigger appropriate SC_THREAD based on the activity on these clock signals. Load on the SystemC-scheduler can be reduced by specifying static clock-period for a wait statement inside an SC_THREAD, as shown in Table 5.1.

The SystemC-scheduler has to keep track of each signal specified in various sensitivity-lists and to trigger the appropriate SC_METHOD / SC_THREAD whenever there is an update on any such signal. Once an SC_THREAD has been made to wait on a static clock-period, the clock signal can be removed from its sensitivity-list, again to reduce burden on the SystemC-scheduler.

5.4.3 Removal of clock-generator (OPT-B)

Once all the SystemC modules are made independent of the clock signal by waiting on the static clock-period, the system level clock-generator is redundant. Even if there is no module working on the clock, the clock-generator keeps ticking and generating the clock-cycles. This is another source of burden on the SystemC-scheduler, consuming simulation-resources. Next optimization is to completely remove the system level clock-generator, as it is of no use now.

Note that this step needs to be done carefully, after completely analyzing the side-effects of this removal of the clock-generator. Strategies like formal proof of transformation correctness, regression testing without any breaks, etc. must follow the removal of the clock-generator to ensure that the system level design is still correct and functional.

5.4.4 Experiments

This section discusses the experiments and results for the approach discussed in the previous sections. A graphics pipeline for edge-detection is used for the validation of the clock interface abstraction discussed in the earlier section.

Figure 5.3 shows the graphics pipeline, which has Gaussian-blur and edge-detect operations performed on an image-frame being read from a file and the final image-frame being saved to the result file. SC_THREADS gauss_blur() and edge_detect() are sensitive to the signal sc_clk driving both the models.

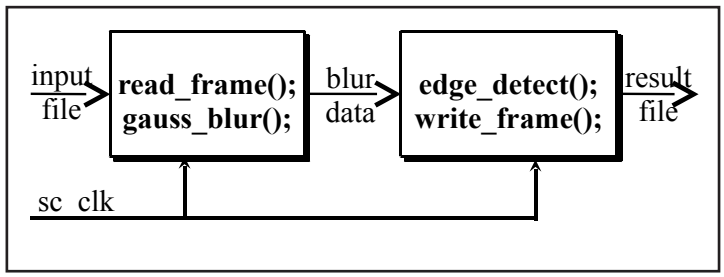


Figure 5.3 Graphics pipeline

Following setups are simulated for the above mentioned graphics pipeline:

1. BASE: Simulate the un-optimized SystemC description.
2. OPT-A: Both IP-models wait on the clock-period and the clock signal is removed from the sensitivity-list.
3. OPT-B: Remove the system level clock-generator.

Figure 5.4 shows the simulation-time for processing number of frames varying from 1 through 10, for the 3 simulation scenarios. The simulation-time taken by models increases with the number of processed frames. Scenario OPT-A takes about 10% lesser simulation time than the BASE scenario, due to waiting on static clock-period and abstraction of the sensitivity-list. Removal of system level clock-generator provides an optimization close to 38% as compared to the BASE, due to no more redundant clock-ticks being generated. This reduces burden on the SystemC scheduler, resulting in faster simulation speed.

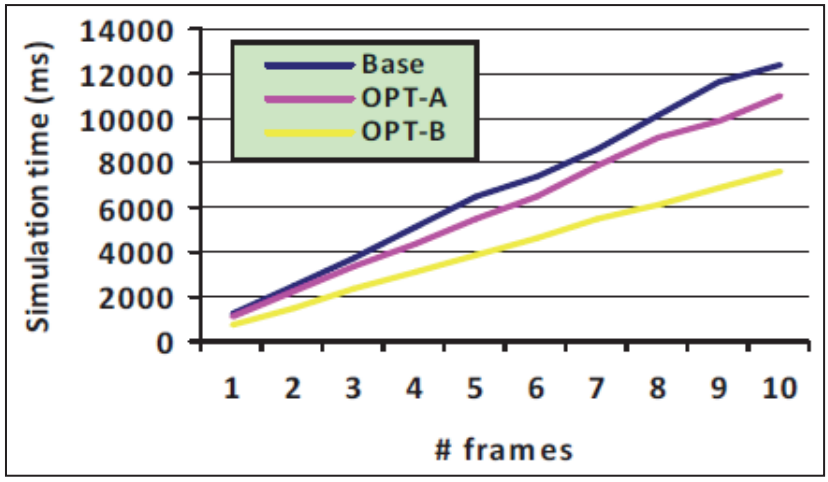


Figure 5.4 Simulation-time for frame processing

5.5 FSMD to ASM Transformation

This section discusses a manipulation approach that transforms an FSMD RTL design (Gajski et al., 1992) in VHDL to an equivalent ASM representation in SystemC with explicit separation of design functionality by states. Finally, the clock interface is abstracted to optimize the simulation performance. The manipulation details are demonstrated on a case study design and the experimental results show 16% to 21% simulation speed-up and prove feasibility of the proposed approach. Other advantages of this approach are:

- automated integration of FSMD RTL designs with system level models,
- ease of debugging with respect to other formal abstraction methods, and
- enables functional abstraction to higher, cycle-inaccurate levels.

5.5.1 Extended Finite State Machine (EFSM)

FSMD RTL descriptions can be represented by EFSM (Cheng et al., 1996). EFSM model allows a more compact representation of the state space than traditional FSM, thus, the risk of state explosion that incurs in modeling a large design by using FSMs is sensibly reduced.

Definition 5.1 *An EFSM is defined as a 5-tuple* M = <S, I, O, D, T >, *where:*

S : *set of states,*

I : *set of input symbols,*

O *: set of output symbols,*

D *: Cartesian product of sets* $D_1 \times \ldots \times D_n$,

T *: transition relation such that T :* $S \times D \times I \rightarrow S \times D \times O$.

A generic point in D is described by an n-tuple $x = (x_1,...,x_n)$; it models the values of the registers of the DUV. A pair <s, x> $\in$ S × D is called configuration of M. An operation on M is defined in this way: if M is in a configuration <s, x> and it receives an input i $\in$ I, it moves to the configuration <t, y> iff ((s, x, i), (t, y, o)) $\in$ T for o $\in$ O.

The EFSM differs from the classical FSM, since each transition does not present only a label in the classical form (*i*)/(*o*), but it takes care of the register values too. Transitions are labeled with an *enabling* function *e* and an *update* function *u*. An update function *u*(*x*, *i*) can be applied to a configuration $<s_1, x>$ if there is a transaction $t : s_1 \rightarrow s_2$, labeled *e*/*u*, such that *e*(*x*, *i*) = 1. In this case it is said that *t* can be *fired* by applying the input *i*.

5.5.2 Algorithmic State Machine Chart

This section introduces the Algorithmic State Machine (ASM) chart (Pong, 2006) that is used for functional design representation. Other models are flowchart, HGS, BFSM, etc. An advantage of ASM is a clear representation of FSM states by blocks. ASMs have three basic elements: state box (denoted by rectangles), decision box (denoted by diamonds) and conditional box (denoted by rounded rectangles) as illustrated in Figure 5.5.

In order to support EFSM's, ASM defined in (Pong, 2006) is improved by allowing register operations in both state and conditional boxes. *State boxes* are used to indicate states in EFSM, register operations and Moore-type output signals generated in the state. *Decision box* reflects the condition to be tested where exit paths represent true and false evaluations. In *conditional boxes* inputs come from one of exit paths of decision boxes and they contain register operation or Mealy-type output signals dependent on the inputs generated during the state. *ASM block* is a structure consisting of one state box, all decision and conditional boxes associated with its exit paths. Such block has one entrance and any number of exits paths. Each ASM block is dedicated to a state of EFSM and is traversed within one clock cycle. ASMs provide for explicit separation of the RTL FSMD functionality performed by one state of EFSM.

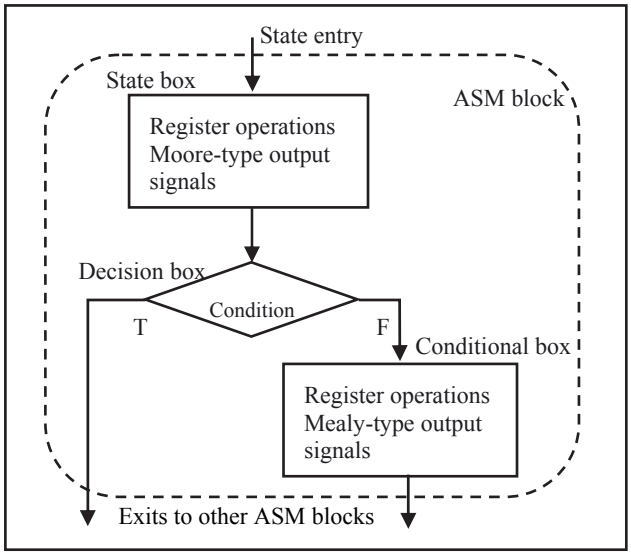


Figure 5.5 State block in the ASM chart

5.5.3 Transformation Methodology

This section describes the methodology performed in the following 2 steps:

Step A: Transformation of an EFSM representation of VHDL FSMD RTL design to an equivalent ASM.

Step B: High-performance SystemC implementation of the ASM from Step-A.

The above 2 steps are described in details in the next paragraphs.

Step A. Transforming EFSM to ASM

The following rules procedure of transformation of an EFSM to an equivalent ASM relies on the rules defined below:

1. Each EFSM state $s_i \in S$ is represented by an ASM block $b_i \in B$, where S is a set of all states s_i in the EFSM and B is a set of all blocks b_i in the ASM.
2. Transitions entering an ASM block b_i correspond to the ones entering the corresponding EFSM state s_i.
3. Similarly, transitions exiting an ASM block b_i correspond to the ones exiting the corresponding EFSM state s_i.,
4. Comparison operations at the *Decision boxes* within an ASM block b_i are derived from the enabling functions e at the transition arcs exiting from an EFSM state s_i.
5. In the case of a Mealy machine, behavior specified by the *condition box* of an ASM block b_i is equivalent to that of the update function u of the corresponding transition exiting EFSM state s_i.
6. In the case of a Moore machine, behavior specified by the *state box* of an ASM block b_i is equivalent to that of the update function u of the corresponding EFSM state s_i.

Figure 5.6 shows a sample EFSM, as a hybrid of Mealy and Moore machines, and its conversion to an ASM chart. FSM states *s0* and *s1* are transformed to ASM blocks *s0* and *s1*. Input *i1* is used to transition from *s0* to *s1*. The decision box checks the value of *i1* in order to either remain in *s0* or to move to state *s1*. While going to *s1*, output *o2* is also asserted, both in EFSM and in ASM.

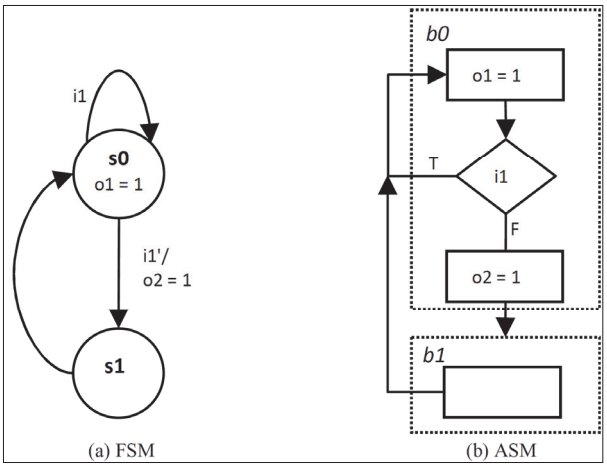


Figure 5.6 Sample conversion from FSM to ASM

Step B. SystemC models of ASMs

This section defines the methodology for an efficient SystemC model of the ASM chart. The methodology is generic and can to be applied to any FSMD design represented by an equivalent ASM chart. The discussed methodology starts with an ASM chart and finally produces a simulation speed optimized SystemC model of it.

Steps in the proposed methodology are the following:

1. For each ASM-block *b*, define a method *m*. Each method *m* is implemented as a C/C++ function.
2. Define an sc_event *v* for each method *m.*
3. Declare each C method *m* as a SC_METHOD, and make it sensitive to corresponding SystemC-event *v*
4. All SC_METHODs are declared as dont_initialize(), except the SC_METHOD that corresponds to the FSM's initial-state. This ensures that the simulation starts with the SC_METHOD that models the EFSM initial-state.
5. Whenever there is a transition from one ASM block to another ASM block, trigger the SystemC-event *v* after the system-wide clock period.
6. Due to the above Rule-5, no SC_METHOD is made sensitive to the clock input signal.

The above methodology results in an efficient SystemC implementation of an ASM chart due to the following optimizations:

- Usage of SC_METHOD that is faster in simulation than SC_THREAD, due to lower overhead of context switching.
- Triggering of SC_METHOD using static global clock period, rather than dynamically waiting on clock edge. This lowers burden on the simulation kernel.

5.5.4 Case Study

This section applies the above discussed methodology to transform a VHDL FSMD RTL implementation of a Greatest Common Divisor (GCD) design to the corresponding ASM chart and further to its SystemC implementation optimized for high performance.

EFSM for the GCD design is shown in Figure 5.7. As a result of application of Step-A, the EFSM is transformed to ASM-chart as shown in Figure 5.8.

Next, Step-B of the methodology is applied to the GCD ASM as follows:

1. Set $B = \{ST0, ST1, ST2\}$
2. Methods for each ASM block: $M = \{m_{ST0}, m_{ST1}, m_{ST2}\}$

e.g. `void m_ST0(); void m_ST1(); void m_ST2();`

3. SystemC event for each method m: $V = \{v_{m\_ST0}, v_{m\_ST1}, v_{m\_ST2}\}$

e.g. `sc_event ev_m_ST0, ev_m_ST1, ev_m_ST2;`

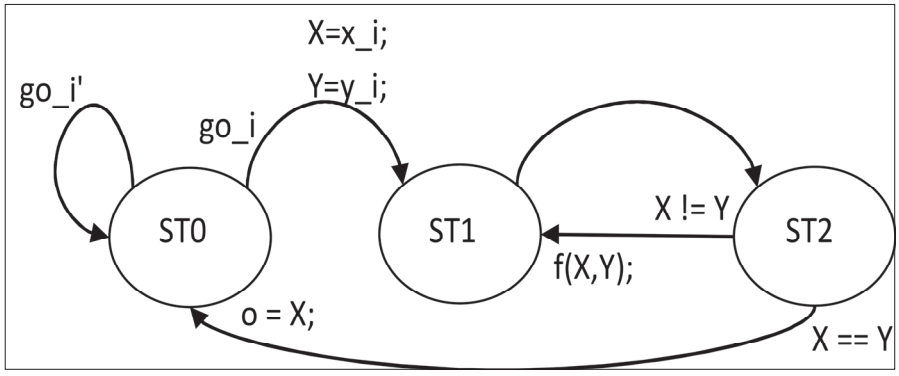


Figure 5.7 EFSM for GCD

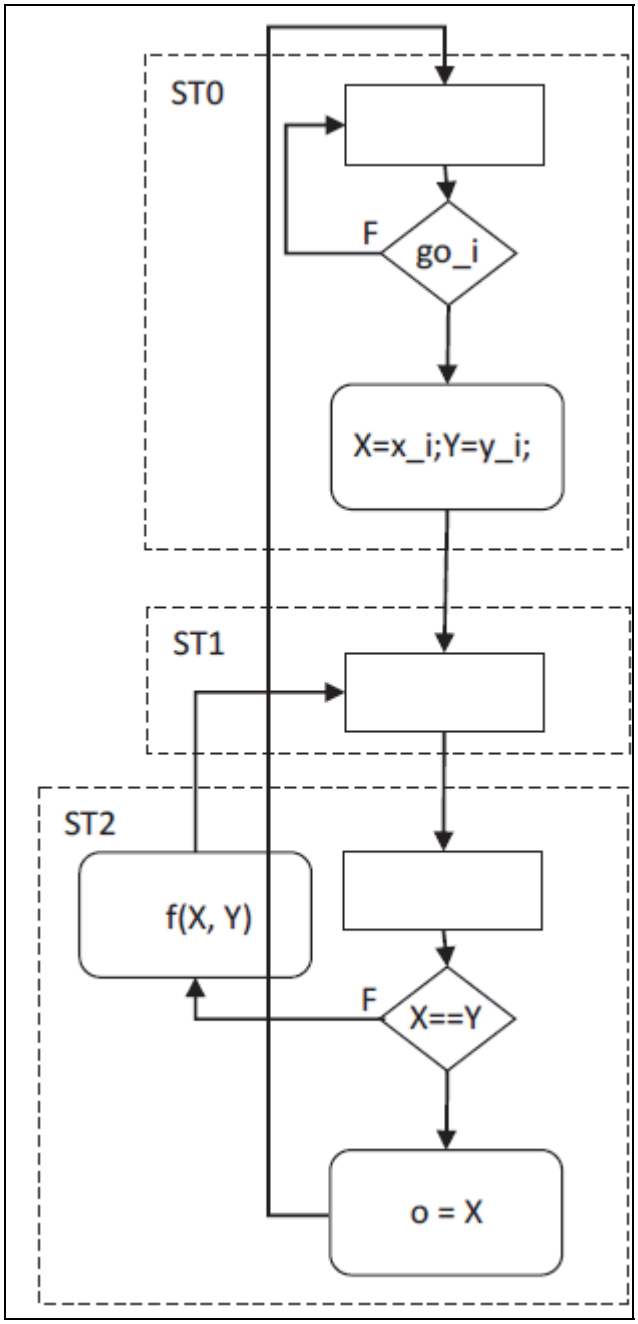


Figure 5.8 ASM chart for GCD

4. Declare each method $\{m_{ST0}, m_{ST1}, m_{ST2}\}$ as an SC_METHOD, and make it sensitive to corresponding sc_event $\{v_{m\_ST0}, v_{m\_ST1}, v_{m\_ST2}\}$

5. All SC_METHODs are declared as `dont_initialize()`, except the SC_METHOD that corresponds to the EFSM's initial state.

e.g.

```
SC_METHOD(m_ST0);       sensitive << ev_m_ST0;
SC_METHOD(m_ST1);       sensitive << ev_m_ST1;
                        dont_initialize();
SC_METHOD(m_ST2);       sensitive << ev_m_ST2;
                        dont_initialize();
```

6. Trigger the SystemC-events $\{v_{m\_ST0}, v_{m\_ST1}, v_{m\_ST2}\}$ after the system-wide clock-period.

e.g. `ev_m_ST0.notify(SC_CLK_PERIOD, SC_NS);`

These steps transform GCD FSMD into corresponding ASM chart and further into high performance SystemC implementation.

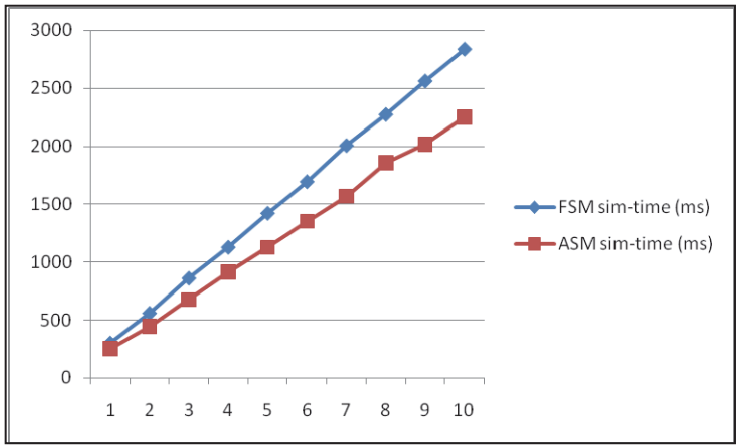


Figure 5.9 Simulation-time for FSMD and ASM models

5.5.5 Experimental results

This section presents the simulation performance comparison for the GCD design's initial RTL FSMD implementation compared against the abstracted ASM-based functional model. Both models were described in SystemC and tested by the same testbench using the OSCI SystemC kernel simulated in the

Microsoft VC 2010 Express environment. ModelSim was not used as there is no RTL design involved in this simulation.

Figure 5.9 demonstrates the CPU time required for the simulation of both the models for 1 to 10 million SystemC clock-cycles on an Intel i5 @2.6GHz, 8GB-RAM PC. The simulation speed-up achieved is in the range of 16.4 to 21.7%.

5.6 Chapter Conclusions

Clock interface is an essential part of a synchronous design, that puts an enormous burden on the simulation kernel. Abstraction of clock interface is a positive step in the improvement of the simulation-performance. The chapter introduces various modes of digital simulation, followed by a discussion on the design of the SystemC simulation-kernel. As the discrete-event simulation is implemented by the SystemC kernel, the chapter proposes an approach for clock interface abstraction based on the minimization of delta-cycles. This is achieved by replacing the clock interface based triggering of SystemC-threads by clock-period based triggering. Then the chapter presented a novel approach to automated FSMD RTL design manipulation for clock interface abstraction. The manipulation approach takes FSMD RTL design in VHDL as an input and transforms it to an equivalent ASM representation in SystemC with explicit separation of design functionality by states. Finally, the clock interface is abstracted to optimize the simulation performance.

Clock interface abstraction by minimization of delta-cycles resulted in 38% simulation speed improvement on the tests conducted in this research. Experiments for abstraction based on FSMD-to-ASM transformation resulted in 20% optimization in the simulation speed.

An additional advantage of separating design functionality by states is that it enables functional abstraction of the model to higher, cycle-inaccurate levels, which will be the basis for next chapter.

6 FUNCTIONALITY ABSTRACTION BY LOOSE MODELS

The current chapter proposes an approach for automated abstraction of the functionality of a cycle-accurate SystemC model to untimed, using a novel concept of SystemC-based Loose Models (SCLM). SCLMs provide for an instrument to neglect design model parts irrelevant for particular manipulation step of the abstraction process, thus simplifying the abstraction flow. As a result, the computational complexity of the abstraction process is reduced, thus increasing the overall scalability. The proposed abstraction flow is demonstrated on a set of benchmark designs and the experimental results prove feasibility of the proposed approach and also show considerable simulation speed-up.

Section 6.1 discusses the related work and Section 6.2 introduces the concept of the SCLM. Section 6.3 defines the rules for abstraction using SCLM, applied to testbench abstraction in Section 6.4 and on a cycle-accurate design in Section 6.5. Results are discussed in Section 6.6 and Section 6.7 concludes this chapter.

6.1 Related Work

(Gajski, 2003) is amongst the initial works on increasing levels of abstraction, or in other words, increasing the size of the basic building blocks. It argues for the modeling language to be C since standard processors come only with C compilers. But the C language was not developed for describing the hardware and lacks basic constructs for expressing hardware concurrency and communication among components. It lays the importance of a well-defined design flow, with well-defined models and a modeling language that can be compiled with standard compilers and that is capable of modeling hardware and software on different levels of abstraction including cycle-level accuracy. (Zhao, 2002) mentions that a new methodology of modeling, simulation and synthesis is needed based on standard RTL semantics in order to improve the productivity of current RTL design practice. It presents the implementation of a C++ class library for RTL modeling and simulation, that provides a foundation for experimentation in the new RTL semantics.

(Calazans et al., 2003) presents a comparison of traditional RTL modeling and TLM through the implementation of a simple processor case study. The R8 processor was described in SystemC TLM and RTL versions and these were compared to an equivalent hand-coded VHDL RTL description in some key points, such as simulation efficiency and implementation results. (Patel et al., 2007) introduces an extension of SystemC, called Bluespec-SystemC (BS-ESL), to manage shared state concurrency using multi-threading in large SystemC models. However, for simulating a model that is partly designed in SystemC and partly using BS-ESL, an interoperability semantics and

implementation of such a semantic is required. The paper formalizes the simulation semantics of BS-ESL and discrete-event simulation of RTL SystemC and provide a solution based on this formalization. A semi-hardware description language called VeriC (Verilog and C) to bridge ESL and RTL is proposed in (Shu-Hsuan et al., 2009). Like SystemC, VeriC is based on C++ and not only describes the design by a syntax very close to Verilog, but also it can model the target design at both pin and cycle accuracy with an implicit clock mechanism. Those VeriC modules can be directly translated to RTL and also can be interoperable with other modules in C/SystemC/Verilog languages by hybrid simulation.

(Bombieri et al., 2006) discusses a methodology for abstracting RTL descriptions to different TLM levels. (Bombieri-a et al., 2010) proposes a methodology to automatically generate SW code by abstracting RTL IP models implemented in hardware description language (HDL). The methodology exploits an abstraction algorithm to eliminate many implementation details typical of the HW descriptions, in order to improve the performance of the generated code. (Tasker et al., 2006) focuses on advanced techniques to cope with the complexity of designing modern digital chips which are complete systems often containing multiple processors, complex IP blocks and high-speed buses and interconnection networks. It focuses on language facilities and synthesis techniques that dramatically simplify and shorten the process of correct chip design by raising the level of abstraction on multiple dimensions without sacrificing final hardware quality.

(IP-XACT, 2014) formulates the conformance checks for eXtensible Markup Language (XML) data designed to describe electronic systems and their abstraction levels of bus interfaces and connections. A set of XML schemas of the form described by the World Wide Web Consortium (W3C) and a set of semantic consistency rules (SCRs) are included.

As discussed above, there have been several works on translating as well as on abstracting RTL models to SystemC. However, this chapter goes beyond the existing state-of-the-art in the following aspects: 1) It eliminates the need for semantic analysis and elaboration of the source RTL IP. 2) It reduces the model size for manipulations, thus contributing to higher scalability. 3) It provides a readable SystemC abstraction for RTL designs.

Different from related abstraction approaches e.g. (Bombieri et al, 2011), the proposed methodology provides for an instrument to neglect design model parts irrelevant for particular manipulation step of the abstraction process, thus simplifying the abstraction flow. As a result, the computational complexity of the abstraction process is reduced, thus increasing the overall scalability.

6.2 Loose Model Definition

In order to facilitate understanding the concept of a Loose Model, let a *strong model* be a formal model representing the complete functionality of a design.

For instance, Extended Finite Sate Machine model (EFSM) is a strong model of a code entity because it explicitly represents the complete functionality of the latter. A *loose model* of a code entity is a formal model that is restricted to *key elements* only, i.e. the constructs sufficient for a specific manipulation task. For each key element, there can be linked a part of the code entity named *a load element* that is not directly relevant to the manipulation.

With this understanding, a loose model in the current context is explained by the following basic definitions.

Definition 6.1. A *code entity C* is a formal textual description of a system in a programming language or a hardware description language (e.g. SystemC, VHDL).

Definition 6.2. A *parse tree* (i.e. concrete syntax tree) $P(N,B)$ of a code entity C is a tree consisting of a set of nodes $n_i \in N$, with a single root node $n_0 \in N$, and a set of branches $b_k \in B$ connecting the nodes. The set of nodes N, in turn, is partitioned into the set of internal nodes N^I and a set of terminal nodes N^T. The terminal nodes N^T of the parse tree P are strings forming the atomic constructs of the language of the code entity.

Definition 6.3. *A loose model M* is a formal model *generated* from the parse tree P of an initial code entity C. The formal model M consists of elements $k_j \in K$ generated by a *key pattern matching function* $\Phi(P)$. Elements $k_j \in K$ are referred to as the *key elements* of *M*.

In the general case, for each key element, a part of entity C referred to as a *load element* may be linked. Load elements $l_j \in L$ are strings generated from the terminal nodes of the parse tree P identified by a *load pattern matching function* $\Psi(P)$ that is executed locally starting from the sub-trees of P identified by $\Phi(P)$.

Definition 6.4. The key elements K are the basis for the *manipulation steps* on the loose model M resulting in a new loose model M'. After one or multiple manipulation steps the loose model M' is *restored* into a new code entity C'. During the restoration, the key elements K are transformed to respective language contracts that are combined with the load elements L.

Definition 6.5. A *SystemC based Loose Model (SCLM)* is a loose model generated from a system design model (i.e. a code entity) expressed in the SystemC language. An SCLM represents particular aspects of the implementation relevant to the given manipulation task on a system design model.

Loose models are exploited to reduce computational complexity and size of the model, where manipulation is performed on and may be subject to incomplete formal analysis (e.g. verification). As a wide scope of practical tasks in the domain, functional correctness of manipulations performed based on loose models can be verified using simulation based approaches or to a limited extent

using partial formal verification. In the current chapter, manipulations on the loose model are applied to perform abstraction from RTL to a higher level.

For instance, different SCLMs can describe the same system design implementation (code entity) at different levels of abstraction, with different coding styles/flavors or may rely on different key elements. The same system design implementation may be represented by SCLM such as clocked RTL FSM and by SCLM as untimed algorithm-level flow-chart. SCLM and its key elements sets of states and transitions may be appropriate for particular manipulations (e.g. optimizations) while SCLM may be appropriate for a different set of tasks. An illustration of a SystemC-based Loose Model is provided in Figure 6.1.

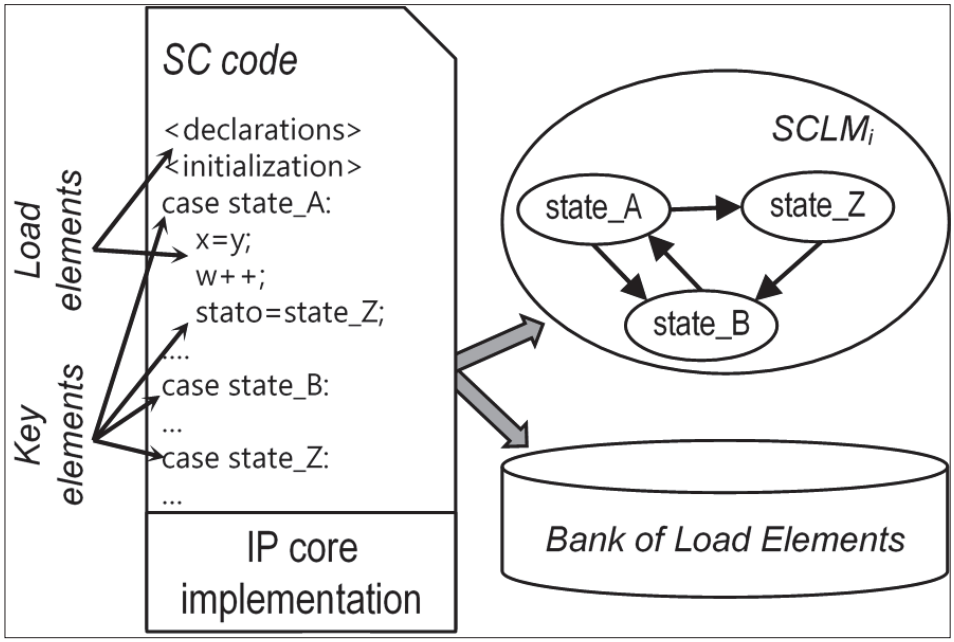


Figure 6.1 Example of SystemC-based Loose Model

6.3 Abstraction Using SCLM

In this methodology, cycle-accurate SystemC is assumed as the starting point of the abstraction process. While many RTL IP cores are implemented in VHDL or Verilog, the flow can be complemented by their automated translation to SystemC (as in Chapter 3).

The loose modeling based abstraction flow is shown in Figure 6.2 and it starts with the following steps:

1. Generate a parse tree P of the cycle-accurate SystemC IP core (SystemC FSMD RTL clocked).
2. Identify the key elements from the parse tree, by traversing the tree, and applying a key pattern matching function Φ in order to obtain key elements and load elements.
3. Identify the load elements (LE) for all the key elements applying the load pattern matching function Ψ.

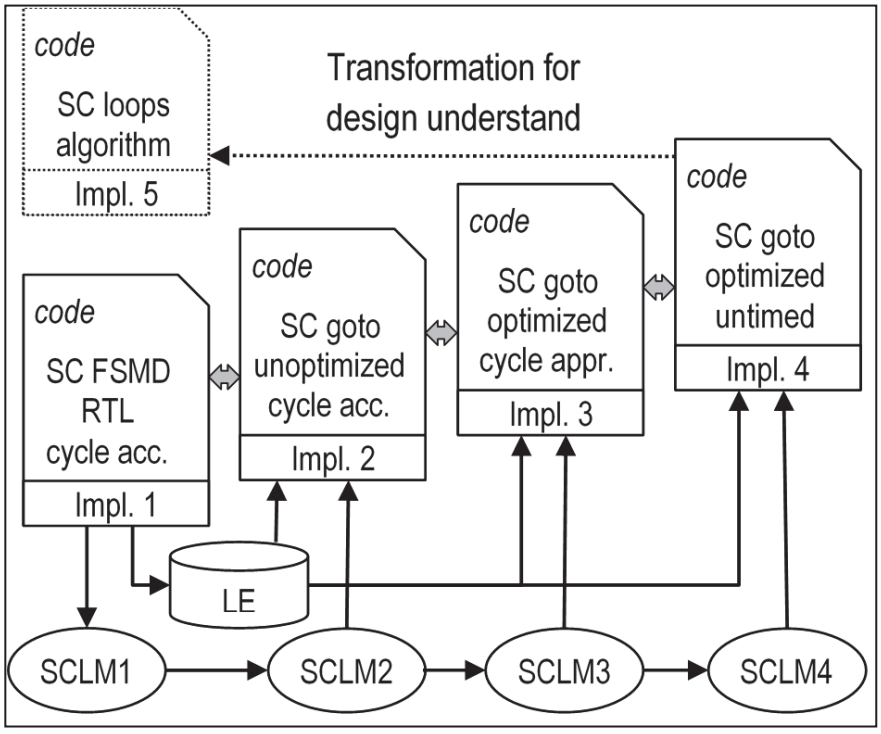


Figure 6.2 Automated RTL to TLM abstraction flow based on SCLMs

Based on the key elements, a loose model M is created, which maintains the links to the load elements. The above steps abstract an FSMD representation (i.e. FSM with Datapath embedded) to generate a loose model, SCLM1, generated from the key elements of the cycle-accurate code entity.

Following definitions are the basis for further manipulations:

Definition 6.6. A *GOTO (GT) model* uses the *goto* construct of the C language to transfer control to the various key elements in the model.

Definition 6.7. An *Optimized GOTO (GT_O) model* refers to a model in which the compute-only key elements are optimized to not relieve the control back to the simulation kernel. Communication with the other models in the system is not affected due to this optimization as the key elements with input/output statements remain as in the original GOTO model.

Definition 6.8. An *Untimed Optimized GOTO (GT_O_U) model* is a model where functionality is completely untimed and only final results are available for consumption by other models in the system.

Further manipulations and optimizations are performed as follows:

a) SCLM1 is manipulated to obtain a SystemC goto (GT) model SCLM2. Assuming that the SCLM1 is implemented as an SC_THREAD using the switch-case construct for the FSMD representation, SCLM2 is obtained by annotating each state by a *label:* and replacing the state assignment statement by a corresponding *goto label* statement. The SystemC wait() statement, used in the SC_THREAD, is replicated after each *label:* annotation to preserve the timing accuracy of the SCLM2.

b) Optimized goto (GT_O) model SCLM3 is derived from SCLM2 by replacing the *goto label* statement to a compute-only state by the functionality of that state. This optimizes the SystemC wait() statement for a compute-only state, resulting in a cycle approximate optimized goto model. Communication with the other designs in the system is not affected due to this optimization as the states with input/output statements remain as they were in the original SCLM1 GT model.

c) Finally, timing within SCLM3 is abstracted to obtain an untimed goto (GT_O_U) model SCLM4. All the SystemC wait() statements are removed and special synchronization signals are used to trigger the model and read the final result(s) back from the model.

Any of these loose models may be recovered to a SystemC code by replacing the loose model by SystemC constructs and substituting the loose model's links by SystemC code corresponding to load elements.

All of the abstraction levels obtained by the proposed abstraction approach are based on goto models, which can be easily transformed to a readable structured form for design understand by applying automated recognition of control loops

(e.g. WHILE, FOR, …). The resulting SystemC code, SCLM5, would represent the algorithmic level description of the IP core. Here, the approaches proposed since publication of the Structured Program Theorem (Corrado, 1966) are relied upon.

Note, that the extent up to which the abstraction can be performed is dependent on the environment interacting with the IP core. For a cycle accurate environment, goto model or optimized goto model can be utilized, whereas for untimed environments the abstraction may proceed until the untimed optimized goto model.

6.4 Case Study

The proposed abstraction approach is demonstrated on the b07 benchmark, with parts of the design code omitted for simplicity and better understanding. Table 6.1 illustrates model implementations in SystemC, and visualizations of their SCLMs is shown in Figure 6.3. For this case study, the source code relevant to the three states of the b07 FSM (START, LOAD_X and INCREMENTA) is considered. The first column in Table 6.1 shows RTL FSMD implementation of the b07. It is based on design states as the key elements, the switch statement is executed on positive edge event of the clock signal. $SCLM_{FSMD}$ is further transformed to $SCLM_{GT}$, where the key elements are goto-labels. A recovered SystemC code (also utilizing necessary load elements) is provided in the second column. As the next step, a manipulation on $SCLM_{GT}$ is performed to optimize out labels related to computation only and not involved in communication (i.e. input/output read/writes in the given design). Thus, column 3 optimizes out LOAD_X label as compared to the column 2. An optimized $SCLM_{GT\_O}$ and its recovered SystemC code are demonstrated in the third column. The load elements linked to the optimized out LOAD_X label part are recovered as a part of the START label code.

These three SCLMs for b07 benchmark IP core are using clock-based synchronization also in the computation part. However, the first two SCLMs are cycle-accurate, whereas the third one is cycle-approximate and here the semantics of the clock signal *clk* is different.

Table 6.1 Code snapshots for abstraction of b07 benchmark IP design

IP_{FSMD}: SystemC RTL FSMD (cycle-accurate)	**IP_{GT}: SystemC goto code (unoptimized, cycle-accurate)**	**IP_{GT_O}: SystemC goto code (optimized, cycle-approximate)**	**$IP_{GT\_O\_U}$: SystemC goto code (optimized, untimed)**
SC_MODULE(sc_b07) { sc_out<**int**>punti_retta; sc_in<**bool**>start,reset, clk; *<snip1>* SC_THREAD(proc_fsmd); sensitive << reset << clk; **void** sc_b07::proc_fsmd() *<init>* *<snip2>* wait(clk.posedge_event()); **switch**(stato) { **case** S_START: **if**(start.read()==1) *{<snip3>* stato =S_LOAD_X; } **else** {stato =S_START; *<snip4>*} break; **case** S_LOAD_X: x = mem[mar]; *<snip5>* Stato =S_INCREMENTA; break; **case** S_INCREMENTA: **if** (mar != lung_mem) *{<snip6>* mar = (mar+1) % 16; stato = S_LOAD_X;} **else** {**if**(x==2) *{<snip7>* punti_retta = cont + 1; stato=S_START;} **else** { stato=S_INCREMENTA;}}} *<snip8>*	SC_MODULE(sc_b07) { sc_out<**int**>punti_retta; sc_in<**bool**>start, reset, clk; *<snip1>* SC_THREAD(proc_GT_unopt); sensitive << reset << clk; **void** sc_b07:: proc_GT_unopt () *<init>* *<snip2>* label_START: wait(clk.posedge_event()); **if**(start.read()==1) *{<snip3>* goto label_LOAD_X;} **else** {<snip4> goto label_START; } label_LOAD_X: wait(clk.posedge_event()); x = mem[mar]; *<snip5>* goto label_INCREMENTA; label_INCREMENTA: wait(clk.posedge_event()); **if** (mar != lung_mem) *{<snip6>* mar = (mar+1) % 16; goto label_LOAD_X;} **else** {**if**(x==2) *{ <snip7>* punti_retta = cont + 1; goto label_START;} **else** { goto label_INCREMENTA;}} *<snip8>*	SC_MODULE(sc_b07) { sc_out<**int**>punti_retta; sc_in<**bool**>start, reset, clk; *<snip1>* SC_THREAD(proc_GT_opt); sensitive << reset << clk; **void** sc_b07:: proc_GT_opt () *<init>* *<snip2>* label_START: wait(clk.posedge_event()); **if**(start.read()==1) *{<snip3>* } **else** { <snip4> goto label_START;} x = mem[mar]; //from LOAD_X *<snip5>* label_INCREMENTA: wait(clk.posedge_event()); **if** (mar != lung_mem) *{<snip6>* mar = (mar+1) % 16; } **else** {**if**(x==2) *{ <snip7>* punti_retta = cont + 1; goto label_START;} **else** { goto label_INCREMENTA;}} *<snip8>*	SC_MODULE(sc_b07) { sc_out<**int**>punti_retta; sc_in<**bool**>start; *<snip1>* SC_THREAD(proc_GT_opt_ut); sensitive << sync; **void** sc_b07:: proc_GT_opt_ut () *<init>* *<snip2>* label_START: wait(sync.posedge_event()); **if**(start.read()==1) *{<snip3>* } **else** { <snip4> goto label_START;} x = mem[mar]; //from LOAD_X *<snip5>* label_INCREMENTA: **if** (mar != lung_mem) *{<snip6>* mar = (mar+1) % 16; } **else** {**if**(x==2) *{ <snip7>* punti_retta = cont + 1; goto label_START;} **else** { goto label_INCREMENTA;}} *<snip8>*

The last column of Figure 6.3 demonstrates $SCLM_{GT\_O\_U}$ where the computational part of the IP core is made untimed. The labels relevant to communication with the simulation environment (i.e. label START) use a synchronization signal *sync*. The final recovered abstract SystemC code is shown in the last column of Table 6.1.

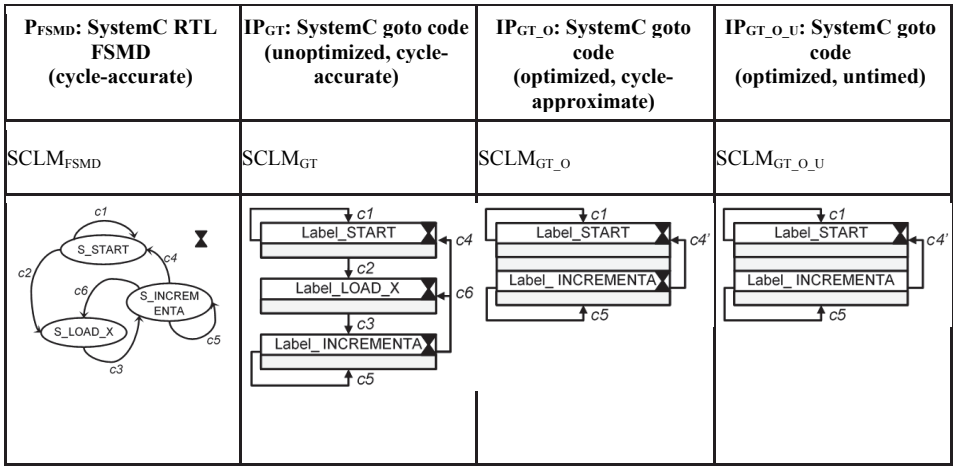


Figure 6.3 Abstraction-flow from FSMD to SCLM

6.5 Results

The methodology proposed in this chapter is implemented as a part of zamiaCAD and exercised on a set of RTL benchmark designs. IP design gcd is an implementation of the greatest common divisor algorithm, b02 and b07 are designs from the ICT'99 benchmarks family. The codec benchmark is a data dominated design which is a product of high level synthesis tool Synthagate (Synthezza, Online). It has 89 states and 900 lines of RTL FSMD VHDL code.

Each FSMD benchmark design (gcd, b02, b07 and codec) is implemented in SystemC and verified by its testbench. Different abstraction stages of these benchmarks are verified by simulating against the original testbench of the RTL FSMD. It is made sure that the abstractions do not break the testbench while optimizing the simulation speed.

Table 6.2 presents the experimental results for simulation speed-up provided by different stages of abstraction. Each of the four RTL benchmarks was abstracted to untimed SystemC implementation using SCLMs. As shown in the Table 6.2, the number of states as the key element remain the same when deriving the GT implementation from the FSMD. The GT_O implementation usually has lesser states due to the optimization of the key elements. The number of wait() statements are also optimized along with the number of key elements. A data dominated design has considerable optimizations of the key elements while abstracting from FSMD to GT_O. This is a result of lesser number of control states that are optimized by various SCLMs.

Table 6.2 Simulation Speed-Up by Different Stages of Abstraction

Design	Parameter	RTL FSMD	GT	GT_O	GT_O_U
Gcd	*speed-up (times)*	1	1.25	1.71	**2.65**
	# of key elements	3 states	3 labels	2 labels	2 labels
	# of wait statements	3	3	2	1
b02	*speed-up (times)*	1	1.03	1.03	**1.94**
	# of key elements	7 states	7 labels	5 labels	5 labels
	# of wait statements	7	7	5	1
b07	*speed-up (times)*	1	1.09	1.10	**12.08**
	# of key elements	8 states	8 labels	4 labels	4 labels
	# of wait statements	8	8	4	1
codec	*speed-up (times)*	1	1.35	1.42	**27.24**
	# of key elements	89 states	89 labels	59 labels	59 labels
	# of wait statements	89	89	59	1

Simulation time of SystemC RTL FSMD implementation is taken as the base for comparison, with the following observations:

- Unoptimized goto-code GT provides for a small speed-up in the range of 1.03 to 1.35 times, it still has the same number labels (key elements in goto-code) as number of states in the corresponding FSMDs.
- In the optimized goto-implementation GT_O the number of key elements (i.e. labels) was reduced in 1.4 to 2 times causing reduction of state transitions during computation in 2 to 3 times for the gcd, b02 and b07 designs and 10 times for the codec design. The number of wait statements was also decreased together with the optimized out states/labels making the simulation faster, but losing the time-accuracy of the model.
- Optimized untimed goto-implementation GT_O_U in the last column demonstrates speed-up achieved with regards to the initial SystemC cycle-accurate implementation. Here the data dominated designs with less interruption of the computational phase by communication such as b07 and codec demonstrate a significant speedup up to 27.24 times.

6.6 Application to Testbench

Table 6.1 demonstrates exploitation of the b07 IP core abstractions for three abstractions of a testbench. The initial RTL FSMD model IP_{FSMD} as well as its

goto-version IP_{GT} can be exploited directly with RTL TB (column 1) or with a TLM TB (column 2) using a Transactor (column 3). The abstracted cycle-approximate optimized IP_{GT_O} would need a different transactor considering the less number of wait statements required for computation. The untimed SystemC goto code $IP_{GT\_O\_U}$ is used directly with a TLM TB (column 4). Note the synchronization signal *sync* for the computation results synchronization between IP and its simulation environment.

Table 6.3 RTL and TLM Testbenches

RTL TB for $IP_{FSMD\,/\,GT}$	**TLM TB for $IP_{FSMD\,/\,GT}$**		**TLM TB for $IP_{GT\_O\_U}$**
	TB	**Transactor**	
SC_MODULE(sc_b07_rtl_tb) { sc_in<int> t_punti_retta; sc_out<bool> t_start, t_reset, t_clk; void sc_b07_tb_rtl::test() { <snip> t_reset.write(1); wait(clk.posedge_event( )); wait(clk.posedge_event( )); t_reset.write(0); wait(clk.posedge_event( )); t_start.write(1); wait(clk.posedge_event( )); while(t_punti_retta.rea d() ==0) wait(clk.posedge_event( )); t_start.write(0); wait(clk.posedge_event( )); }	#include "rtl2tlm_if.h" /* methods in this interface are implemented in the transactor */ SC_MODULE(sc_b07_rtl2tlm_ tb) { void sc_b07_tb_rtl2tlm::test() { <snip> do_reset(); set_start(); set_sync(); while(t_punti_retta.read( ) == 0) wait(clk.posedge_event()) ; clear_start(); }	#include "rtl2tlm_if.h" SC_MODULE(sc_b07_rtl2tlm_ TX) { sc_in<int> t_punti_retta; sc_out<bool> t_start, t_reset, t_clk; void sc_b07_TX::do_reset() { wait(clk.posedge_event()) ; t_reset.write(1); wait(clk.posedge_event()) ; t_reset.write(0); } void sc_b07_TX::set_start() { t_start.write(1); } void sc_b07_TX::clear_start() { t_start.write(0); } void sc_b07_TX::set_sync() { wait(clk.posedge_event()) ; t_sync.write(0); wait(clk.posedge_event()) ; t_sync.write(1);}	#include "b07_tlm_if.h" SC_MODULE(sc_b07_tlm_t b) { void sc_b07_tb_tlm::test() { <snip> do_reset(); /* untimed implementation */ set_start(); /* untimed implementation */ set_sync(); /* untimed implementation */ while(get_punti_retta( ) == 0) wait(SC_ZERO_TIME); /* wait for a delta-cycle to get the updated result */ clear_start(); }

6.7 Chapter Conclusions

This chapter introduces a novel approach (methodology and implementation) to automated abstraction of cycle-accurate FSMD cores into untimed SystemC models using SCLMs. SCLMs provide for an instrument to neglect design model parts irrelevant for particular manipulation step of the abstraction process, thus simplifying the abstraction flow, resulting in considerable simulation speed-up. The approach goes beyond the existing state-of-the-art in

eliminating the need for semantic analysis and elaboration of the cycle-accurate IP. This increases the overall scalability and provides a readable SystemC abstraction for RTL designs.

Experimental results show up to 27 times simulation speed-up in the tests conducted for the automatically generated abstract untimed SystemC models in comparison to the cycle accurate cores.

7 CONCLUSIONS

This thesis has proposed methodologies for *comprehensive* abstraction of RTL IP cores, focusing on interfaces as well as functionality of a design.

A novel methodology for the translation of VHDL RTL IP core (Chapter 3) provides rules and recommendations such that the cycle-accurate SystemC output is functionally correct, human-readable as well as with a clear correspondence to the source VHDL code. This enables further manual code changes and debug of the translated SystemC model.

Interesting insights were obtained by optimizing the SystemC models translated from VHDL RTL (Chapter 4). Co-simulating VHDL with SystemC model taking the highest simulation-time lowers the simulation-performance, whereas co-simulation with the SystemC model taking least simulation-time is 20% faster than VHDL-only simulation. Implementation of VHDL combinational-statements using multiple SystemC processes simulated about 25% faster than the implementation with a single SystemC process. SystemC synchronization strategies using single and multiple events take similar simulation-time, although single-event implementation is easier to develop and maintain.

A new methodology for abstracting the clock interface by minimization of delta-cycles and FSMD RTL transformation to an equivalent ASM representation was proposed (Chapter 5). Clock interface abstraction by minimization of delta-cycles resulted in 38% simulation speed improvement. Abstraction based on FSMD-to-ASM transformation resulted in 20% optimization in the simulation speed. Additional advantage of separating design functionality by ASM-states is that it enables functional abstraction of the model to higher, cycle-approximate levels.

Concept of SystemC-based Loose Models (SCLMs) is introduced (Chapter 6) for the first time for automated abstraction of cycle-accurate FSMD cores into untimed SystemC models. SCLMs provide for an instrument to neglect design model parts irrelevant for particular manipulation step of the abstraction process. Since semantic analysis and elaboration of the cycle-accurate IP is not required, the SCLM methodology has high scalability and provides a readable SystemC abstraction for RTL designs. Upton 27 times simulation speed-up was achieved by the untimed SystemC models generated using the SCLM approach.

Encouraging results, in terms of higher simulation speeds, are obtained for various methodologies proposed in this thesis. In author's knowledge, this research is unique in its attempt to comprehensively abstract an RTL IP-core.

Outcomes of this research, in terms of methodologies and results, are generic and applicable for abstraction of any VHDL RTL IP core translated to cycle-accurate SystemC model. The resulting methodologies are beneficial for simulation speed optimization goals, as well as easy design understanding by abstraction. As a whole, this research is extremely relevant for the SystemC

TLM-2.0 model providers, ESL/EDA-vendors and general SystemC community.

7.1 Future Work

This research paves the way for future work in multiple directions, such as:

- Synthesizable SystemC translated from VHDL
- Translation of Verilog to SystemC
- Native simulation of SystemC within zamiaCAD
- Co-simulation of VHDL/Verilog and SystemC within zamiaCAD
- Formal equivalence of VHDL translated to SystemC
- Formal equivalence of cycle-accurate and abstract SystemC models
- Abstraction of side-band signals like interrupts, reset, etc.
- Functional abstraction for control-dominated designs
- Optimization of SystemC kernel

A unique future work of this thesis is to explore the requirements and process for *filing a patent* on the abstraction of memory-mapped signal-level interface to TLM-2.0 standard.

REFERENCES

(Abba, Jeong-a, 2013) Abba, S.; Jeong-a Lee, "GSASRA: A Globally Self-Adaptive and Scalable Routing Algorithm for Network-on-Chips Architecture Using Particle Swarm Optimization," in Artificial Intelligence, Modelling and Simulation (AIMS), 2013 1st International Conference on, vol., no., pp.393-399, 3-5 Dec. 2013.

(Acosta et al., 2015) Acosta Hernandez, R.; Strum, M.; Wang Jiang Chau, "Transformations on the FSMD of the RTL code with combinational logic statements for equivalence checking of HLS," in Test Symposium (LATS), 2015 16th Latin-American, vol., no., pp.1-6, 25-27 March 2015.

(Ahuja et al., 2009) Ahuja, S.; Mathaikutty, D.A.; Lakshminarayana, A.; Shukla, S., "Accurate power estimation of hardware co-processors using system level simulation," in SOC Conference, 2009. SOCC 2009. IEEE International, vol., no., pp.399-402, 9-11 Sept. 2009.

(Alemzadeh et al., 2010) Alemzadeh, H.; Aminzadeh, S.; Saberi, R.; Navabi, Z., "Code optimization for enhancing SystemC simulation time," in Design & Test Symposium (EWDTS), 2010 East-West, vol., no., pp.431-434, 17-20 Sept. 2010.

(Arden, 2011) Arden, W., "More-than-Moore White Paper,", ITRS 2011.

(Bailey et al., 2007) Bailey, B., Grant Martin, Andrew Piziali, "ESL Design and Verification: A Prescription for Electronic System Level Methodology," Morgan Kaufmann Publishers, ISBN-13: 978-0123735515, 2007.

(Bailey et al., 2012) Bailey, B., Martin, G., "ESL Models and their Application: Electronic System Level Design and Verification in Practice," Springer 2012. ISBN 978-1-4419-0965-7.

(Balarin et al., 2007) Balarin, F., R. Passerone, "Specification, Synthesis, and Simulation of Transactor Processes," IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems, vol. 26, no. 10, pp. 1749-1762, Oct. 2007.

(Baranov, 2008) Baranov, S., "Logic and System Design of Digital Systems", TUT Press, Tallinn 2008.

(Benini et al., 2002) Benini, L., Bertozzi, D., Bruni, D., Drago, N., Fummi, F., Poncino, M. "Legacy SystemC co-simulation of multi-processor systems-on-chip". IEEE Int'l Conf Computer Design, 2002.

(Bhasker, 2004) Bhasker, J., "A SystemC Primer", Second Edition, Star Galaxy Publishing, 2004. ISBN 0-9650391-2-9.

(Bombieri et al., 2006) Bombieri, N.; Fummi, F.; Pravadelli, G., "A methodology for abstracting RTL designs into TL descriptions." 4th

ACM/IEEE International Conference on Formal Methods and Models for Co-Design, MEMOCODE '06, pp. 103-112, 2006.

(Bombieri et al., 2010) Bombieri, N., M. Ferrari, F. Fummi, et al., “HIFSuite: Tools for HDL Code Conversion and Manipulation,” in EURASIP Journal on Embedded Systems, vol. 1155, no. 10, 2010.

(Bombieri-a et al., 2010) Bombieri, N.; Fummi, F.; Pravadelli, G., “Abstraction of RTL IPs into embedded software,” in Design Automation Conference (DAC), 2010 47th ACM/IEEE, vol., no., pp.24-29, 13-18 June 2010.

(Bombieri et al, 2011) Bombieri, N.; Fummi, F.; Pravadelli, G., "Automatic Abstraction of RTL IPs into Equivalent TLM Descriptions," in IEEE Transactions on Computers, vol. 60, no. 12, pp. 1730-1743, Dec. 2011.

(Bouhadiba et al, 2013) Bouhadiba, T.; Moy, M.; Maraninchi, F.; Cornet, J.; Maillet-Contoz, L.; Materic, I., “Co-simulation of Functional SystemC TLM Models with Power/Thermal Solvers,” in Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International, vol., no., pp.2176-2181, 20-24 May 2013.

(Brandenburg, Stabernack, 2013). Brandenburg, J.; Stabernack, B., “Memory access analysis and optimization of a parallel H.264/SVC decoder for an embedded multi-core platform,” in Design and Architectures for Signal and Image Processing (DASIP), 2013 Conference on, vol., no., pp.304-311, 8-10 Oct. 2013.

(C11, 2011) ISO International Standard ISO/IEC 9899:2011 http://www.open-std.org/jtc1/sc22/wg14/www/standards.html#9899 , March 12, 2016.

(C++, 2014) ISO International Standard ISO/IEC 14882:2014(E) https://isocpp.org/std/the-standard/ , March 12, 2016.

(C-to-Si, Online) C-to-Si compiler, Cadence, http://www.cadence.com, March 12, 2016.

(Cadence, Online) Cadence Design Systems, http://www.cadence.com/products/sd/enterprise_manager/, March 12, 2016.

(Calazans et al., 2003) Calazans, N.; Moreno, E.; Hessel, F.; Rosa, V.; Moraes, F.; Carara, E., “From VHDL register transfer level to SystemC transaction level modeling: a comparative case study,” in Integrated Circuits and Systems Design, 2003. SBCCI 2003. Proceedings. 16th Symposium on, vol., no., pp.355-360, 8-11 Sept. 2003.

(Carbon, Online) Model Studio, Carbon Design Systems, http://carbondesignsystems.com/, March 12, 2016.

(Catapult, Online) Catapult C, Calypto, http://www.calypto.com/, March 12, 2016.

(Cheng et al., 1996) Cheng, K., Krishnakumar, A., “Automatic Generation of Functional Vectors Using the Extended Finite State Machine Model.” ACM Trans. on Design Automation of Electronic Systems, vol. 1, pp. 57–79, 1996.

(Chik et al., 2014) Chik, M.A., Rahim, A.B., Rejab, A.Z.M., Ibrahim, K., Hashim, U., “Discrete event simulation modeling for semiconductor fabrication operation,” in Semiconductor Electronics (ICSE), 2014 IEEE International Conference on, vol., no., pp.325-328, 27-29 Aug. 2014.

(Corrado, 1966) Corrado B., Jacopini, G., “Flow diagrams, turing machines and languages with only two formation rules.” Commun. ACM 9, 5 May 1966, 366-371.

(Cucchetto et al., 2014) Cucchetto, F., Lonardi, A., Pravadelli, G., “A common architecture for co-simulation of SystemC models in QEMU and OVP virtual platforms,” in Very Large Scale Integration (VLSI-SoC), 2014 22nd International Conference on, vol., no., pp.1-6, 6-8 Oct. 2014.

(Cynthesizer, Online) Cynthesizer, Forte, http://www.forteds.com/, March 12, 2016.

(Eclipse, Online) Eclipse, https://eclipse.org/, March 12, 2016.

(FreeHDL, Online) Naroska, E., “FreeHDL,” http://www.freehdl.seul.org/, March 12, 2016.

(Gajski et al., 1992) Gajski, D., Dutt, N., Allen, S., Wu, C., Lin, Y., “High-Level Synthesis: Introduction to Chip and System Design,” Kluwer Academic Publishers, first ed., 1992.

(Gajski, 2003) Gajski, D.D., “System design based on C/C++: what is needed and what is not,” in Information Technology Interfaces, 2003. ITI 2003. Proceedings of the 25th International Conference on, vol., no., pp.21-, 16-19 June 2003.

(Ghenassia, 2006) Ghenassia, F., “Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems,” Springer 2006. ISBN 0-387-26232-6.

(Glass et al., 2012) Glass, M., Teich, J., Liyuan Z., “A co-simulation approach for system-level analysis of embedded control systems,” in Embedded Computer Systems (SAMOS), 2012 International Conference on, vol., no., pp.355-362, 16-19 July 2012.

(Görgen et al., 2012) Görgen, R., Oetjens, J.H., Nebel, W. “Automatic integration of hardware descriptions into system-level models,” Proc. IEEE DDECS 2012, Tallinn, pp.105-110.

(Grötker et al., 2002) Grötker, T., Liao, S., Martin, G., Swan, S., “System Design with SystemC,” Springer, 2002. ISBN 1-4020-7072-1.

(HIFSuite, Online) HIFSuite, EDA-Lab, http://www.hifsuite.com/, March 12, 2016.

(IP-XACT, 2014) IEEE Draft Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows," in IEEE P1685/D4, February 2014, vol., no., pp.1-506, May 2 2014.

(ITC99, 1999) Politecnico di Torino, "ITC-99 Benchmarks,", http://www.cad.polito.it/tools/itc99.html , March 12, 2016.

(ITRS, 2012) International Technology Roadmap for Semiconductors. ITRS 2012 update. www.itrs2.net , March 12, 2016

(Jenihhin et al., 2014) Jenihhin, M., Tihhomirov, V., Syed, Saif Abrar, Raik, J., Bartsch, G., "zamiaCAD: Understand, Develop and Debug Hardware Designs," Design Automation for Understanding Hardware Designs DUHDE – Friday Workshop at DATE, 2014.

(Karfa et al., 2011) Karfa, C., Mandal, C., Sarkar, D., "Verification of Register Transfer Level Low Power Transformations," in VLSI (ISVLSI), 2011 IEEE Computer Society Annual Symposium on, vol., no., pp.313-314, 4-6 July 2011.

(Kirchner et al., 2010) Kirchner, T.; Bannow, N.; Kerstan, C.; Grimm, C., "Mixed signal simulation with SystemC and Saber," in Specification & Design Languages (FDL 2010), 2010 Forum on, vol., no., pp.1-6, 14-16 Sept. 2010.

(Kramer et al., 2013) Kramer, M.; Bader, S.; Oelmann, B., "Implementing Wireless Sensor Network applications using hierarchical finite state machines," in Networking, Sensing and Control (ICNSC), 2013 10th IEEE International Conference on, vol., no., pp.124-129, 10-12 April 2013.

(Leon3, Online) Aeroflex Gaisler,
http://www.gaisler.com/index.php/products/processors/leon3/, March 12, 2016

(Liao et al., 1997) Liao, S.Y., Steven, W.K., Gupta, Rajesh K., "An Efficient Implementation of Reactivity for Modeling Hardware in the Scenic Design Environment," DAC 1997.

(Mendoza et al, 2011) Mendoza, F., Kollner, C., Becker, J., Muller-Glaser, K. D., "An automated approach to SystemC/Simulink co-simulation," Rapid System Prototyping (RSP), 2011 22nd IEEE International Symposium on, vol., no., pp.135-141, 24-27 May 2011.

(Meyers, 2005) Meyers, S., "Effective C++," Addison-Wesley 2005, ISBN-13: 078-5342334876.

(Mihhailov et al, 2010) Mihhailov, D., Sklyarov, V., Skliarova, I., Sudnitson, A., "Hardware implementation of recursive algorithms," in Circuits and Systems (MWSCAS), 2010 53rd IEEE International Midwest Symposium on, vol., no., pp.225-228, 1-4 Aug. 2010.

(Ming et al., 2011) Ming-Chao, C., Tse-Chen, Yeh, Guo-Fu Tseng, “A QEMU and SystemC-Based Cycle-Accurate ISS for Performance Estimation on SoC Development,” in Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, vol.30, no.4, pp.593-606, April 2011.

(OStatic, Online) OStatic, “VHDLC,” http://ostatic.com/vhdlc/, March 12, 2016.

(Patel et al., 2007) Patel, H.D.; Shukla, S.K., “Tackling an Abstraction Gap: Co-simulating SystemC DE with Bluespec ESL,” in Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE '07, vol., no., pp.1-6, 16-20 April 2007.

(Plasma, Online) Plasma CPU a small synthesizable 32-bit RISC microprocessor, http://opencores.org/project,plasma/, March 12, 2016.

(Pong, 2006) Pong, P., “RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability,” Wiley-IEEE Press, March 12, 2006.

(Posadas et al., 2010) Posadas, H., Villar, E., Ragot, D., Martinez, M., “Early Modeling of Linux-Based RTOS Platforms in a SystemC Time-Approximate Co-simulation Environment,” in Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2010 13th IEEE International Symposium on, vol., no., pp.238-244, 5-6 May 2010.

(Roth et al., 2014) Roth, C., Reder, S., Bucher, H., Sander, O., Becker, J., “Adaptive Algorithm and Tool Flow for Accelerating SystemC on Many-Core Architectures,” in Digital System Design (DSD), 2014 17th Euromicro Conference on, vol., no., pp.137-145, 27-29 Aug. 2014.

(Samuel, 1976) Samuel C., “Digital circuits and logic design”, Prentice Hall PTR, 1976.

(Sanguinetti et al., 2012) Sanguinetti, J.; Meredith, M.; Dart, S., “Transaction-accurate interface scheduling in high-level synthesis,” in Electronic System Level Synthesis Conference (ESLsyn), vol., no., pp.31-36, 2-3 June 2012.

(SAVANT, Online) SAVANT: VHDL Analysis Tools, University of Cincinnati, http://www.ece.uc.edu/~paw/, https://github.com/wilseypa/savant, March 12, 2016.

(Shu-Hsuan et al., 2009) Shu-Hsuan Chou; Che-Neng Wen; Yan-Ling Liu; Tien-Fu Chen, “VeriC: A semi-hardware description language to bridge the gap between ESL design and RTL models,” in Quality of Electronic Design, 2009. ISQED 2009. Quality Electronic Design, vol., no., pp.535-540, 16-18 March 2009.

(Sklyarov, 1999) Sklyarov, V., “Hierarchical Finite-State Machines and Their Use for Digital Control”, IEEE Trans. on VLSI Systems, vol. 7, no 2, pp. 222-228, 1999.

(Smith et al, 1994) Smith, J.S., Wysk, R.A., Sturrock, D.T., Ramaswamy, S.E., Smith, G.D., Joshi, S.B., “Discrete event simulation for shop floor control,” in Simulation Conference Proceedings, 1994. Winter, vol., no., pp.962-969, 11-14 Dec. 1994.

(Stattelmann et al., 2011) Stattelmann, S., Bringmann, O., Rosenstiel, W., “Fast and accurate resource conflict simulation for performance analysis of multi-core systems,” in Design, Automation & Test in Europe Conference & Exhibition (DATE), vol., no., pp.1-6, 14-18 March 2011.

(Sumaryo et al, 2013) Sumaryo, S., Halim, A., Ramli, K., “Improved discrete event simulation model of traffic light control on a single intersection,” in QiR (Quality in Research), 2013 International Conference on, vol., no., pp.116-120, 25-28 June 2013.

(SV, 2009) IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language," in IEEE STD 1800-2009 , vol., no., pp.1-1285, Dec. 11 2009. doi: 10.1109/IEEESTD.2009.5354441. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5354441&isnumber=5354440 , March 12, 2016.

(Syed-a et al., 2013) Syed, Saif Abrar, Jenihhin, M., Raik, J., “Extensible Open-Source Framework for Translating RTL VHDL IP Cores to SystemC,” IEEE Int'l Symp. Design and Diagnostics of Electronic Circuits & Systems (DDECS), 2013.

(Syed-b et al., 2013) Syed, Saif Abrar, Kiran, S., Jenihhin, M., Raik, J., Babu, C., “Performance Analysis of Cosimulating Processor Core in VHDL and SystemC,” IEEE Int'l Conf. Advances in Computing, Communications and Informatics (ICACCI), 2013.

(Synopsys, Online) Synopsys, Inc., http://www.synopsys.com/Prototyping/VirtualPrototyping/DigitalSignalProcessing/Pages/system-studio.aspx, March 12, 2016.

(Synthezza, Online) Baranov, S., Synthagate, Synthezza Corporation, http://synthezza.com/, March 12, 2016.

(SystemC, 2011) IEEE 1666 Standard SystemC Language Reference Manual, 2011. doi:10.1109/IEEESTD.2012.6134619. ISBN 978-0-7381-6801-2. http://standards.ieee.org/getieee/1666/ , March 12, 2016.

(Takach, 1995) Takach, A., Wolf, W., “An automaton model for scheduling constraints in synchronous machines,” IEEE Transactions on Computers, vol.44, no.1, pp. 1-12, Jan 1995.

(Tasker et al., 2006) Tasker, S.; Nikhil, R.S., “Beyond RTL: advanced digital system design,” in VLSI Design, 2006. Held jointly with 5th International Conference on Embedded Systems and Design., 19th International Conference on, vol., no., pp.2 pp.-, 3-7 Jan. 2006.

(Trinh et al, 2011) Trinh, H., Fan, Q., Jiyan Pan; Gabbur, P.; Miyazawa, S.; Pankanti, S., “Detecting human activities in retail surveillance using hierarchical finite state machine,” in Acoustics, Speech and Signal Processing (ICASSP), IEEE International Conference on, vol., no., pp.1337-1340, 22-27 May 2011.

(Tse-Chen et al., 2010) Tse-Chen, Y., Zin-Yuan, L., Ming-Chao Chiang, “Optimizing the Simulation Speed of QEMU and SystemC-Based Virtual Platform,” in Information Engineering and Computer Science (ICIECS), 2010 2nd International Conference on, vol., no., pp.1-4, 25-26 Dec. 2010.

(Tšepurov, 2012) Tšepurov, A., Bartsch, G., Dorsch, R., Jenihhin, M., Raik, J., Tihhomirov, V., “A Scalable Model Based RTL Framework zamiaCAD for Static Analysis,” Proc. IEEE VLSI-SOC, pp. 1-6, 2012.

(V2SC-EKUT, Online) VHDL-to-SC Converter, Eberhard-Karls-University of Tübingen, http://www-ti.informatik.uni-tuebingen.de/~systemc/, March 12, 2016.

(VAUL, Online) VHDL Analyzer and Utility Library (VAUL) http://www.freehdl.seul.org/, March 12, 2016.

(Verilator, Online) Snyder, W., Verilator, Veripool, http://www.veripool.org/wiki/verilator/ , March 12, 2016.

(Verilog, 2005) IEEE Standard for Verilog Hardware Description Language," in IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001) , vol., no., pp.0_1-560, 2006. doi:10.1109/IEEESTD.2006.99495. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1620780&isnumber=33945 , March 12, 2016.

(VH2SC, Online) VH2SC, HT-Lab, http://www.ht-lab.com/, March 12, 2016.

(VHDL, 2008) IEEE Standard VHDL Language Reference Manual," in IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002) , vol., no., pp.c1-626, Jan. 26 2009. doi: 10.1109/IEEESTD.2009.4772740. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4772740&isnumber=4772739 , March 12, 2016.

(VHDL2SC, 2012) Görgen, R., “VHDL-to-SystemC Transformation,” http://vhome.offis.de/ralphg/vhdl2sc.pdf , March 12, 2016

(Vitor et al, 2014) Vitor, A., Almeida, A., Miranda, Ciro T., de Barros Vidal, Flavio, “A parallel hierarchical finite state machine approach to UAV control for search and rescue tasks,” in Informatics in Control, Automation and Robotics (ICINCO), 2014 11th International Conference on, vol.01, no., pp.410-415, 1-3 Sept. 2014.

(Wong et al., 2012) Wong, M., Yong, C., Chaw, L., “Enhancing Discrete Event Simulation in multi-core processors environments using Multi-Dimensional Queue Mechanism,” in Information Science and Service Science and Data

Mining (ISSDM), 2012 6th International Conference on New Trends in, vol., no., pp.297-302, 23-25 Oct. 2012.

(Xuexiang et al, 2012) Xuexiang W., Shan, W., Hao Liu, “Uniform SystemC Co-Simulation Methodology for System-on-Chip Designs,” in Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2012 International Conference on, vol., no., pp.261-267, 10-12 Oct. 2012.

(Zaidi et al., 2011) Zaidi, Y., Grimm, C., Haase, J., “Simulation based tuning of system specification,” in Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011, vol., no., pp.1-6, 14-18 March 2011.

(zamiacad, Online) zamiaCAD Open-source Hardware Design Framework, http://zamiacad.sf.net/, March 12, 2016.

(Zhao, 2002) Zhao, S.; Gajski, D., “Modeling a new RTL semantics in C++,” in Circuits and Systems, 2002. ISCAS 2002. IEEE International Symposium on, vol.5, no., pp. V-741-V-744 vol.5, 2002.

ABSTRACT

We are witnessing an era in which consumer-electronic devices are supporting ever increasing set of features in shortening form factors. International Technology Roadmap for Semiconductors (ITRS) identifies this dual trend as: miniaturization of the digital functions ("More Moore") and functional diversification ("More-than-Moore"). Traditional design approaches at Register Transfer Level (RTL) are no longer suitable, resulting in design methodologies at higher levels of abstraction. The most recent approach in this direction is the Electronic System Level (ESL) design methodology, made practical with the introduction of Transaction Level Modeling (TLM). In order to enable the ESL and TLM methodologies, amongst various languages and tools, SystemC has emerged as the dominant language standardized as IEEE Standard 1666-2011.

Growing focus on ESL and TLM has put IP design houses under pressure to provide the SystemC models of their legacy VHDL or Verilog IP cores to remain competitive in today's market. Manual abstraction of RTL is error-prone and requires enormous time and effort. The only preferred way is to automate the VHDL/Verilog translation and abstraction to SystemC. This guarantees consistent models at various abstraction levels, in minimal time and effort.

Aim of this research is to develop a methodology to automatically abstract RTL IP cores into higher-level models, with zero or minimal manual-interaction. The resulting SystemC model can be generated for both cycle-accurate as well as untimed PV abstractions. Results of this research are relevant for the SystemC model providers, ESL/EDA tool-vendors and to the wider SystemC community.

The thesis starts with the translation of VHDL RTL to SystemC. Various rules are proposed to automatically generate SystemC from input VHDL RTL. These rules are generic in nature and can be applied to any VHDL RTL design.

Once VHDL RTL has been translated to SystemC, the focus of the thesis is on code-level optimization of translated SystemC. Various optimizations techniques are introduced for the VHDL to SystemC translated code. These optimizations allow higher simulation speeds as compared to the directly translated SystemC-models.

In practice, co-simulation of VHDL-SystemC is often employed while moving from complete system simulation in RTL towards SystemC implementation. Different strategies are discussed for selecting the modules to replace from VHDL to SystemC so that faster co-simulation speed is realized.

After the optimizations discussed in the previous paragraphs, the focus is on abstraction of input/output interfaces. It deals with the conversion of signal-level and cycle-accurate protocol to the standard TLM-2.0 protocol in SystemC. The introduced methodology can be used to abstract any arbitrary signal-level

protocol to TLM-2.0. Only side-band signals like clock and interrupt are not converted to TLM-2.0 protocol.

Clock signal abstraction is the next focus of the thesis. Clock abstraction aims at removing the clock signal, either partially or completely, preserving the behavior of the system design. Minimization of delta-cycles and events in a system level model is the core idea behind the clock signal abstraction. Another explored approach is to transform VHDL Finite State Machine with Datapath (FSMD) design to an equivalent Algorithmic State Machine (ASM) representation in SystemC that enables event-based triggering of ASM states.

Finally, the thesis explores a new idea for functionality abstraction. Its focus is on abstracting the behavioral implementation of a module. Notion of SystemC-based Loose Modeling (SCLM) is introduced to functionally abstract a design implemented as VHDL FSMD. SCLM provides for an instrument to neglect design model parts irrelevant for particular manipulation step of the abstraction process, thus simplifying the abstraction flow.

Various approaches proposed in this thesis are experimented upon, with encouraging results in terms of faster simulation speed. In author's knowledge, this research is unique in its attempt to *comprehensively* abstract various interfaces and functionality of RTL IP cores.

KOKKUVÕTE

Elame ajastul, mil aina vähenevate mõõtmetega laiatarbe elektroonikaseadmed pakuvad üha laiemaid võimalusi. Rahvusvaheline Pooljuhttehnollogiate Teeviit ITRS kirjeldab seda trendi kui digitaalfunktsioonide miniaturiseerimist ("rohkem Moore'i seadust") ja funktsionaalset mitmekesistumis ("rohkem kui Moore'i seadus"). Traditsioonilised register-siirde tasemel toimivad projekteerimislähenemised ei ole enam piisavad mistõttu on tekkinud uued projekteerimis-metodoloogiad kõrgematel abstraktsioonitasemetel. Uusim lähenemine selles suunas on elektroonilise süsteemitaseme (ingl. k. Electronic System Level ehk ESL) metodoloogia, mis tekkis koos transaktsioonitaseme modelleerimise kasutuselevõtmisega. ESL metodoloogia võimaldamisel on domineerivaks osutunud SystemC keel (IEEE Standard 1666-2011).

ESL metodoloogia kasvav populaarsus on tekitanud tuumade projekteerijatele, selleks et konkurentsis püsida, vajaduse pakkuda oma olemasolevate VHDL või Verilog tuumade jaoks SystemC mudeleid. Tuumade register-siirde taseme abstraheerimine käsitsion vaeva nõudev ning võib põhjustada vigu. Seega oleks eelistatav VHDL/Verilog tuumade automaatne tõlkimine ja abstraheerimine SystemC keeled. Sellega garanteeritaks korrektsed mudelid erinevatel abstraktsioonitasemetel, minimaalse töökuluga.

Käesoleva uurimistöö eesmärgiks on metodoloogia väljatöötamine automaatseks, minimaalset käsitööd nõudvaks register-siirde taseme tuumade automaatseks abstraheerimiseks süsteemitaseme mudeliteks. Seejuures vaadeldakse töös nii täpse ajastusega (cycle accurate) kui ka abstraktseid (programmers view) SystemC mudeleid. Uurimistöö tulemused on olulised nii SystemC mudelite projekteerijatele, raalprojekteerimistarkvara väljatöötajatele kui ka laiemalt SystemC keele arendamisega töötavale kogukonnale.

Töös pakutakse välja reeglid SystemC mudelite genereerimiseks VHDL kirjeldusest. Reeglid on üldised ning neid saab rakendada mistahes VHDL disainile. Lisaks tegeleb töö erinevate genereeritud SystemC koodi optimeerimistehnikatega. Nimetatud tehnikad võimaldavad tõsta simuleerimiskiirust võrreldes vahetult tõlgitud SystemC mudelitega.

Praktikas rakendatakse üleminekul register-siirde tasemelt süsteemitasemele VHDL-SystemC koossimuleerimist. Käesolevas töös vaadeldakse erinevaid strateegiaid VHDL-moodulite asendamiseks SystemC mudelitega, et maksimeerida koossimuleerimise kiirust.

Lisaks transleerimisele, käsitleb töö sisend/väljund-liideste abstraheerimist. Vaadeldakse madala taseme protokollide teisendamist SystemC keeles esitatud TLM-2.0 standardi protokollideks . Välja pakutud metodoloogia on rakendatav

suvalise signaalitaseme protokolli abstraheerimiseks TLM-2.0 tasemele. Abstraheerimine ei käsitle vaid kõrvalsignaale nagu taktsignaal, katkestused jne.

Seetõttu käsitleb töö eraldi taktsignaali abstraheerimist, mille eesmärgiks on taktsignaali osaline või täielik eemaldamine säilitades seejuures süsteemi funktsionaalsuse. Peamiseks ideeks on simuleerimise deltatsüklite ja sündmuste minimiseerimine süsteemitaseme mudelis. Lisaks vaadeldakse VHDL automaatide teisendamist ekvivalentseks algoritmilisteks automaatideks (AA), mis on esitatud SystemC keeles ning võimaldavad sündmustepõhist üleminekut automaadi olekute vahel.

Töö esitab uudse idee mooduli funktsionaalsuse abstraheerimiseks, mis tugineb SystemC-põhisele "lõdvale" modelleerimisele (SCLM). SCLM on instrument, mis võimaldab ignoreerida mudeli osi, mis antud teisenduse kontekstis tähtsust ei oma ning seetõttu lihtsustada abstraheerimisprotsessi.

Töös esitatud eksperimendid näitavad, kuidas välja töötatud lähenemised aitavad simuleerimise kiirust tõsta. Doktoritöö uudsus seisneb komplekssе lahenduse pakkumises erinevate liideste ja funktsionaalsuse abstraheerimiseks register-siirde taseme disainide jaoks.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to everybody who has either directly or indirectly contributed to my PhD studies and this thesis.

In particular, I would like to thank my supervisors Dr. Maksim Jenihhin and Prof. Jaan Raik, my mentors and friends throughout the duration of PhD. They made my PhD journey much more pleasurable than I expected!

I would like to acknowledge everyone from Tallinn University of Technology who contributed to my PhD work with discussions and ideas. Special mention must be made of all the faculty members who encouraged me to take their courses and provided all the necessary help and support. Also, I would like to thank Guenter Bartsch, the main developer and project founder of zamiaCAD.

I would specially thank Dr. Margus Kruus, director of the Dept. of Computer Engineering, for his support with many administrative issues, especially with my remote PhD facilitation.

I would like to acknowledge several organizations that have supported my PhD studies, including the research presented in this thesis. They are Tallinn University of Technology, National Graduate School in Information and Communication Technologies (IKTDK), Estonian Information Technology Foundation (EITSA), Information Technology Foundation for Education (HITSA), EU's FP7 collaborative research project DIAMOND, European Regional Development Fund through the Centre for Integrated Electronic Systems and Biomedical Engineering (CEBE).

Finally, I would like to appreciate my family and friends who have supported me throughout my studies. They provided the much needed strength and encouragement during this long and memorable journey.

Thank you all!

Syed, Saif Abrar

Bangalore, February 2016

Appendix A

Research paper [A]

Syed, Saif Abrar; Jenihhin, Maksim; Raik, Jaan. “FSMD RTL Design Manipulation for Clock Interface Abstraction.” IEEE International Conference on Advances in Computing, Communications and Informatics (ICACCI), Kochi, India, 2015. pp 1-6.

FSMD RTL Design Manipulation for Clock Interface Abstraction

Syed Saif Abrar, Maksim Jenihhin, Jaan Raik
Tallinn University of Technology, Estonia
syedsaifabrar@gmail.com, [maksim, jaan]@ati.ttu.ee

ABSTRACT
The rapid raise of embedded systems design complexity and size has emphasized the importance of high-performance simulation models. This has resulted in emergence of design methodologies at the higher levels of abstraction such as Electronic System Level (ESL) and Transaction Level Modelling (TLM) and SystemC language as the main instrument. In practice, system architects and system integrators often have access to a library of legacy Register Transfer Level (RTL) HW IP cores or obtain new ones from IP design houses. To address simulation performance, such RTL IP cores are manually recreated at more abstract levels, which implies significant tedious and error-prone effort. The current paper addresses this problem by proposing a novel approach for automated FSMD RTL design manipulation for clock interface abstraction. The manipulation approach takes as an input FSMD (Finite State Machine with datapath embedded) RTL design in VHDL and transforms it to an equivalent Algorithmic State Machine (ASM) representation in SystemC with explicit separation of design functionality by states. Finally, the clock interface is abstracted up to optimize the simulation performance. The manipulation details are demonstrated on a case study design and the first experimental results show simulation speed-up and prove feasibility of the proposed approach.

Keywords
Register Transfer Level - RTL; Algorithmic State Machine - ASM; Finite State Machine – FSM; SystemC; abstraction; simulation; modelling.

1. INTRODUCTION
Embedded systems are becoming increasingly complex and large, posing hardware and software design challenges. The time-to-market constraints push design vendors to optimize the design flow. The traditional design approaches at Register-Transfer Level (RTL) using VHDL or Verilog are no longer suitable. This has resulted in emergence of design methodologies at the higher levels of abstraction such as Electronic System Level (ESL) and Transaction Level Modelling (TLM) with SystemC language as the main instrument.

Electronic System Level (ESL) [1] design is an emerging methodology to model the embedded systems at a high level of abstraction. Transaction Level Modeling (TLM) is the widely recognized and adopted approach to realize the ESL in practice. TLM models are applicable for a range of design tasks, e.g. early software development, performance analysis, architecture exploration, HW/SW partitioning, etc. In practice, system architects and system integrators often have access to a library of legacy Register Transfer Level (RTL) HW IP cores or obtain new ones from IP design houses. To address simulation performance, such RTL IP cores are manually recreated at more abstract levels, which implies significant tedious and error-prone effort.

This paper contributes to automated abstraction of RTL IP cores to SystemC models by proposing a novel approach to automated FSMD RTL design manipulation for clock interface abstraction. The manipulation approach takes FSMD (Finite State Machine with datapath embedded) RTL design in VHDL as an input and transforms it to an equivalent Algorithmic State Machine (ASM) representation in SystemC with explicit separation of design functionality by states. Finally, the clock interface is abstracted to optimize the simulation performance. The manipulation details are demonstrated on a case study design and the first experimental results show simulation speed-up and prove feasibility of the proposed approach. As shown in the next Section, there have been several works on translating as well as abstracting RTL models to SyctemC. However, this paper goes beyond the existing state-of-the-art in the following aspects:

1. It develops a *readable SystemC abstraction* for FSMD based RTL designs.
2. It develops fully automated *clock interface abstraction* for RTL FSMD.

The core advantages are the automated integration of RTL descriptions based on the FSMD formalism into ESL system descriptions, the increase in simulation speed and the ease of debugging with respect to other formal abstraction methods. An additional advantage of separating design functionality by states is that it enables functional abstraction of the model to higher, cycle-inaccurate levels.

The rest of the paper is organized as follows. Section 2 provides an overview of the related work. Section 3 presents the background by defining the Extended Finite State Machine (EFSM) and Algorithmic State Machine (ASM) chart models. In Section 4, the transformation methodology is defined. Section 5 presents a case study. Finallt, Experimental results and conclusions are provided.

2. RELATED WORK
Cosimulation of RTL descriptions with SystemC has been of interest since long. ISS cosimulation is discussed first in [2]. Simulink cosimulation is discussed in [3]. Cycle-accurate and TLM cosimulation is described in [4].

Concerning facilitating conversion of RTL designs into SystemC, there have been a number of tools. Among the commercial solutions there is Carbon Model Studio [5] that allows creating configurable SystemC models from RTL VHDL or Verilog descriptions. It is targeted mainly at simulation speedup and does not intend to create human-readable output. HIFSuite [6] is a design and verification framework addressing manipulation and integration of heterogeneous design parts. It allows translation of RTL VHDL descriptions into RTL as well as

TLM SystemC. The output result also does not consider human-readability and correspondence to the source VHDL. The approach supports equivalence checking to prove the correctness of the result.

Non-commercial and free solutions to generate SystemC from VHDL are VHDLParser by University of Tuebingen [7] and VH2SC by HT-Lab [8]. Both approaches consider mapping of the source VHDL to a limited set of SystemC constructs. These tools have demonstrated significant limitations in terms of the supported language subset, do not guarantee equivalence and they are not maintained anymore. The most relevant approach is published by OFFIS [9]. It assumes creation of readable SystemC representations from VHDL that are targeted to be wrapped and simulated in the Simulink environment. There are known approaches for creating SystemC models from Verilog [10, 11] and tools targeting creation of other C++ subsets. Finally, [12] discusses a methodology for abstracting RTL descriptions to different TLM levels.

In the authors' previous work a tool for readable translation of RTL VHDL descriptions to SystemC was developed [13]. In [14], the concept of abstracting clock interfaces was introduced. In the current paper we propose automated formal transformations for FSMD RTL design manipulation meaning significantly wider application possibilities for the clock abstraction concept.

3. BACKGROUND

3.1 Extended Finite State Machine (EFSM)

In this paper we assume Register-Transfer Level (RTL) design descriptions in an HDL that implement FSMD (Finite State Machine with datapath embedded) [2]. Such RTL descriptions may be represented by EFSM [16] models.

The EFSM model allows a more compact representation of the state space than traditional FSM, thus, the risk of state explosion that incurs to model a large design by using FSMs is sensibly reduced. A simple example of EFSM is reported in Figure 1.

Definition 1 *An EFSM is defined as a 5-tuple* $M = <S, I, O, D, T>$, *where:* S *is a set of states,* I *is a set of input symbols,* O *is a set of output symbols,* D *is an n-dimensional linear space* $D_1 \times \ldots \times D_n$, T *is a transition relation such that* $T : S \times D \times I \rightarrow S \times D \times O$. *A generic point in* D *is described by an* n*-tuple* $x = (x_1, ..., x_n)$*; it models the values of the registers of the DUV. A pair* $<s, x> \in S \times D$ *is called configuration of* M.

An operation on M is defined in this way: if M is in a configuration $<s, x>$ and it receives an input $i \in I$, it moves to the configuration $<t, y>$ iff $((s, x, i), (t, y, o)) \in T$ for $o \in O$.

The EFSM differs from the classical FSM, since each transition does not present only a label in the classical form $(i)/(o)$, but it takes care of the register values too. Transitions are labeled with an *enabling* function e and an *update* function u. An update function $u(x, i)$ can be applied to a configuration $<s_1, x>$ if there is a transaction $t : s_1 \rightarrow s_2$, labeled e/u, such that $e(x, i) = 1$. In this case we say that t can be *fired* by applying the input i. Figure 3 presents an example of a graphical representation for an EFSM.

In this work, all EFSMs are considered to be deterministic. Generation of EFSM models for RTL designs is not discussed in this paper and assumed to be performed by existing approaches e.g. VAUL [17].

3.2 Algorithmic State Machine Chart

There exists a number of flowchart-like models for functional design representation, e.g. flowchart [18], ASM [19], HGS [20] BFSM [21]. In this paper, we rely on the formalism defined in [22] as *Algorithmic State Machine chart* (ASM). One of its advantages is a clear representation of FSM states by blocks. ASMs have three basic elements state box (denoted by rectangles), decision box (denoted by diamonds) and conditional box (denoted by rounded rectangles) as illustrated in Figure 1. Different from [22] to support EFSMs we allow register operations in both state and conditional boxes. *State boxes* are used to indicate states in EFSM, register operations and Moore-type output signals generated in the state. *Decision box* reflects the condition to be tested where exit paths represent true and false evaluations. In *conditional boxes* inputs come from one of exit paths of decision boxes and they contain register operation or Mealy-type output signals dependent on the inputs generated during the state. *ASM block* is a structure consisting of one state box, all decision and conditional boxes associated with its exit paths. Such block has one entrance and any number of exits paths. Each ASM block in is dedicated to state of EFSM and is traversed within one clock cycle.

ASMs provide for explicit separation of the RTL FSMD functionality performed by one state of EFSM.

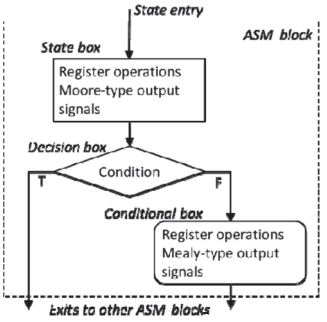


Figure 1. A state block in the ASM chart [22]

4. TRANSFORMATION METHODOLOGY

This section describes the methodology for transformation of an EFSM representation of VHDL FSMD RTL design to an equivalent ASM with further implementation of high-performance SystemC model of this ASM.

A. Transforming EFSM to ASM

The procedure of transformation of an EFSM to an equivalent ASM relies on the rules defined bellow. Here we use the EFSM and ASM terminology as specified in Section 3.

- Let B be a set of all blocks b_i in the ASM.
- Each EFSM state $s_i \in S$ is represented by an ASM block $b_i \in B$.
- Transitions entering an ASM block b_i correspond to the ones entering the corresponding EFSM state s_i.
- Similarly, transitions exiting an ASM block b_i correspond to the ones exiting the corresponding EFSM state s_i., i.e. lead to the ASM blocks corresponding to the next states of the given state in EFSM.
- Comparison operations at the *Decision boxes* within an ASM block b_i are derived from the enabling functions e at the transition arcs exiting from an EFSM state s_i.
- In the case of a Mealy machine, behavior specified by the *condition box* of an ASM block b_i is equivalent to that of the update function u of the corresponding transition exiting EFSM state s_i.
- In the case of a Moore machine, behavior specified by the *state box* of an ASM block b_i is equivalent to that of the update function u of the corresponding EFSM state s_i.

Figure 2 shows a sample EFSM, as a hybrid of Mealy and Moore machines, and its conversion to an ASM chart. FSM states *s0* and *s1* are transformed to ASM blocks *s0* and *s1*. Input *i1* is used to transition from *s0* to *s1*. The decision box within ASM block *b0* checks the value of *i1* in order to either remain in *s0* or to move to state *s1*. While going to *s1*, output *o2* is also asserted, both in EFSM and in ASM.

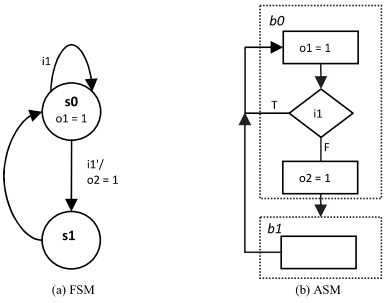


Figure 2. Sample conversion from FSM to ASM

B. SystemC models of ASMs

This section defines the methodology for an efficient SystemC model of the ASM chart. The methodology is generic and can to be applied to FSMD design represented by ASM chart according to the rules in Section 4.A. The discussed methodology starts with an ASM chart and finally produces a simulation-speed optimized SystemC model of it.

Steps in the proposed methodology are the following:

1. For each ASM-block b, define a method m. Each method m is implemented in C language
2. Define an sc-event v for each C method m
3. Declare each C method m as a SC_METHOD, and make it sensitive to corresponding sc-event v
4. All SC_METHODs are declared as dont_initialize(), except the SC_METHOD that corresponds to the FSM's initial-state. This ensures that the simulation starts with the SC_METHOD that models the EFSM initial-state.
5. Wherever there is a transition from one ASM block to another ASM block, trigger the sc-event v after the system-wide clock period.
6. Due to the above rule, no SC_METHOD is made sensitive to the clock input signal.

The above methodology results in an efficient SystemC implementation of an ASM chart due to the following optimizations:

1. Usage of SC-METHOD that is faster in simulation than SC-THREAD, due to lower overhead of context switching.
2. Triggering of SC_METHOD using static global clock period, rather than dynamically waiting on clock edge. This lowers burden on the simulation kernel.

5. CASE STUDY

This section applies the above discussed methodology to transform a VHDL FSMD RTL implementation of a Greatest Common Divisor (GCD) design to the corresponding ASM chart and further to its SystemC implementation optimized for high performance.

Figure 3 represents an EFSM for the VHDL description of GCD shown in Figure 5. As a result of application of part A of the methodology the EFSM is transformed to ASM-chart as shown in Figure 4.

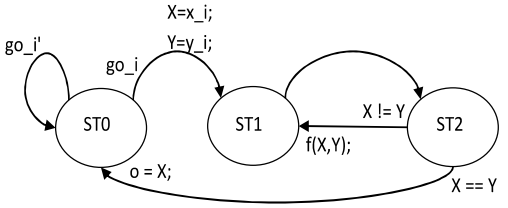


Figure 3. EFSM for GCD

Next, part-B of the methodology is applied to the GCD ASM as follows:

1. Set $B = \{ ST0, ST1, ST2\}$
2. Methods for each ASM block: $M = \{m_{ST0}, m_{ST1}, m_{ST2}\}$

e.g.

```
void m_ST0();
void m_ST1();
void m_ST2();
```

3. SystemC event for each method m: $V = \{v_{m\_ST0}, v_{m\_ST1}, v_{m\_ST2}\}$

e.g.

```
sc_event ev_m_ST0, ev_m_ST1, ev_m_ST2;
```

4. Declare each method {m_{ST0}, m_{ST1}, m_{ST2}} as an SC_METHOD, and make it sensitive to corresponding sc-evenst {v_{m_ST0}, v_{m_ST1}, v_{m_ST2}}
5. All SC_METHODs are declared as `dont_initialize()`, except the SC_METHOD that corresponds to the EFSM's initial state.

e.g.

```
SC_METHOD(m_ST0);
    sensitive << ev_m_ST0;
SC_METHOD(m_ST1);
    sensitive << ev_m_ST1;
    dont_initialize();
SC_METHOD(m_ST2);
    sensitive << ev_m_ST2;
    dont_initialize();
```

6. Trigger the sc-events {v_{m_ST0}, v_{m_ST1}, v_{m_ST2}} after the system-wide clock-period.

e.g.

```
ev_m_ST0.notify(SC_CLK_PERIOD, SC_NS);
```

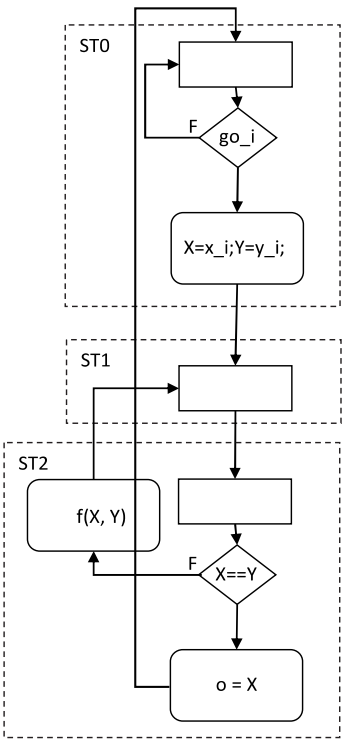


Figure 4. ASM chart for GCD

A complete GCD implementation both in RTL FSMD and SystemC ASM are shown in the Figure 5.

6. EXPERIMENTAL RESULTS

In this section, we present the simulation performance comparison for the GCD design's initial RTL FSMD implementation compared against the abstracted ASM-based functional model. Both models were described in SystemC and simulated with the same test bench in the Microsoft VC 2010 Express environment.

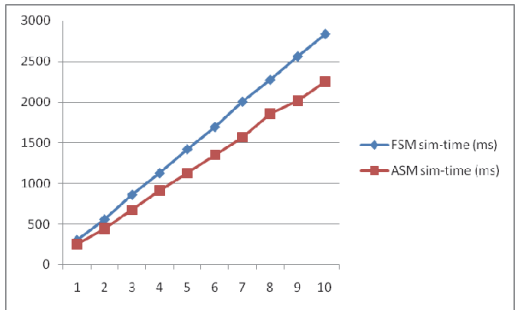


Figure 6. Simulation results for GCD in represented by RTL FSMD and the resulted ASM-based functional model.

Figure 6 demonstrates the CPU time required for simulation of the both models for 1 to 10 million SystemC clock cycles on a Intel i5 @2.6GHz, 8GB-RAM PC. The simulation speed-up achieved is in the range of 16.4 to 21.7%.

7. CONCLUSIONS

The paper presented a novel approach to automated FSMD RTL design manipulation for clock interface abstraction. The manipulation approach takes FSMD RTL design in VHDL as an input and transforms it to an equivalent Algorithmic State Machine (ASM) representation in SystemC with explicit separation of design functionality by states. Finally, the clock interface is abstracted to optimize the simulation performance.

The manipulation details were demonstrated on a case study design and the first experimental results show simulation speed-up between 16.4 to 21.7%.

An additional advantage of separating design functionality by states is that it enables functional abstraction of the model to higher, cycle-inaccurate levels, which will be the future research direction.

VHDL RTL FSMD implementation of GCD

```vhdl
entity gcd is

port(     clk:      in std_logic;
          rst:      in std_logic;
          go_i:     in std_logic;
          x_i:      in unsigned(3 downto 0);
          y_i:      in unsigned(3 downto 0);
          d_o:      out unsigned(3 downto 0)
);
end gcd;
architecture FSMD of gcd is
begin
  process(rst, clk)
    type S_Type is (ST0, ST1, ST2);
    variable State: S_Type := ST0 ;
    variable Data_X, Data_Y: unsigned(3 downto 0);
    begin
          if (rst='1') then               -- initialization
              d_o <= "0000";
              State := ST0;
          elsif (clk'event and clk='1') then
              case State is
                when ST0 =>               -- starting
                        if (go_i='1') then
                            Data_X := x_i;
                            Data_Y := y_i;
                            State := ST1;
                        else
                            State := ST0;
                        end if;
                    when ST1 =>       -- idle state
                        State := ST2;
                    when ST2 =>       -- computation
                        if (Data_X/=Data_Y) then
                            if (Data_X<Data_Y) then
                                Data_Y := Data_Y - Data_X;
                            else
                                Data_X := Data_X - Data_Y;
                            end if;
                          State := ST1;
                        else
                            d_o <=Data_X;        -- done
                            State := ST0;
                        end if;
                    when others =>        -- go back
                        d_o <= "ZZZZ";
                        State := ST0;
             end case;
          end if;
```

SystemC ASM implementation of GCD

```systemc
SC_MODULE(gcd) {
sc_in<bool >  clk; sc_in<bool >  rst; sc_in<bool >  go_i;
sc_in<sc_uint<4> >  x_i;
sc_in<sc_uint<4> >  y_i; sc_out<sc_uint<4> > d_o;

sc_event ev_m_ST0, ev_m_ST1, ev_m_ST2;
void m_ST();
void m_ST1();
void m_ST2();

gcd(sc_module_name _n) : sc_module(_n) {
  SC_METHOD(m_ST0);
    sensitive << ev_m_ST0;
  SC_METHOD(m_ST1);
    sensitive << ev_m_ST1;
    dont_initialize();
  SC_METHOD(m_ST2);
    sensitive << ev_m_ST2;
    dont_initialize();
} //CTOR
~gcd() {} //DTOR
SC_HAS_PROCESS(gcd);
};

void gcd::m_ST0() {
  if ((go_i.read() == 1)) {
    data_x = x_i.read();
    data_y = y_i.read();
    ev_m_ST1.notify(SC_CLK_PERIOD, SC_NS);
   } else {
    ev_m_ST0.notify(SC_CLK_PERIOD, SC_NS);
  }
} //end process st0

void gcd::m_ST1() {
  ev_m_ST2.notify(SC_CLK_PERIOD, SC_NS);
} //end process st1

void gcd::m_ST2() {
  if ((data_x != data_y)) {
    if ((data_x < data_y)) {
      data_y = (data_y - data_x);
    } else {
      data_x = (data_x - data_y);
    }
    ev_m_ST1.notify(SC_CLK_PERIOD, SC_NS);
  }
  else
  {
```

Figure 5. Initial VHDL FSMD RTL and generated SystemC ASM implementations of GCD

ACKNOWLEDGMENTS

The work has been supported by EU FP7 STREP project BASTION, institutional research grant IUT 19-1, and research grants 8478, 9429, funded by Estonian Ministry of Education and Research, and by EU through the European Structural and Regional Development Funds.

REFERENCES

[1] B Bailey, G Martin. *ESL Models and their Application: Electronic System Level Design and Verification in Practice (Embedded Systems)*, Springer. 2012.

[2] Benini, L.; Bertozzi, D. ; Bruni, D.; Drago, N.; Fummi, F.; Poncino, M., Legacy SystemC co-simulation of multi-processor systems-on-chip. *IEEE Int'l Conf Computer Design*, 2002.

[3] Mendoza, F.; Kollner, C.; Becker, J.; Muller-Glaser, K.D., An automated approach to SystemC/Simulink co-simulation. *IEEE Int'l Symp. System Prototyping (RSP)*, 2011.

[4] Uniform SystemC Co-Simulation Methodology for System-on-Chip Designs. IEEE CyberC, 2012.

[5] Model studio, Carbon Design Systems, http://carbondesignsystems.com

[6] HIFSuite, EDA-Lab, http://hifsuite.edalab.it, 2012

[7] VHDL-to-SystemC-Converter, Eberhard-Karls-University of Tübingen, http://www-ti.informatik.uni-tuebingen.de/~systemc/, 2012

[8] VH2SC, HT-Lab, http://www.ht-lab.com, 2012

[9] R. Görgen, J.H. Oetjens, W. Nebel, Automatic integration of hardware descriptions into system-level models, *Proc. of IEEE DDECS*, 2012.

[10] W. Snyder, Verilator, Veripool, http://www.veripool.org/wiki/verilator

[11] Edwin Naroska, "FreeHDL," http://www.freehdl.seul.org/, 2012.

[12] Bombieri, N.; Fummi, F.; Pravadelli, G., A methodology for abstracting RTL designs into TL descriptions. *4th ACM/IEEE International Conference on Formal Methods and Models for Co-Design, MEMOCODE '06*, pp. 103-112, 2006.

[13] Syed, S.A.; Jenihhin, M.; Raik, J., "Extensible open-source framework for translating RTL VHDL IP cores to SystemC," *IEEE 16th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, pp.112,115, 8-10 April, 2013.

[14] Syed, S.A.; Jenihhin, M.; Raik, J., "Abstraction of clock interface for conversion of RTL VHDL to SystemC," *IEEE International Advance Computing Conference (IACC)*, pp.50,55, 21-22 Feb., 2014.

[15] D. Gajski, N. Dutt, S. Allen, C.Wu, and Y. Lin. *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, first ed., 1992.

[16] K. Cheng and A. Krishnakumar. Automatic Generation of Functional Vectors Using the Extended Finite State Machine Model. *ACM Trans. on Design Automation of Electronic Systems*, vol. 1(1):pp. 57–79, 1996.

[17] VHDL Analyzer and Utility Library (VAUL) http://www.freehdl.seul.org/

[18] Samuel C. Lee. "Digital circuits and logic design", Prentice Hall PTR, 1976.

[19] S. Baranov, "Logic and System Design of Digital Systems", TUT Press, Tallinn 2008.

[20] V. Sklyarov, "Hierarchical Finite-State Machines and Their Use for Digital Control", *IEEE Trans. on VLSI Systems*, vol. 7, vo 2, pp. 222-228, 1999.

[21] Takach, A.; Wolf, W., "An automaton model for scheduling constraints in synchronous machines," *IEEE Transactions on Computers*, vol.44, no.1, pp. 1-12, Jan 1995.

[22] Pong P. Chu, *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability*, 2006, Wiley-IEEE Press.

Appendix B

Research paper [B]

Syed, Saif Abrar; Jenihhin, Maksim; Raik, Jaan. “SystemC-Based Loose Models for Simulation Speed-Up by Abstraction of RTL IP Cores.” IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems, Belgrade, Serbia, 2015. pp 1-4.

SystemC-Based Loose Models for Simulation Speed-Up by Abstraction of RTL IP Cores

Syed Saif Abrar[1,2], Maksim Jenihhin[2], Jaan Raik[2]
[1]IBM, Bangalore, INDIA; [2]Tallinn University of Technology, Tallinn, ESTONIA

Abstract*— The rapid increase of embedded systems design complexity has resulted in emergence of design methodologies at higher levels of abstraction such as Electronic System Level (ESL) and Transaction Level Modeling (TLM) with SystemC language as the main instrument. In practice, system architects and system integrators often have access to a library of legacy Register Transfer Level (RTL) IP (Intellectual Property) cores or obtain new ones from IP design houses. To address architectural exploration, early prototyping and simulation performance, such RTL IP cores are manually recreated at more abstract levels, which implies significant and error-prone effort. The current paper proposes an approach for automated abstraction of the computational part of cycle-accurate RTL IP cores to untimed TLM using a novel concept of SystemC-based Loose Models (SCLM). SCLMs provide for an instrument to neglect design model parts irrelevant for particular manipulation step of the abstraction process, thus simplifying the abstraction flow. As a result, the computational complexity of the abstraction process is reduced, thus increasing the overall scalability. The proposed abstraction flow is demonstrated on a set of benchmark designs and the first experimental results prove feasibility of the proposed approach and also show considerable simulation speed-up.

I. Introduction

Embedded systems are becoming increasingly complex and large, posing hardware and software design challenges. The time-to-market constraints push design vendors towards optimization of the design flow. The traditional design approaches at Register-Transfer Level (RTL) based on VHDL or Verilog are no longer sufficient. This has resulted in emergence of design methodologies at higher levels of abstraction such as Electronic System Level (ESL) and Transaction Level Modelling (TLM) with SystemC language as the main instrument.

Electronic System Level (ESL) design is an emerging methodology to model the embedded systems at a high level of abstraction. Transaction Level Modeling (TLM) is a widely recognized and adopted approach to realize the ESL in system design. TLM models are applicable for a range of design tasks, e.g. early software development, performance analysis, architecture exploration, HW/SW partitioning and early system verification. In practice, system architects and system integrators often have access to a library of legacy Register Transfer Level (RTL) HW IP cores or obtain new ones from IP design houses. Here, approaches for mixed RTL-TLM cosimulation based on transactors have a number of drawbacks, such as slow simulation and complicated debug. To address simulation performance and ease of debug flow, such RTL IP cores are manually reimplemented at more abstract levels, which implies significant tedious and error-prone effort.

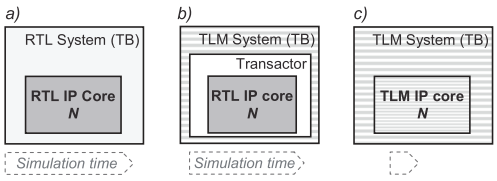


Fig. 1 Exploitation of legacy RTL IP cores in system design

Fig.1 demonstrates exploitation of a legacy RTL IP core *N* in: (a) RTL systems, e.g. RTL testbenches (TB), which it was initially created for; (b) emerging TLM systems, e.g. a TLM TB, keeping the RTL IP core *N* unchanged while developing an appropriate transactor; and (c) TLM systems when the IP core N is abstracted to the level of the environment where it will be used. The simulation speed of the computations performed by the IP Core N remains slow for both (a) and (b). Moreover, in (b) it serves as a bottleneck for the complete fast TLM system simulation. (c) is the fastest setup for simulation but assumes RTL IP core N abstraction till the level of the surrounding system.

This paper introduces a novel approach (methodology and implementation) to automated abstraction of RTL IP cores into high-level SystemC models using SystemC-based Loose Models (SCLMs). As shown in the next Section, there have been several works on translating as well as on abstracting RTL models to SystemC. However, **this paper goes beyond the existing state-of-the-art in the following aspects:** 1) It *eliminates the need for semantic analysis and elaboration* of the source RTL IP. 2) It *reduces the model size for manipulations*, thus contributing to higher scalability. 3) It provides a *readable SystemC abstraction* for RTL designs.

II. Related Works

There have been a number of tools facilitating abstraction of RTL VHDL/Verilog designs into SystemC. Among the commercial solutions there is HIFSuite [1] that is a design and verification framework addressing manipulation and integration of heterogeneous design parts. It allows translation of RTL VHDL descriptions into RTL as well as abstraction into TLM SystemC. The output result also does not consider human-readability and correspondence to the source VHDL. The approach supports equivalence checking to prove the correctness of the result.

Non-commercial solutions are represented by an approach published by OFFIS [2] assumes creation of readable SystemC representations from VHDL that are targeted to be wrapped and simulated in the Simulink environment. There are known approaches for creating

SystemC models from Verilog (e.g. [3] and [4]) and tools targeting creation of other C++ subsets. [5] discusses a methodology for abstracting RTL descriptions to different TLM levels. Finally, in the authors' previous work a tool for readable translation of RTL VHDL descriptions to SystemC was developed in [6] and in [7], the concept of abstracting clock interfaces was introduced.

In the current paper we propose an approach to automated RTL IP cores abstraction to a higher level using a novel concept of SystemC-based Loose Models. This approach reduces the computational complexity of the abstraction process, eliminates the need for semantic analysis and elaboration of the source RTL IP, increases the overall scalability and provides a readable SystemC abstraction for RTL designs. As a result a considerable simulation speed-up is achieved.

III. Loose Modeling

In this paper, we introduce a concept of *loose modeling* to computing systems design and analysis domain. The concept is explained by the following basic definitions.

Definition 1. A *code entity* C is a textual description of a system in some programming language, hardware description language or format.

Definition 2. A *parse tree* (aka concrete syntax tree) P(N,B) of a code entity C is a structure consisting of a set of nodes $n_i \in N$ with a single root node $n_0 \in N$ and a set of branches $b_k \in B$ connecting the nodes. The nodes N of the parse tree P are strings forming the atomic constructs of the language of the code entity.

Definition 3. *A loose model* M is a formal model *generated* from a reduct of the parse tree P of an initial code entity C. The reduct $\{P_K(K,B_K)\}$ is a set of sub-trees of P which contain the nodes $n_j \in K$, where $K \subset N$ and of branches $B_K \subset B$ connecting the nodes K. The nodes K are selected by a *key pattern matching function* $\Phi(P)$. The reduct $\{P_K(K,B_K)\}$ is referred to as *key elements* of P. Nodes $L \subset N$, $L=N \setminus K$ that are not part of P_K, and the branches $B_L \subset B$ connecting them form another set of sub-trees of $\{P_L(L,B_L)\}$, which is referred to as *load elements* of P.

In case *recovery* of the model back into a new code entity C' is needed, the *loose model* M also needs to maintain links to the load elements $\{P_L(L,B_L)\}$.

To explain, *key elements* are a reducted representation of a parse tree P of a code entity C to its semantic subset relevant to a specific task (e.g. a manipulation on FSM states). *Load elements*, in turn, are elements of a formal model irrelevant to the specific task and left over after reduction of P.

Definition 4. We refer to converting a loose model M to a different loose model M' as a *manipulation step.* There may occur several manipulation steps in a series. Eventually, after the manipulations, the resulting loose model M' is *recovered* to a parse tree P', or directly to a new code entity C' by combining the new key elements $\{P_K'(K',B_K')\}$ generated from the manipulated model M' with the load elements $\{P_L(L,B_L)\}$. Note that after manipulation the elements $\{P_K'(K',B_K')\}$ may be different from the initial key elements $\{P_K(K,B_K)\}$.

As opposed to a loose model, a model that takes into account the semantics and represents the full functionality of the code entity is called a *strong model.* For instance, Extended Finite Sate Machine model (EFSM) is a strong model of a code entity since it represents the complete functionality of the latter. A loose model of the entity (and also of the EFSM) can be a sub-FSM with selected states and transitions relevant to a specific manipulation task as the key elements, while the assignments, initializations, irrelevant states with transitions and the reminder of the entity is not included to the manipulation.

Loose models are exploited to reduce computational complexity and size of the model, where manipulation is performed on and may be subject to incomplete formal analysis (e.g. verification). As a wide scope of practical tasks in the domain, functional correctness of manipulations performed based on loose models can be verified using simulation based approaches or to a limited extent using partial formal verification. In the current paper, we apply manipulations on the loose model in order to perform abstraction from RTL to a higher level.

Definition 5. A *SystemC based Loose Model* (SCLM) is a loose model generated from a system design model (i.e. a code entity) expressed in the SystemC language. An SCLM represents particular aspects of the implementation relevant to the given manipulation task on a system design model.

For instance, different SCLMs can describe the same system design implementation (code entity) at different levels of abstraction, with different coding styles/flavors or may rely on different key elements. The same system design implementation may be represented by SCLM' such as clocked RTL FSM and by SCLM" as untimed algorithm-level flow-chart. SCLM' and its key elements sets of states and transitions may be appropriate for particular manipulations (e.g. optimizations) while SCLM" may be appropriate for a different set of tasks.

IV. An Approach for Abstraction using SCLM

The loose modeling based abstraction flow is shown in Fig. 2 and it takes place as follows. As a first step, a parse tree P of the RTL SystemC IP core (SC FSMD RTL clocked) is generated. Key elements are identified from the parse tree by traversing it implementing a tree-walking algorithm and applying a key pattern matching function Φ in order to obtain key elements and load elements. The load elements are saved into a repository called the bank of Load Elements (LE). Based on the key elements, a loose model M is created, which maintains the links to the load elements.

In the current abstraction approach, the loose model ($SCLM_1$) is based on an FSMD representation (i.e. FSM with datapath embedded) generated from the key elements of the RTL code entity. This loose model is manipulated to obtain first a SystemC goto model ($SCLM_2$), then, after further manipulations, an optimized goto model ($SCLM_3$) and finally an untimed optimized goto model ($SCLM_4$). Any of these loose models may be recovered to a SystemC code by replacing the loose model by SystemC constructs and substituting the loose model's links by SystemC code corresponding to load elements.

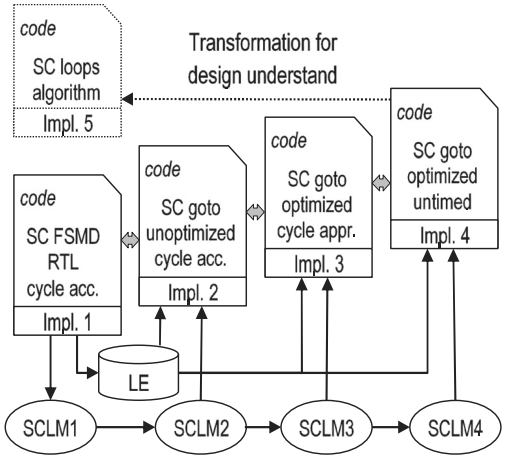


Fig. 2 Automated RTL to TLM abstraction flow based on SCLMs

All of the abstraction levels obtained by the proposed approach are based on goto models, which can be easily transformed to a readable structured form for design understand by applying automated recognition of control loops (e.g. WHILE, FOR, …). The resulting SystemC code represents the algorithmic level description of the IP core.

Note, that the extent up to which the abstraction can be performed is dependent on the environment interacting with the IP core. For a cycle accurate environment, goto model or optimized goto model can be utilized, whereas for untimed environments the abstraction may proceed until the untimed optimized goto model. The next Section explains the manipulations and the different abstraction levels on a case study example.

V. Case Study and Experiments

The proposed approach is explained on the b07 benchmark [10]. Parts of the design code have been omitted. Fig. 3 illustrates IP implementations in SystemC with snips and visualizations of their SCLMs. For this case study we consider the source code relevant to three states of the b07 FSM (START, LOAD_X and INCREMENTA). The first column in the figure shows RTL FSMD implementation of the IP design. A corresponding generated $SCLM_{FSMD}$ is demonstrated in the lower part of the table. It is based on design states as the key elements, the switch statement is executed on positive edge event of the clock signal. $SCLM_{FSMD}$ was further transformed to $SCLM_{GT}$, where the key elements are goto-labels. A recovered SystemC code (also utilizing necessary load elements) is provided in the second column. As the next step, a manipulation on $SCLM_{GT}$ is performed to optimize out labels related to computation only and not involved in input/output read/writes. An optimized $SCLM_{GT\_O}$ and its recovered SC code are demonstrated in the third column. The load elements linked to the optimized out LOAD_X label part are recovered as a part of the START label code.

These three SCLMs are using clock-based synchronization also in the computation part. However, the first two SCLMs are cycle-accurate, whereas the third one is cycle-approximate and here the semantics of the clock signal *clk* is different. The last column of Fig. 3 demonstrates $SCLM_{GT\_O\_U}$ where the computational part of the IP core is made untimed. The labels relevant to communication with the simulation environment (i.e. label START) use a synchronization signal *sync*. The final recovered TLM SC code is shown in the last column.

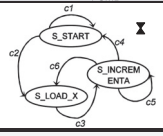
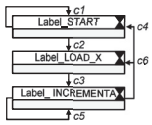
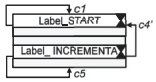
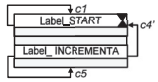
The last row in the table demonstrates exploitation of the abstractions in the three simulation environments as described in Fig 1 (Section 1). The initial RTL FSMD model IP_{FSMD} as well as its goto-version IP_{GT} can be exploited directly with RTL TB (column 1) or with a TLM TB (column 2) using a Transactor (column 3). The abstracted cycle-approximate optimized IP_{GT_O} would need a different transactor considering the smaller number of wait statements required for computation. The untimed SC goto code $IP_{GT\_O\_U}$ is used directly with the TLM TB (column 4). Please note the signal sync for the synchronization of computation results between the IP and its simulation environment.

The methodology proposed in this paper was implemented as a part of an open source design analysis framework [9] and exercised on a set of RTL benchmark designs. *gcd* is an implementation of the greatest common divisor algorithm, *b02* and *b07* are designs from the ICT'99 benchmarks family [10] Here we use the complete b07 benchmark and only its part as in the case study described in Section V. The *codec* benchmark is a data dominated design which is a product of high level synthesis tool Synthagate [11]. It has 89 states and 900 lines of RTL FSMD VHDL code.

Table 1 presents experimental results for simulation speed-up provided by different stages of abstraction. The benchmarks were abstracted to untimed TLM implementation using SCLMs. Simulation time of SystemC *RTL FSMD* is taken as the base for comparison. The bold values in the last column demonstrate speed-up achieved by the generated untimed SystemC implementation with regards to the initial RTL implementation. Here, data dominated designs such as *b07* and *codec* demonstrate a speed-up up to 27.24 times.

Table 1. Simulation Speed-Up by Stages of Abstraction

Design	Parameter	FSMD	GT	GT_O	GT_O_U
gcd	*speed-up (times)*	1	1.25	1.71	**2.65**
	# key elements	3 states	3 labels	2 labels	2 labels
	# wait statements	3	3	2	1
b02	*speed-up (times)*	1	1.03	1.03	**1.94**
	# key elements	7 states	7 labels	5 labels	5 labels
	# wait statements	7	7	5	1
b07	*speed-up (times)*	1	1.09	1.10	**12.08**
	# key elements	8 states	8 labels	4 labels	4 labels
	# wait statements	8	8	4	1
codec	*speed-up (times)*	1	1.35	1.42	**27.24**
	# key elements	89 states	89 labels	59 labels	59 labels
	# wait statements	89	89	59	1

IP_{FSMD}: SystemC RTL FSMD (cycle-accurate)	IP_{GT}: SystemC goto code (unoptimized, cycle-accurate)	IP_{GT_O}: SystemC goto code (optimized, cycle-approximate)	$IP_{GT\_O\_U}$: SystemC goto code (optimized, untimed)
`SC_MODULE(sc_b07) {` ` sc_out<int>punti_retta;` ` sc_in<bool>start,reset, clk;` `<snip1>` `  SC_THREAD(proc_fsmd);` `  sensitive << reset << clk;` `void sc_b07::proc_fsmd()` `<init>` `<snip2>` `wait(clk.posedge_event());` `switch(stato) {` `  case S_START:` `  if(start.read()==1)` `  {<snip3>` `  stato =S_LOAD_X; }` `  else` `  {stato =S_START; <snip4>}` `  break;` `  case S_LOAD_X:` `  x = mem[mar];` `  <snip5>` `  Stato =S_INCREMENTA; break;` `  case S_INCREMENTA:` `  if (mar != lung_mem)` `  {<snip6>` `  mar = (mar+1) % 16;` `  stato = S_LOAD_X;}` `  else` `  {if(x==2)` `  {<snip7>` `  punti_retta = cont+1;` `  stato=S_START;}` `  else` `  { stato=S_INCREMENTA;}}}` `<snip8>`	`SC_MODULE(sc_b07) {` `sc_out<int>punti_retta;` `sc_in<bool>start, reset, clk;` `<snip1>` `  SC_THREAD(proc_GT_unopt);` `  sensitive << reset << clk;` `void sc_b07:: proc_GT_unopt ()` `<init>` `<snip2>` `label_START:` `  wait(clk.posedge_event());` `  if(start.read()==1)` `  {<snip3> goto label_LOAD_X;}` `  else` `  {<snip4> goto label_START; }` `label_LOAD_X:` `  wait(clk.posedge_event());` `  x = mem[mar];` `  <snip5>` `  goto label_INCREMENTA;` `label_INCREMENTA:` `  wait(clk.posedge_event());` `  if (mar != lung_mem)` `  {<snip6>` `  mar = (mar+1) % 16;` `  goto label_LOAD_X;}` `  else` `  {if(x==2)` `  { <snip7>` `  punti_retta = cont + 1;` `  goto label_START;}` `  else` `  { goto label_INCREMENTA;}}` `<snip8>`	`SC_MODULE(sc_b07) {` `sc_out<int>punti_retta;` `sc_in<bool>start, reset, clk;` `<snip1>` `  SC_THREAD(proc_GT_opt);` `  sensitive << reset << clk;` `void sc_b07:: proc_GT_opt ()` `<init>` `<snip2>` `label_START:` `  wait(clk.posedge_event());` `  if(start.read()==1)` `  {<snip3>` `  }` `  else` `  { <snip4> goto` `label_START;}` `  x = mem[mar]; //from LOAD_X` `  <snip5>` `label_INCREMENTA:` `  wait(clk.posedge_event());` `  if (mar != lung_mem)` `  {<snip6>` `  mar = (mar+1) % 16;` `  }` `  else` `  {if(x==2)` `  { <snip7>` `  punti_retta = cont + 1;` `  goto label_START;}` `  else` `  { goto label_INCREMENTA;}}` `<snip8>`	`SC_MODULE(sc_b07) {` `sc_out<int>punti_retta;` `sc_in<bool>start;` `<snip1>` `  SC_THREAD(proc_GT_opt_ut);` `  sensitive << sync;` `void sc_b07:: proc_GT_opt_ut ()` `<init>` `<snip2>` `label_START:` `  wait(sync.posedge_event());` `  if(start.read()==1)` `  {<snip3>` `  }` `  else` `  { <snip4> goto label_START;}` `  x = mem[mar]; //from LOAD_X` `  <snip5>` `label_INCREMENTA:` `  if (mar != lung_mem)` `  {<snip6>` `  mar = (mar+1) % 16;` `  }` `  else` `  {if(x==2)` `  { <snip7>` `  punti_retta = cont + 1;` `  goto label_START;}` `  else` `  { goto label_INCREMENTA;}}` `<snip8>`
$SCLM_{FSMD}$	$SCLM_{GT}$	$SCLM_{GT\_O}$	$SCLM_{GT\_O\_U}$
			

Testbenches for cycle-accurate IPs

RTL TB for $IP_{FSMD\,/\,GT}$	TLM TB for $IP_{FSMD\,/\,GT}$		TLM TB for $IP_{GT\_O\_U}$
	TB	Transactor	
`SC_MODULE(sc_b07_rtl_tb) {` ` sc_in<int> t_punti_retta;` ` sc_out<bool> t_start, t_reset, t_clk;` `void sc_b07_tb_rtl::test() {` `<snip>` `  t_reset.write(1);` `  wait(clk.posedge_event());` `  wait(clk.posedge_event());` `  t_reset.write(0);` `  wait(clk.posedge_event());` `  t_start.write(1);` `  wait(clk.posedge_event());` `while(t_punti_retta.read()==0)` `  wait(clk.posedge_event());` `  t_start.write(0);` `  wait(clk.posedge_event());` `}`	`#include "rtl2tlm_if.h"` `/* methods in this interface are implemented in the transactor */` `SC_MODULE(sc_b07_rtl2tlm_tb) {` ` void sc_b07_tb_rtl2tlm::test() {` `<snip>` `  do_reset();` `  set_start();` `  set_sync();` `  while(t_punti_retta.read()==0)` `  wait(clk.posedge_event());` `  clear_start();` `}`	`#include "rtl2tlm_if.h"` `SC_MODULE(sc_b07_rtl2tlm_TX) {` ` sc_in<int> t_punti_retta;` ` sc_out<bool> t_start, t_reset, t_clk;` `void sc_b07_TX::do_reset() {` `  wait(clk.posedge_event());` `  t_reset.write(1);` `  wait(clk.posedge_event());` `  t_reset.write(0);` `}` `void sc_b07_TX::set_start() {` `  t_start.write(1);` `}` `void sc_b07_TX::clear_start() {` `  t_start.write(0);` `}` `void sc_b07_TX::set_sync() {` `  wait(clk.posedge_event());` `  t_sync.write(0);` `  wait(clk.posedge_event());` `  t_sync.write(1);}`	`#include "b07_tlm_if.h"` `SC_MODULE(sc_b07_tlm_tb) {` ` void sc_b07_tb_tlm::test() {` `<snip>` `  do_reset();` `/* untimed implementation */` `  set_start();` `/* untimed implementation */` `  set_sync();` `/* untimed implementation */` `  while(get_punti_retta() == 0)` `  wait(SC_ZERO_TIME);` `/* wait for a delta-cycle to get the updated result */` `  clear_start();` `}`

Fig. 3 Code snapshots for abstraction of *b07* benchmark IP design (including testbenches as in Fig. 1).

VI. Conclusions

This paper introduced a novel approach to automated abstraction of RTL IP cores into TLM SystemC models using SystemC-based Loose Models eliminating the need for semantic analysis and elaboration of the RTL IP and reducing the model size for manipulations, thus contributing to higher scalability. Experimental results show up to 27.24x simulation speed-up in comparison to the RTL IP cores.

References

[1] HIFSuite, EDA-Lab, http://hifsuite.edalab.it, 2014

[2] http://www-ti.informatik.uni-tuebingen.de/~systemc/, 2014

[3] VH2SC, HT-Lab, http://www.ht-lab.com, 2014

[4] R. Görgen, et al., Automatic integration of hardware descriptions into system-level models, *Proc. of IEEE DDECS*, 2012.

[5] W. Snyder, Verilator, http://www.veripool.org/wiki/verilator

[6] Edwin Naroska, "FreeHDL," http://www.freehdl.seul.org/, 2014

[7] Bombieri, N, et al., A methodology for abstracting RTL designs into TL descriptions. *MEMOCODE '06*, pp. 103-112, 2006.

[8] Syed, S. A.; Jenihhin, M.; Raik, J., Extensible Open-Source Framework for Translating RTL VHDL IP Cores to SystemC. *15th IEEE DDECS Symposium*, April 8-10, 2013, pp. 112 - 115.

[9] Syed, S. A.; Jenihhin, M.; Raik, J., Abstraction of clock interface for conversion of RTL VHDL to SystemC. *IEEE IACC*, 2014.

[10] Politecnico di Torino, "ITC-99 Benchmarks," 1999, http://www.cad.polito.it/tools/itc99.html

[11] S. Baranov, Synthagate, http://synthezza.com/

Appendix C

Research paper [C]

Syed, Saif Abrar; Jenihhin, Maksim; Raik, Jaan. “Abstraction of clock interface for conversion of RTL VHDL to SystemC.” IEEE International Advance Computing Conference, Gurgaon, India, 2014. pp 50-55.

Abstraction of Clock Interface for Conversion of RTL VHDL to SystemC

Syed Saif Abrar, Maksim Jenihhin, Jaan Raik
Tallinn University of Technology
Tallinn, ESTONIA
{saif | maksim | jaan} @ati.ttu.ee

***Abstract*—IP design-houses are hard-pressed by their customers to provide SystemC models of their portfolio IPs, despite already existing VHDL views. VHDL IPs can be translated to SystemC, ensuring correctness, quality and maintainability of the translated code. This paper explores optimization scenarios that affect the simulation performance, resulting in upto 38% faster - simulation. In addition to the plain VHDL-to-SystemC conversion, there are possibilities of alternate implementations for a SystemC model. This paper explores these alternate scenarios to get 25% better simulation-speed. The optimization methodologies in this paper are relevant to architects, designers, verification-teams, IP design-houses that need to provide high-speed simulation-models, and can be used for optimizing simulation tools as well system-level models.**

Keywords—VHDL, SystemC, conversion, optimization

I. INTRODUCTION

Consumer electronic devices are becoming increasingly complex, presenting lots of hardware and software design challenges. The traditional design approaches at Register Transfer Level (RTL) using VHDL or Verilog are no longer suitable, resulting in the search for a newer abstraction-level that enables unified development of the System-on-a-Chip (SoC) designs. This level has been called system level, behavioral level, C-level, algorithmic level, Electronic System Level (ESL), etc.

Electronic System Level (ESL) [1] design is an emerging methodology to model the (sub-)system at a high-level of abstraction. Transaction Level Modeling (TLM) is the widely recognized and adopted approach to realize the ESL in practice. TLM models are applicable for a range of design-tasks, e.g. early software development, performance analysis, architecture exploration, HW/SW partitioning, etc. A big pain-point for the IP design-houses is the challenge of providing the TLM models of their IPs. Their customers are increasingly asking for SystemC models [2] along with the RTL description. Developing the SystemC-models manually has a potential of mismatch between RTL and SystemC, and is impractical in terms of time and effort. Hence, there is an urgent need of automated generation of SystemC models from existing RTL description.

The rest of the paper is organized as follows. Section 2 gives an overview of the related work. Section 3 describes the VHDL-to-SystemC translation methodology. Section 4 details the abstraction of clock-interface. Section 5 describes the optimizations of VHDL combinational-statements when translated to SystemC. Finally, section 6 discusses the experiments and results for both the approaches discussed in the previous sections.

II. RELATED WORK

Carbon Model Studio [3] is a commercial offering that allows creating configurable SystemC models from RTL VHDL or Verilog descriptions. It does not intend to create human-readable output. HIFSuite [4], [5] is a design and verification framework addressing manipulation and integration of heterogeneous design parts. The output result also does not consider human-readability and correspondence to the source VHDL. The approach supports equivalence checking to prove the correctness of the result.

Example of non-commercial and free solutions to generate SystemC from VDHL are VHDLParser by University of Tuebingen [6] and VH2SC by HT-Lab [7]. Both approaches consider mapping of the source VHDL to a limited set of SystemC constructs. These tools have demonstrated significant limitations, do not guarantee equivalence and they are not maintained anymore. VHDLParser dates to 2001 and addresses SystemC 1.0. Closed sources of the tools do not allow engineers to extend them to their needs.

The most relevant approach is published by OFFIS in [8]. It assumes creation of readable SystemC representations from VHDL that are targeted to be wrapped and simulated in the Simulink environment. The approach is claimed to support industrial designs, however only an illustrative example details are available [9]. This practical work also does not provide for equivalence checking mechanisms or results. There are known approaches for creating SystemC models from Verilog [10] and tools targeting creation of other C++ subsets [11], [12].

Optimizing the execution speed of compiled software is an active area of academic and industrial research. Source-level optimization of C/C++ code has been discussed in [13], whereas compiler-level optimizations are discussed in [14]. Simulation-speed is extremely important in the early stages of system-development. Several techniques to improve SystemC simulation-time are suggested in [15]. An interesting technique of process-splitting is introduced in [16].

VHDL

```
01  entity b09 is port (
02   reset,clock: in bit;
03   <snip>
04   y: out bit
05  );
06  end b09;
07
08  architecture BEHAV of b09 is
09   constant Bit_start : bit := '1';
10   <snip>
11   constant Zero_8 : bit_vector ( 7 downto 0)   :=
12  "00000000";
13   <snip>
14   signal d_in: bit_vector ( 8 downto 0);
15   <snip>
16
17  begin
18   process(clock,reset)
19    variable stato: integer range 3 downto 0;
20    <snip>
21   begin
22    if    reset = '1' then
23    stato := INIT;
24    <snip>
25    elsif clock'event and clock='1' then
26     case stato is
27     when INIT
28      =>
29       stato := RECEIVE;
30       <snip>
31     when LOAD_OLD
32      =>
33       if  d_in(0) = Bit_start
34       then
35        if  d_in(8 downto 1) = old
36        then
37         d_in <= Zero_9;
38         <snip>
39        else
40         <snip>
41        end if;
42         <snip>
43       else
44        d_in <= x & d_in(8 downto 1);
45       end if;
46      end case;
47     end if;
48   end process;
    end BEHAV;
```

SystemC

```
01    class sc_WORK_B09_BEHAV : public sc_module
02    {
03     public:
04     sc_WORK_B09_BEHAV(sc_module_name mn_);
05     sc_in < bool > RESET;
06     <snip>
07     sc_out < bool > Y;
08     private:
09      sc_signal< sc_bv < 9 >  > D_IN;
10      void process_line_23(void);
11       <snip>
12    };
13    const bool BIT_START = 1;
14    <snip>
15    const sc_bv  < 8 >  ZERO_8 = "00000000";
16
17    sc_WORK_B09_BEHAV ::
18    sc_WORK_B09_BEHAV(sc_module_name mn_)
19    : sc_module(mn_)
20    {
21     SC_METHOD(process_line_23);
22     sensitive  << CLOCK << RESET;
23     dont_initialize();
24    }
25    void sc_WORK_B09_BEHAV::process_line_23(void)
26    {
27     sc_int < 4 > STATO;
28     <snip>
29    if ( RESET.read() == 1 )
30    {
31     STATO = INIT;
32     <snip>
33    }
34    else
35    {
36     if ( (CLOCK.read() == 1)  )
37     {
38      if ( STATO == INIT )
39      {
40       STATO = RECEIVE;
41       <snip>
42      }
43      if ( STATO == LOAD_OLD )
44      {
45       if (D_IN.read().bit(0).to_bool()==BIT_START)
46       {
47        if ( D_IN.read().range(8,1) == OLD.read() )
48        {
49         D_IN = ZERO_9;
50          <snip>
51        }
52        else
53        {
54          <snip>
55        }
56       }
57       else
58       {
59        D_IN=concat(X.read(),D_IN.read().range(8,1));
60        <snip>
61       }
62      }
63     }
64    }
65    } //end sc_WORK_B09_BEHAV::process_line_23()
```

Fig. 1. ITC b09 translated from VHDL to SystemC using the proposed approach

III. VHDL TO SYSTEMC TRANSLATION

VHDL to SystemC translation follows a set a guidelines discussed in [17]. Without any loss of generality, Fig. 1 shows the ITC99 benchmark design b09 translated from VHDL to SystemC using our methodology. Only the code of interest is shown in Fig. 1, and the omitted code is marked with a <snip> tag. Translation methodology is described below by set of rules, referring to the Line-numbers in the SystemC column to aid the explanations.

A. *Handling multiple architecture definitions*

VHDL enables definition of multiple architectures for a single entity. However, SystemC is limited in this regards that there is only a single SystemC class available for each representation. A practical approach in this regard is to derive the name of each SystemC module from a combination of the VHDL entity and architecture names, e.g. by concatenating VHDL entity and architecture names, as in Line-01.

B. *Making SystemC module constructor on-the-fly*

SystemC module needs a constructor, unlike a VHDL model. Some information, like sensitivity-list, is part of the SystemC constructor that is only available in VHDL process declaration, as in Line-18. This demands that the information for the SystemC constructor be gathered while parsing the VHDL module, accumulated on-the-fly and finally written to the SystemC model code.

C. Using constructors rather than SC_CTOR

VHDL allows model parametrization using generics. The methodology uses class constructor, as shown in Line-04, instead of SC_CTOR, to use parameters as VHDL generics.

D. Virtual Destructors

As it is also discussed in the guideline 7 in Effective C++, if the SystemC model has any virtual function then it should have a virtual destructor.

E. Mapping VHDL and SystemC port types

Both VHDL and SystemC have similar types of ports. As in Line-05 and Line-07, sc_in and sc_out for VHDL in and out ports respectively. Additionally, VHDL *buffer* port can be mapped to *sc_inout* port.

F. Naming the SystemC process

SystemC process names are required, whereas optional in VHDL. The methodology uses VHDL process name, if it exists, or derives it from the process line-number, as shown in Line-10.

G. Exploiting native C++ data-types for faster simulation

This is conventional wisdom to use C++ datatypes. As shown in Line-13, *bool* type is used instead of SystemC defined SC_*bit* data type. However, this approach must be used with care, to make sure that the left-out values (e.g. 'X'/'Z' in this case) are not used anywhere in the code.

H. Writing SystemC module constructor on-the-fly

SystemC module has a constructor with information, like sensitivity list, that is spread across VHDL code. Information for the SystemC constructor has to be gathered while parsing the VHDL module, accumulated on-the-fly and finally written to the SystemC file. This is shown in Line-20.

I. Translating the VHDL process

SystemC has SC_METHOD and SC_THREAD, similar to a VHDL process. A VHDL process with a wait statement is implemented as SC_THREAD, otherwise SC_METHOD might be preferred. As shown in Line-21, SC_METHOD is used, and the sensitivity of the VHDL-process becomes the sensitivity of SystemC translation, as shown in Line-22.

J. Preventing invocation of SystemC process at start-up

SystemC scheduler invokes every process (SC_METHOD or SC_THREAD) at the start of simulation, even in absence of any event! Use dont_initialize() to stop this default invocation, as shown in Line-23.

K. Using port-methods for clarity

SystemC uses same operator '=' for variables and ports. To remove ambiguity in understanding the translated source-code, use .read() and .write() methods for ports and '=' for variable assignments. As in Line-29, port is read using its .read() method.

L. Handling clock-edge sensitivity

VHDL and SystemC use varying notations to describe clock edge sensitivity. Line-36 shows the translation for positive-edge clock sensitivity. But the process is invoked on both the edges of clock. Efficient and recommended, but elaborate approach is to analyze the VHDL to determine the clock sensitivity of interest, and then use it in SystemC. Such scheme is better for an event-based simulator like SystemC.

M. Translating switch-cases

VHDL allows variables, logic-types as well as ports in the switch-case construct, whereas SystemC allows only integers. Hence, VHDL switch literal must be converted to an integer, if possible. Another approach is to use if-then-else constructs, as shown in Line-38 and Line-43, where the VHDL switch literal

- uses a range of values
- cannot be converted to an integer, e.g. strings

N. Chain of SystemC library calls

Sometimes it is necessary in practice to employ a chain of SystemC library calls to achieve the desired behavior, e.g. comparing a single-bit value from a port, as shown in Line-45.

In addition to the above details highlighted in Fig. 1, few other rules are followed for translation, described below:

O. Operator precedence differences

This consideration is extremely important to follow. VHDL and SystemC have different precedence for certain operators. As is common, use parenthesis for clarity and overwriting precedence.

P. Handling concurrent VHDL statements

Concurrent VHDL statements can be handled as:

a) Single SC_METHOD for all concurrent statements, sensitive to all the source RHS operations in these statements. Drawback is that all the statements are executed whenever any RHS operation changes, affecting the simulation performance.

b) Separate SC_METHOD for each statement, sensitive to only this statement's source variable. This reduces the simulation overhead, but increases the code-size.

Q. Executing the SystemC model constructor

SystemC models instantiated inside a top-level module need their constructors to be executed, e.g. implementing a 4-bit adder from 4 instances of 1-bit adder.

SystemC code generated by the zamiaCAD[18] implementation has following highly desirable qualities:

Human readability: This is an important consideration about the usage of the generated SystemC code: whether the SystemC code is only to be fed to a compiler or is it going to be maintained by human developers. Many VHDL to SystemC converters lack this feature and generate an obscure code which is not fit for human use. It goes a long way to decide the human relationship with the generated SystemC code.

Correspondence of the translated SystemC to VHDL: If a team of designers needs to maintain both the VHDL and SystemC code-bases, then it is appropriate to have a consistent view between the two. The translation mechanism must decide about this early on and take care of. Usage of the same module names, variables, constructs, etc. must be adhered too, unless SystemC does not support a feature inherently.

IV. ABSTRACTION OF CLOCK INTERFACE

The clock-interface is an essential part of a synchronous design. Various IP blocks in the design are fed by a single or multiple clock-signals to make the design synchronous. Such clock-signals are driven by a clock-generator at the system-level. Table-I shows a sample RTL VHDL description and the equivalent SystemC description. As shown in the Table-I, SystemC description consists of a clock in-port (CLOCK), SC-THREAD named P1() that is sensitive to the clock in-port. The sc_main() routine, instantiates a clock-generator and connects it to the clock in-port of the module.

TABLE I. VHDL AND SYSTEMC DESCRIPTIONS

VHDL	SystemC
entity b09 is port (clock: in bit;); end b09; architecture BEHAV of b09 is begin P1: process(clock) <snip> end process; end BEHAV;	class sc_WORK_B09_BEHAV : public sc_module { public: sc_in < bool > CLOCK; private: void P1(void); }; sc_WORK_B09_BEHAV :: sc_WORK_B09_BEHAV(sc_module_name mn_) : sc_module(mn_) { SC_THREAD(P1); sensitive << CLOCK; } #define SC_CLK_PERIOD 10 int sc_main(int argc, char **argv) { sc_clock my_sc_clk("my_sc_clk", SC_CLK_PERIOD, SC_NS); sc_WORK_B09_BEHAV my_mod("my_mod"); my_mod.CLOCK(my_sc_clk); sc_start(); return 0; } //end sc-main

A simple heuristic for recognizing the clock interface in an RTL VHDL is based on the following general observations:

1. Clock is 1-bit input port
2. Appears in the sensitivity list of a process
3. Appears with a 'event qualifier
4. Does not appear in any other control/data statement

Abstraction of clock-interface while generating SystemC from VHDL is based on the understanding of the SC scheduler. SC scheduler employs delta-cycles and events for proper scheduling g of various ports, channels, signals, interfaces, etc. Minimization of these delta-cycles and events in a SC description is the core idea behind abstraction of clock-interface, as dicussed below:

A. *Waiting on clock period* ***(OPT-A)***

A fundamental property of a clock-interface is its period of pulses. Simple designs have static clock-periods, whereas a complex design might employ varying clock-periods to achieve power savings. As shown in Table-I, an SC_THREAD waits on the arrival of clock-edge (positive, negative or both). This waiting requires the SC-scheduler to keep track of the changes in the clock-signal and trigger appropriate SC_THREAD based on the activity on these clock-signals. Load on the SC-scheduler can be reduced by specifying static clock-period for a wait statement inside an SC_THREAD, as shown in Table-I.

The SC-scheduler has to keep track of each signal specified in various sensitivity-lists and to trigger the appropriate SC_PROCESS / SC_THREAD whenever there is an update on any such signal. Once an SC_THREAD has been made to wait on a static clock-period, the clock-signal can be removed from its sensitivity-list, again to reduce burden on the SC-scheduler.

B. *Removal of clock-generator* ***(OPT-B)***

Once all the SC modules are made independent of the clock-signal by waiting on the static clock-period, the system-level clock-generator is redundant. Even if there is no modules working on the clock, the clock-generator keeps ticking and generating the clock-cycles. This is another source of burden on the SC-scheduler, consuming simulation-resources. Next optimization is to completely remove the system-level clock-generator, as it is of no use now.

Notes that this step needs to be done carefully, after completely analyzing the side-effects of this removal of the clock-generator. Regression-testing must follow the removal of clock-generator to ensure that the system-level design is still functional, without any breaks.

V. OPTIMIZATION FOR COMBINATIONAL STATEMENTS

The SystemC-model available from the methodology in Section-III is obtained by converting the VHDL model on a line-by-line basis. This plain translation approach does not take into account the simulation characteristics as well as implementation style of the SystemC model. Further optimization can be done to combinational statements, resulting in faster simulation-speed[19].

Combinational statements in VHDL are implemented outside of any VHDL process and are executed concurrently, in parallel. SystemC does not have this feature of concurrent statements, as all statements are executed within a SystemC process, either SC_METHOD or SC_THREAD. Since a VHDL combinational method does not have a 'wait' feature, only SC_METHOD is to be considered for converting combinational statements to SystemC.

Multiple combinational VHDL statements can be converted to SystemC SC_METHOD in the following 2 alternatives:

A. *Single SC_METHOD for all combinational statements*

This approach implements all VHDL combinational statements within a single SC_METHOD. The single SC_METHOD is made sensitive to all the source RHS elements in these statements. Table-2 shows an example of this approach. Advantage of this implementation is its simplicity and ease of implementation. Drawback is that all the statements in the SC_METHOD are executed even if only 1 RHS element changes.

TABLE II. Single SC_METHOD

VHDL syntax	SystemC syntax
architecture behav of E is begin X <= A and B; Y <= not A; Z <= X or Y; end behav;	SC_METHOD(P1); sensitive << A << B << X << Y; void P1(void) { X = A && B; Y = !A; Z = X || Y; }

TABLE III. Separate SC_METHOD's

VHDL syntax	SystemC syntax
architecture behav of E is begin X <= A and B; Y <= not A; Z <= X or Y; end behav;	SC_METHOD(P_X); sensitive << A << B; void P_X(void) { X = A && B; } SC_METHOD(P_Y); sensitive << A; void P_Y(void) { Y = !A; } SC_METHOD(P_Z); sensitive << X << Y; void P_Z(void) { Z = X || Y; }

B. *Separate SC_METHOD for each combinational statement*

In this approach, each combinational statement is implemented in a separate SC_METHOD. This SC_METHOD is made sensitive only to the RHS elements appearing in source VHDL statement. Table-3 shows an example of this approach. Advantage of this approach is that only the relevant SystemC statements are executed, within a particular SC_METHOD. Limitation is the increase in the source-code size due to an SC_METHOD implemented for each combinational statement.

VI. EXPERIMENTS

This section discusses the experiments and results for both the approaches discussed in previous sections.

A. *Clock-interface abstraction*

A graphics pipeline for edge-detection is used for the validation of the clock-interface abstraction methodology discussed in the earlier section. Fig-2 shows the graphics pipeline, which has Gaussian-blur and edge-detect operations performed on an image-frame being read from a file and the final image-frame being saved to the result file. The SC_THREADS gauss_blur() and edge_detect() are sensitive to the clock-signal sc_clk driving both the IP-models.

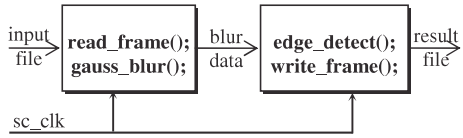


Figure 2. Graphics pipeline for edge-detection

Following setups are simulated for the above mentioned graphics pipeline:

1. BASE: Simulate the un-optimized SC description.
2. OPT-A: Both IP-models wait on the clock-period and the clock-signal is removed from the sensitivity-list.
3. OPT-B: Remove the system-level clock-generator.

Figure-3 shows the simulation-time for processing number of frames varying from 1 through 10, for the 3 simulation scenarios.

As seen from the Fig-3, the simulation-time increases linearly as the number of processed frames increases. Scenario OPT-A takes about 10% lesser simulation time than the BASE scenario, due to waiting on static clock-period and abstraction of the sensitivity-list. Removal of system-level clock-generator provides an optimization close to 38% as compared to the BASE, due to no more redundant clock-ticks being generated. This reduces enormous burden on the SC-scheduler, resulting in fast simulation speed.

Note that these techniques are general and can be used in any SC module, making them extremely useful for simulation-speed optimization goals.

B. *Combination statements optimization*

To experiment the 2 alternative methods of optimization, a module with 12 combinational-statements is selected. As is seen from the Figure-4, the multiple process approach takes about 25% lesser simulation-time, resulting in faster simulation-speed. This can be attributed to the fact that in multiple-process implementation, only a single statement is executed, resulting in optimized simulation-model.

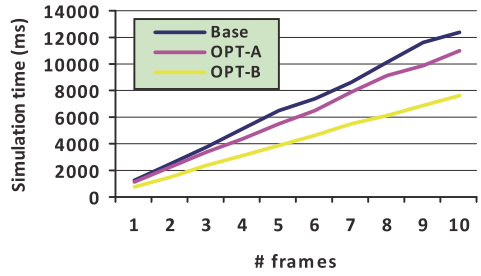


Figure 3. Simulation-time for frame processing

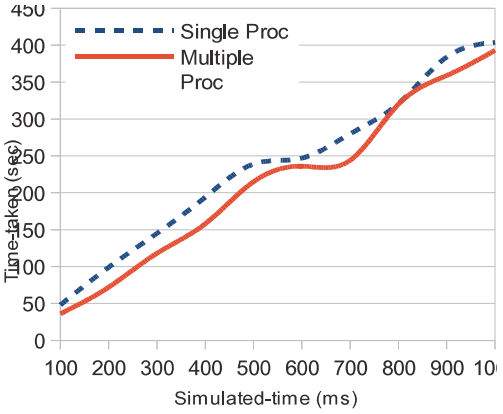


Figure 4. Simulation-time for single and multiple processes

VII. Conclusions

With the acceptance of TLM methodology for system-level design, IP-customers need ever increasing support for the TLM models. IP-providers are faced with time and accuracy challenges while manually converting RTL to TLM-models. This paper presented a methodology to automatically convert RTL VHDL to cycle-accurate SystemC. Abstraction of clock-interface provided 38% increase in simulation-time whereas optimization for combination-statements provided upto 25% increase in the simulation-speed. These results are extremely relevant for the SystemC TLM model providers and ESL/EDA-vendors.

VIII. Acknowledgements

The work has been supported in part by EU FP7 STREP project BASTION, Estonian ICT Program project FUSETEST, by European Union through the European Structural and Regional Development Funds and by Estonian SF grants 8478, 9429.

References

[1] ESL Models and their Application: Electronic System Level Design and Verification in Practice(Embedded Systems) B Bailey, G Martin. 2012.

[2] SystemC IEEE 1666-2011

[3] Carbon Design Systems, http://carbondesignsystems.com

[4] HIFSuite, EDA-Lab, http://hifsuite.edalab.it, 2012

[5] N. Bombieri, M. Ferrari, F. Fummi, et al., "HIFSuite: Tools for HDL Code Conversion and Manipulation," in EURASIP Journal on Embedded Systems, vol. 1155, no. 10, 2010.

[6] VHDL-to-SystemC-Converter, Eberhard-Karls-University of Tübingen, http://www-ti.informatik.uni-tuebingen.de/~systemc/, 2012

[7] VH2SC, HT-Lab, http://www.ht-lab.com, 2012

[8] R. Görgen, J.H. Oetjens, W. Nebel, Automatic integration of hardware descriptions into system-level models, Proc. of IEEE DDECS 2012, Tallinn, pp.105-110

[9] R. Görgen, "VHDL-to-SystemC Transformation," 2012, http://vhome.offis.de/ralphg/vhdl2sc.pdf

[10] W. Snyder, Verilator, Veripool, http://www.veripool.org/wiki/verilator

[11] Edwin Naroska, "FreeHDL," http://www.freehdl.seul.org/, 2012

[12] OStatic, "VHDLC," http://ostatic.com/vhdlc, 2012

[13] "Optimizing C++", Steve Heller, Prentice Hall, ISBN-13: 978-0139774300

[14] "The Next Generation of Compilers", IEEE Code Generation and Optimization, 2009.

[15] "Code optimization for enhancing SystemC simulation time", Alemzadeh, H., Aminzadeh, S. ; Saberi, R. ; Navabi, Z. IEEE East-West Design & Test Symposium (EWDTS), 2010

[16] "Speeding Up SystemC Simulation through Process Splitting", Naguib, Y.N., Guindi, R.S. DATE 2007.

[17] Extensible Open-Source Framework for Translating RTL VHDL IP Cores to SystemC, Syed Saif Abrar, Maksim Jenihhin, Jaan Raik; IEEE Int'l Symp. Design and Diagnostics of Electronic Circuits & Systems (DDECS), 2013

[18] zamiaCAD Open-source HW Design Framework, http://zamiacad.sf.net

[19] "Optimization Methodologies For Cycle-Accurate SystemC Models Converted From RTL VHDL", Syed Saif Abrar, Maksim Jenihhin, Jaan Raik. IP-SoC 2013.

Appendix D

Research paper [D]

Syed, Saif Abrar; Shyam, K.; Jenihhin, Maksim; Raik, Jaan; Babu, C. “Performance Analysis of Cosimulating Processor Core in VHDL and SystemC.” IEEE International Conference on Advances in Computing, Communications & Informatics, Mysore, India, 2013.pp 1-6.

Performance Analysis of Cosimulating Processor Core in VHDL and SystemC

Syed Saif Abrar
IBM, Bangalore, INDIA
saif.abrar@in.ibm.com

Shyam Kiran .A
Dept of ECE, Amrita School of Engineering.
Bangalore,INDIA
shyamkiran.a@gmail.com

Maksim Jenihhin, Jaan Raik
Tallinn University of Technology, Tallinn, ESTONIA
{maksim | jaan}@ati.ttu.ee

C. Babu
Dept of ECE, Amrita School of Engineering.
Bangalore,INDIA
c_babu@blr.amrita.eduline

***Abstract*—*Advances in SoC design complexities require newer methodologies and tools. Traditional RTL-level approach has become a bottleneck, resulting in emergence and standardization of SystemC as a design-language. IP design-houses are interested in providing SystemC models of their portfolio IPs, despite already existing VHDL views. This paper describes a methodology to translate existing VHDL IPs to SystemC, ensuring correctness, quality as well as maintainability of the translated code. The standard practice is to translate a subset of IPs at a time and cosimulate with the rest of the system to validate the translated IPs. Hence, this paper explores scenarios that affect the cosimulation performance. Varying cosimulation scenarios affect the performance upto 30%. Both the contributions of this paper, translation methodology and the cosimulation performance-analysis are relevant to a wider SystemC community, including designers and architects. Furthermore, the results can be used for optimizing cosimulation tools as well system-level-models.*

Keywords— VHDL; SystemC; Co-Simulation; Performance analysis.

I. INTRODUCTION

Consumer electronic devices are becoming increasingly complex, presenting lots of hardware and software design challenges. Hardware designers need performance evaluation to select proper architecture. Software developers need to develop and debug the software much before the availability of the physical hardware. The traditional design approaches at Register Transfer Level (RTL) using VHDL or Verilog are no longer suitable, resulting in the search for a newer abstraction-level that enables unified development of the System-on-a-Chip (SoC) designs. This level has been called *system level*, *behavioral level*, *C-level*, *algorithmic level*, *Electronic System Level (ESL)*, or just *a higher level* (i.e. higher than RTL).

Various languages and tools are proposed and tried out to enable different parts of the system level design. *Behavioral Verilog or VHDL*, *C*, *Java*, *C++ class libraries*, C-based *SpecC* and *HandelC* , *SystemVerilog*, and special-purpose languages like *Bluespec* and *Esterel* . Among these, *SystemC* has emerged as the dominant language, standardized as IEEE-1666 [1].

Product development companies are focusing their methodologies around SystemC. Architects, hardware designers, software developers, etc. have started using SystemC for their project tasks. Either complete or partial systems are now being modeled in SystemC, proliferating its usage and acceptance.

Thus, the IP providers are deeply pressed to make available the SystemC models of their legacy VHDL or Verilog IPs, to remain competitive and relevant in today's market. The abstraction level of the converted SystemC models can range from cycle-accurate RTL to untimed TLM (Transaction-Level Modeling). Availability of cycle-accurate SystemC model, instead of VHDL, has advantages across the product development, namely:

• Architects integrate and update their C/C++ algorithms directly in the hardware model.

• Hardware designers, transitioning from VHDL to SystemC, learn relationship between VHDL and SystemC.

• Verification engineers directly use SystemC testbenches and can avoid expensive co-simulation tools.

• Firmware developers working with assembly/C/C++ find familiar development environment for their tasks.

This paper presents a methodology to convert a VHDL design to cycle-accurate SystemC model. The methodology has been verified by converting Plasma RISC-core [2] from VHDL to SystemC and then co-simulating with the original VHDL test-suite accompanying the Plasma source-code. Cosimulation performance analysis is subsequently performed to analyze the aspects giving better and worse performance than the VHDL-only simulation.

The rest of the paper is organized as follows. Section 2 gives an overview of the state-of-the-art. Section 3 describes the proposed translation methodology. Sections 4 demonstrates the validation of the translation methodology. Section 5 discusses the cosimulation performance analysis and the results obtained. Section 6 concludes the paper, with direction for future work.

II. RELATED WORK

Among the commercial solutions there is *Carbon Model Studio* [3] that allows creating configurable SystemC models from RTL VHDL or Verilog descriptions. It is targeted mainly

Table 1. SystemC model constructor

VHDL syntax	*SystemC syntax*
`P1 : process(A,B)` `begin` `...` `end process P1;`	`class module : public sc_module` `{... void P1(void); …};` `module::module()` `{...` `SC_METHOD(P1); sensitive << A << B;` `...}`

Table 2. SystemC model naming

VHDL syntax	*SystemC syntax*
`entity E is . . . end E;` `architecture behav of E is` `begin . . . end behav;`	`class E_behav : public sc_module` `{ . . . };`
`architecture RTL of E is` `begin . . . end RTL;`	`class E_RTL : public sc_module` `{ . . . };`

at simulation speedup and does not intend to create human-readable output. *HIFSuite* [4] is a design and verification framework addressing manipulation and integration of heterogeneous design parts. Similarly, it allows dumping out RTL VHDL decryptions into SystemC. The output result also does not consider human-readability and correspondence to the source VHDL. The approach supports equivalence checking to prove the correctness of the result. Non-commercial and free solutions to generate SystemC from VHDL are *VHDLParser* by University of Tuebingen [5] and *VH2SC* by HT-Lab [6]. Both approaches consider mapping of the source VHDL to a limited set of SystemC constructs. These tools have demonstrated significant limitations, do not guarantee equivalence and they are not maintained anymore. VHDLParser dates to 2001 and addresses SystemC 1.0. Closed sources of the tools do not allow engineers to extend them to their needs. The most relevant approach is published by OFFIS [7]. It assumes creation of readable SystemC representations from VHDL that are targeted to be wrapped and simulated in the Simulink environment. There are known approaches for creating SystemC models from Verilog [8], [9] and tools targeting creation of other C++ subsets.

Cosimulation with SystemC has been of interest since long. ISS cosimulation is discussed first in [10]. Simulink cosimulation is discussed in [11]. Cycle-accurate and TLM cosimulation is described in [12].

III. VHDL TO SYSTEMC TRANSLATION METHODOLOGY

This section provides a methodology to translate RTL VHDL design to cycle-accurate SystemC model.

1. *Human readability*

First and the foremost is the consideration about the usage of the translated SystemC code. Whether the SystemC code is only to be fed to the compiler or is it going to be maintained by human developers. Many VHDL-to-SystemC converters lack this feature and generate an obscure code which is not fit for human consumption. It goes a long way to decide the human relationship with the generated SystemC code.

2. Similarity of the translated SystemC to VHDL

If a team of designers needs to maintain both the VHDL and SystemC code-bases, then it is appropriate to have a consistent view between the two. The translation mechanism must decide about this early on and take care of. Usage of the same module-names, variables, constructs, etc. must be adhered too, unless SystemC does not support a feature inherently.

3. SystemC module constructor on-the-fly

SystemC model needs a constructor, unlike a VHDL module. Some information, like sensitivity-list, is part of the SystemC constructor that is only available in VHDL process declaration, as shown in Table 1. Hence, SystemC constructor can only be written once complete VHDL code has been parsed. This demands that the information for the SystemC constructor be gathered while parsing the VHDL module, accumulated on-the-fly and finally written to the SystemC file.

Table 3. SystemC port method

VHDL syntax	*SystemC syntax*
`entity E is` `port (A: in std_logic);` `end E;` `architecture behav of E is` `signal X, Y: std_logic;` `begin` `X <= A;` `Y <= X;` `end behav;`	`class E_behav : public sc_module` `{` `sc_in<sc_logic> A;` `sc_logic X, Y;` `. . .` `X = A.read();` `Y = X;` `. . .` `};`

Table 4. SystemC process declarations

VHDL syntax	*SystemC syntax*
`P1: process (A,B)` `begin` `  . . .` `end P1;` `P2: process` `begin` `  . . .` `  wait . . .;` `  . . .` `end P2;`	`SC_METHOD(P1);` `sensitive << A << B;` `void P1(void){. . .}` `SC_THREAD(P2);` `void P2(void)` `{` `  . . .` `  wait(. . .);` `  . . .` `}`

4. Multiple architecture definitions

VHDL enables definition of multiple architectures for a single entity. However, SystemC is limited in this regards that there is only a single SystemC class available for each representation. A practical approach in this regard is to derive the name of each SystemC module from a combination of the VHDL entity and architecture names, e.g. by concatenation. Table 2 shows this approach.

5. *Using constructors rather than SC_CTOR*

VHDL allows model parameterization using generics. In order to achieve the same in SystemC, it is recommended to use the class constructors instead of SC_CTOR, enabling the VHDL generics to be used as constructor parameters.

6. Virtual destructor

As it is also discussed in the guideline 7 in Effective C++ [13], if the SystemC model has any virtual function then it should have a virtual destructor. Additionally, the classes not designed to be base classes or not designed to be used polymorphically should not declare virtual destructors.

7. Native C++ data-types for faster simulation

This is conventional wisdom to use C++ datatypes. Using lesser state types might suffice instead of higher state types. Take an example of VHDL std_logic type. If only values '0'/'1' are relevant, and not the 'X'/'Z' states, then C++ bool type is recommended instead of exact replacement by SystemC sc_logic type. However, this approach must be used with care. The entire code needs to be analyzed to make sure that the left-out values (e.g. 'X'/'Z' in this case) are not used anywhere in the code. Also, this restricts the SystemC model for later design updates when we might need to use the dropped values.

8. Operator precedence between VHDL and SystemC

This consideration is extremely important to keep up with. VHDL and SystemC have different precedence for certain operators. As is common, use parenthesis for clarity and overwriting precedence.

9. *Using port-methods for clarity*

SystemC uses the same operator '=' for reading the variables as well as the ports. To remove ambiguity, use methods for read/write on ports and to reserve '=' for variable assignments, as shown in Table 3.

Table 5. Single SC_METHOD

VHDL syntax	*SystemC syntax*
`architecture behav of E is` `begin` `  X <= A and B;` `  Y <= not A;` `  Z <= X or Y;` `end behav;`	`SC_METHOD(P1);` `sensitive << A << B << X << Y;` `void P1(void)` `{` `  X = A && B;` `  Y = !A;` `  Z = X || Y;` `}`

Table 6. Separate SC_METHOD's

VHDL syntax	*SystemC syntax*
`architecture behav of E is` `begin` `  X <= A and B;` `  Y <= not A;` `  Z <= X or Y;` `end behav;`	`SC_METHOD(P_X);` `sensitive << A << B;` `void P_X(void)` `{  X = A && B; }` `SC_METHOD(P_Y);` `sensitive << A;` `void P_Y(void)` `{  Y = !A; }` `SC_METHOD(P_Z);` `sensitive << X << Y;` `void P_Z(void)` `{  Z = X || Y; }`

Table 7. SystemC switch-case

VHDL syntax	*SystemC syntax*
entity E is port(SEL: in std_logic_vector(2 downto 1)); end E; architecture behav of E is begin case SEL is when "00" => . . . ; when "01" => . . . ; . . . end case; end behav;	class E_behav : public sc_module { sc_in<sc_lv<2> > SEL; }; void p(void) { int p_SEL = SEL.read().to_uint(); switch(p_SEL) { case 0:. . . ; break; case 1:. . . ; break; . . . } }

Table 8. SystemC if-then-else

VHDL syntax	*SystemC syntax*
entity E is port(VAL: in integer range 0 to 100); end E; architecture behav of E is begin case VAL is when 0 to 10 => . . . ; when 11 to 25 => . . . ; . . . end case; end behav;	class E_behav : public sc_module { sc_in<int> VAL; }; void p(void) { int p_VAL = VAL.read(); if ((0 <= p_VAL) && (p_VAL <= 10)) {. . .} else if((11 <= p_VAL) && (p_VAL <= 25)) {. . .} . . . }

10. *VHDL and SystemC port types*

Both VHDL and SystemC have similar types of ports and can be mapped uniquely from VHDL to SystemC. VHDL has a unique 'buffer' port type, an out-port whose value can be read inside the entity. For all practical purposes this can be mapped to SystemC 'sc_inout' port.

11. Translating VHDL process

VHDL process is used to implement sequential behavior. SystemC allows either SC_METHOD or SC_THREAD for such purpose. Sensitivity of the VHDL-process becomes the sensitivity of SystemC translation. Since an SC_METHOD cannot have wait statement, any VHDL process with a wait statement should be implemented as SC_THREAD, otherwise SC_METHOD might be preferred, as in Table 4.

12. Concurrent statements in VHDL

A sequential statement within a VHDL process goes inside a similar SC_METHOD or SC_THREAD in SystemC, but what about the concurrent statements outside of any VHDL process? There are 2 ways to handle such statements in SystemC.

a) Use single SC_METHOD for all concurrent statements, and sensitive to all the source (right-hand-side) variables in these statements, as shown in Table 5. This approach is easy to implement but the SC_METHOD executes whenever any variable changes, affecting the simulation performance.

b) Separate SC_METHOD for each statement, being sensitive to only this statement's source variable, as shown in Table 6.

The approach adds to the translation effort as well as increases the code-size but only single statement is executed each time, reducing the simulation overhead.

13. Handling clock-edge sensitivity

VHDL and SystemC uses varying notations to describe clock-edge sensitivity. Efficient and recommended, but elaborate, approach is to analyze the VHDL implementation of the process to determine the clock-sensitivity of interest, and then use it in SystemC. Such scheme is better for an event-based simulator like SystemC.

14. Switch-case translation

VHDL allows variables, logic-types as well ports in the switch-case construct, whereas SystemC allows only integer variables. Hence, VHDL switch literal must be converted to an integer, as shown in Table 7. Another possibility is to use if-then-else constructs, as shown in Table 8. This is particularly useful where the VHDL switch literal

- uses a range of values
- Cannot be converted to an integer, e.g. String-types.

15. SystemC process at start-up

SystemC scheduler has an interesting feature that each process (SC_METHOD or SC_THREAD) is always invoked once at the start-of-simulation. This happens even in absence of any event! Use dont_initialize() to stop this default invocation.

16. Executing the SystemC model constructor

SystemC has a unique requirement to execute the constructor of each module instance. This is unlike VHDL as there is no constructor for a VHDL design.

IV. Translation Validation

In order to validate the translation methodology described in the previous section, we have manually translated Plasma RISC-core[20] from VHDL to SystemC.

Plasma is a MIPS-ISA compatible, 32-bit synthesizable RISC core, implemented in VHDL. It is available as open-source, and used for various research projects, like this paper. Using translation rules described earlier, Plasma-core in VHDL has been successfully translated manually to SystemC.

The translated cycle-accurate SystemC module was successfully co-simulated in the Mentor Graphics ModelSim environment with the original testbench accompanying Plasma source-code. The cosimulation produced an equivalent system behavior, proving that the VHDL-to-SystemC translation methodology is applicable even to complex design, like a processor-core.

V. Cosimulation Performance Analysis

After developing the methodology for translating the VHDL to SystemC model, this section analyzes the performance of cosimulating VHDL and SystemC models. The aim of this analysis is to find possible optimizations in VHDL and SystemC models, as well as to recommend an effective co-simulation approach. VHDL design can be developed at RTL and behavioral levels of abstraction. Similarly, SystemC model can be developed at RTL and TLM abstraction-levels. In order to have reliable results, RTL level is employed for both VHDL and SystemC models.

Cosimulation plays an important role in SoC system-level-design (SLD)[14]. Architects initialize system-design at a higher abstraction level, e.g. Matlab, C, etc, then refine each component in a step-wise manner, while cosimulating with the rest of the system still at higher level. HW designers cosimulate their designs under development in a low level language e.g. VHDL/Verilog while the rest of the test environment, e.g. CPU, memory, bus, etc., is still in higher level language. SW developers typically cosimulate the instruction-set-simulator (ISS) at a higher level while the HW for which driver is being developed is simulated at a lower level. Hence co-simulation is important all over the SoC design-cycle.

Table 9. Plasma VHDL simulation-profile

Module	Simulation time (%)	Remarks
ALU	31.4	Highest sim-time
Register-bank	0.9	Lowest sim-time

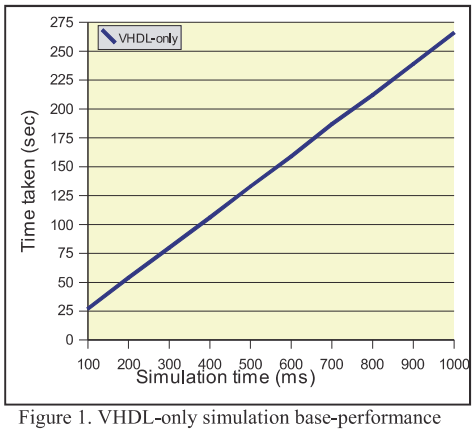


Figure 1. VHDL-only simulation base-performance

Simulation profiling: Cosimulation performance-analysis begins with profiling the Plasma VHDL simulation. Profiling the simulation highlights the simulation-time, and its %-age, taken by each VHDL module. For our analysis, we have considered the module taking the overall highest simulation-time and the module taking the lowest simulation-time. It is observed from Table-9 that the ALU module takes 31.4% simulation-time, whereas register-bank takes only 0.9% simulation-times. Translating these 2 modules to SystemC will give interesting results for co-simulation performance-analysis.

Base performance analysis: VHDL-only simulation is taken as the base for performance analysis. The term VHDL-only implies the original Plasma-core in VHDL, without any module in SystemC for co-simulation. The base performance will be used to compare against the cosimulation performance. Figure-1 shows the actual time-taken for varying simulation-times. As shown in Figue-1, the time-taken varies linearly with the simulation-time. This is a desired characteristic for

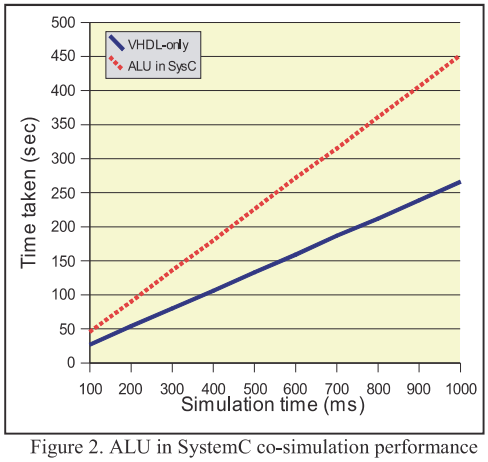


Figure 2. ALU in SystemC co-simulation performance

performance analysis as it ensures reliable and consistent results for varying simulation times.

Co-simulating ALU in SystemC: As the ALU module is profiled in Table-9 taking the highest %-age of simulation-time, it is a good candidate for co-simulation performance analysis. The ALU module is translated to SystemC, replacing the ALU module in VHDL, and co-simulated with the rest of the Plasma-core still in VHDL. Figure-2 shows the actual time-taken against the simulation-time. As expected, the variation of time-taken is linear. For comparison, also shown in Figure-2 is the base-performance of the VHDL-only simulation. The co-simulation with ALU is taking more time than the VHDL-only simulation. The reason for lower performance of co-simulation is that both VHDL and SystemC simulation- engines are now being invoked to perform the co-simulation. This leads to the increase in time-taken for co-simulation scenario.

Co-simulating register-bank in SystemC: The least %-age of simulation-time is spent in the register-bank, as shown by the profiling data. Hence, the register-bank is considered next for co-simulation performance-analysis. As earlier, the co-simulation setup consists of the register-bank translated to SystemC while the rest of the Plasma design is in VHDL. The ALU module is also reverted back to VHDL from SystemC. Figure-3 shows the actual time-taken against the simulation-time, for both register-bank in SystemC as well as the base-performance of the VHDL-only simulation. As in earlier cases, time-taken varies linearly with the simulation-time.

Significant and interesting observation in this case is that the register-bank in SystemC has better performance than the VHDL-only base-performance. This is against the earlier ALU-case which had lower performance than the base-performance. The reason for better co-simulation performance of register-bank in SystemC is due to the event-based design of the SystemC kernel. Event-based design invokes the SystemC kernel only when there is an event to execute the register-bank functionality. As earlier shown in Table-9, the register-bank takes negligible simulation-time, implying that there are almost no events for the register-bank module. Hence the SystemC kernel is almost not invoked in this case, leading to a better performance than the VHDL-only scenario.

VI. Conclusions and Future Work

With the acceptance of SystemC in the SoC design, IP providers are hard pressed to provide SystemC views of their legacy IPs. This paper has presented a methodology for translating RTL VHDL IPs to cycle-accurate SystemC models. The methodology ensures the translated SystemC code to be human readable with close correspondence to the source VHDL for easy maintainability. The methodology has been successfully employed to translate the Plasma-core from VHDL-RTL to cycle-accurate SystemC model, and verified by original VHDL testbench accompanying Plasma.

Performance-analysis of cosimulating VHDL and SystemC exhibits interesting and relevant results. The performance-analysis is based on the simulation-profile that highlights the %-age time taken by each module. The base-performance is obtained by profiling original VHDL-only

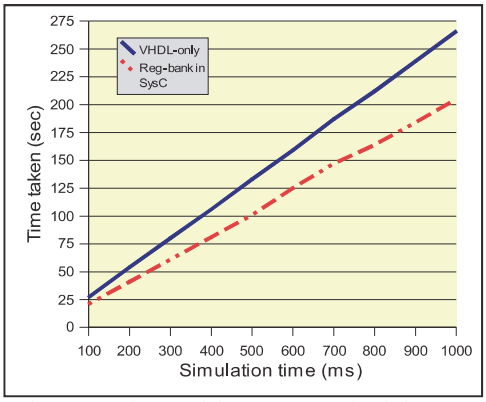


Figure 3. Register-bank in SystemC co-simulation

simulation. Cosimulating with the SystemC implementation of the module taking the highest simulation-time lowers the simulation-performance. On the other hand, it is observed that the cosimulation with the module taking least simulation-time is 30% faster than VHDL-only simulation!

The cosimulation performance results obtained in the paper are relevant to a wider SystemC community, including designers and architects. Furthermore, the results can be used for optimizing cosimulation tools as well system-level-models.

Future work aims at automating the complete VHDL translation to SystemC as well as explores a methodology to raise the abstraction-level from SystemC RTL to TLM.

References

[1] IEEE 1666 Standard SystemC Language Reference Manual, 2011.
[2] Plasma Overview, http://opencores.org/project,plasma
[3] Model studio, Carbon Design Systems, http://carbondesignsystems.com
[4] HIFSuite, EDA-Lab, http://hifsuite.edalab.it, 2012
[5] VHDL-to-SystemC-Converter, Eberhard-Karls-University of Tübingen, http://www-ti.informatik.uni-tuebingen.de/~systemc/, 2012
[6] VH2SC, HT-Lab, http://www.ht-lab.com, 2012
[7] R. Görgen, J.H. Oetjens, W. Nebel, "Automatic integration of hardware descriptions into system-level models," Proc. of IEEE DDECS 2012.
[8] W. Snyder, Verilator, Veripool, http://www.veripool.org/wiki/verilator
[9] Edwin Naroska, "FreeHDL," http://www.freehdl.seul.org/, 2012
[10] Benini, L. ; Bertozzi, D. ; Bruni, D. ; Drago, N. ; Fummi, F. ; Poncino, M. "Legacy SystemC co-simulation of multi-processor systems-on-chip," IEEE Int'l Conf Computer Design, 2002.
[11] Mendoza, F.; Kollner, C.; Becker, J.; Muller-Glaser, K.D, "An automated approach to SystemC/Simulink co-simulation," IEEE Int'l Symp. System Prototyping (RSP), 2011
[12] "Uniform SystemC Co-Simulation Methodology for System-on-Chip Designs," IEEE CyberC, 2012.
[13] Scott Meyers, "Effective C++ Digital Collection: 140 Ways to Improve Your Programming," 2012
[14] Brian Bailey, Grant Martin, "ESL Models and their Application: Electronic System Level Design and Verification in Practice (Embedded Systems)," 2012.

Appendix E

Research paper [E]

Syed, Saif Abrar; Jenihhin, Maksim; Raik, Jaan. “Optimization Methodologies for Cycle-Accurate SystemC Models Converted from RTL VHDL.” IP-SoC 2013 (IP Based Electronic System Conference), Grenoble, France, 2013. pp 1-6.

IP-SOC 2013

OPTIMIZATION METHODOLOGIES FOR CYCLE-ACCURATE SYSTEMC MODELS CONVERTED FROM RTL VHDL

Syed Saif Abrar[1,2], Maksim Jenihhin[1], Jaan Raik[1]
[1]{ *saif* | *maksim* | *jaan*}*@ati.ttu.ee* , Tallinn University of Technology, ESTONIA
[2]*saif.abrar@in.ibm.com* , IBM, India

Abstract:

IP design-houses are hard-pressed by their customers to provide SystemC models of their portfolio IPs, despite already existing VHDL views. VHDL IPs can be translated to SystemC, ensuring correctness, quality and maintainability of the translated code. VHDL and SystemC are frequently co-simulated by architects as well as verification teams. This paper explores optimization scenarios that affect the cosimulation performance, resulting in 30% faster co-simulation. In addition to the plain VHDL-to-SystemC conversion, there are possibilities of alternate implementations for a SystemC model. This paper explores these alternate scenarios to get 25% better simulation-speed. The optimization methodologies in this paper are relevant to architects, designers, verification-teams, IP design-houses that need to provide high-speed simulation-models, and can be used for optimizing cosimulation tools as well system-level models.

Keywords: VHDL, SystemC, conversion, optimization

1. INTRODUCTION

Consumer electronic devices are becoming increasingly complex, presenting lots of hardware and software design challenges. The traditional design approaches at Register Transfer Level (RTL) using VHDL or Verilog are no longer suitable, resulting in the search for a newer abstraction-level that enables unified development of the System-on-a-Chip (SoC) designs. This level has been called *system level*, *behavioral level*, *C-level*, *algorithmic level*, *Electronic System Level (ESL)*, etc.

Electronic System Level (ESL) design is an emerging methodology to model the (sub-)system at a high-level of abstraction. Transaction Level Modeling (TLM) is the widely recognized and adopted approach to realize the ESL in practice. TLM models are applicable for a range of design-tasks, e.g. early software development, performance analysis, architecture exploration, HW/SW partitioning, etc. A big pain-point for the IP design-houses is the challenge of providing the TLM models of their IPs. Their customers are increasingly asking for SystemC models [1] along with the RTL description. Developing the SystemC-models manually has a potential of mismatch between RTL and SystemC, and is impractical in terms of time and effort. Hence, there is an urgent need of automated generation of SystemC models from existing RTL description.

The rest of the paper is organized as follows. Section 2 gives an overview of the related work. Section 3 describes the VHDL-to-SystemC translation methodology. Section 4 details the experiments conducted for simulation-speed optimization. Finally, section 5 concludes the paper along with future work.

2. RELATED WORK

Carbon Model Studio [3] is a commercial offering that allows creating configurable SystemC models from RTL VHDL or Verilog descriptions. It does not intend to create human-readable output. HIFSuite [4], [5] is a design and verification framework addressing manipulation and integration of heterogeneous design parts. The output result also does not consider human-readability and correspondence to the source VHDL. The approach supports equivalence checking to prove the correctness of the result.

Example of non-commercial and free solutions to generate SystemC from VDHL are VHDLParser by University of Tuebingen [6] and VH2SC by HT-Lab [7]. Both approaches consider mapping of the source VHDL to a limited set of SystemC constructs. These tools have demonstrated significant limitations, do not guarantee equivalence and they are not maintained anymore. VHDLParser dates to 2001 and addresses SystemC 1.0. Closed sources of the tools do not allow engineers to extend them to their needs.

The most relevant approach is published by OFFIS in [8]. It assumes creation of readable SystemC representations from VHDL that are targeted to be wrapped and simulated in the Simulink environment. The approach is claimed to support industrial designs, however only an illustrative example details are available [9]. This practical work also does not provide for equivalence checking mechanisms or results. There are known approaches for creating SystemC models from Verilog [10] and tools targeting creation of other C++ subsets [11], [12].

Cosimulation with SystemC has been of interest since long. Instruction-set-simulator (ISS) cosimulation is discussed first in [16]. Simulink cosimulation is discussed in [17].

VHDL

```
01  entity b09 is port (
02   reset,clock: in bit;
03   <snip>
04   y: out bit
05  );
06  end b09;
07
08  architecture BEHAV of b09 is
09   constant Bit_start : bit := '1';
10   <snip>
11   constant Zero_8 : bit_vector ( 7 downto 0)   :=
12  "00000000";
13   <snip>
14   signal d_in: bit_vector ( 8 downto 0);
15   <snip>
16
17  begin
18   process(clock,reset)
19    variable stato: integer range 3 downto 0;
20    <snip>
21   begin
22    if    reset = '1' then
23    stato := INIT;
24    <snip>
25    elsif clock'event and clock='1' then
26     case stato is
27     when INIT
28      =>
29       stato := RECEIVE;
30       <snip>
31     when LOAD_OLD
32      =>
33       if  d_in(0) = Bit_start
34       then
35        if  d_in(8 downto 1) = old
36        then
37         d_in <= Zero_9;
38         <snip>
39        else
40         <snip>
41        end if;
42         <snip>
43       else
44        d_in <= x & d_in(8 downto 1);
45       end if;
46      end case;
47     end if;
48   end process;
    end BEHAV;
```

SystemC

```
01    class sc_WORK_B09_BEHAV : public sc_module
02    {
03     public:
04     sc_WORK_B09_BEHAV(sc_module_name mn_);
05     sc_in < bool > RESET;
06     <snip>
07     sc_out < bool > Y;
08     private:
09      sc_signal< sc_bv < 9 >  > D_IN;
10      void process_line_23(void);
11      <snip>
12    };
13    const bool BIT_START = 1;
14    <snip>
15    const sc_bv  < 8 >  ZERO_8 = "00000000";
16
17    sc_WORK_B09_BEHAV ::
18    sc_WORK_B09_BEHAV(sc_module_name mn_)
19    : sc_module(mn_)
20    {
21     SC_METHOD(process_line_23);
22     sensitive  << CLOCK << RESET;
23     dont_initialize();
24    }
25    void sc_WORK_B09_BEHAV::process_line_23(void)
26    {
27     sc_int < 4 > STATO;
28     <snip>
29    if ( RESET.read() == 1 )
30    {
31     STATO = INIT;
32     <snip>
33    }
34    else
35    {
36     if ( (CLOCK.read() == 1)  )
37     {
38      if ( STATO == INIT )
39      {
40       STATO = RECEIVE;
41       <snip>
42      }
43      if ( STATO == LOAD_OLD )
44      {
45       if (D_IN.read().bit(0).to_bool()==BIT_START)
46       {
47        if ( D_IN.read().range(8,1) == OLD.read() )
48        {
49         D_IN = ZERO_9;
50          <snip>
51        }
52        else
53        {
54          <snip>
55        }
56       }
57       else
58       {
59        D_IN=concat(X.read(),D_IN.read().range(8,1));
60        <snip>
61       }
62      }
63     }
64    }
65    } //end sc_WORK_B09_BEHAV::process_line_23()
```

Figure. 1. ITC b09 translated from VHDL to SystemC using the proposed approach

3. VHDL-TO-SYSTEMC TRANSLATION

VHDL to SystemC translation follows a set a guidelines discussed in [18]. Without any loss of generality, Fig. 1 shows the ITC99 benchmark design *b09* translated from VHDL to SystemC using our methodology. Only the code of interest is shown in Fig. 1, and the omitted code is marked with a <*snip*> tag. Translation methodology is described below by set of rules, referring to the Line-numbers in the SystemC column to aid the explanations.

A. Handling multiple architecture definitions

VHDL enables multiple architectures for a single entity. The methodology names a SystemC module by concatenating VHDL entity and architecture names, as in Line-01.

B. Using constructors rather than SC_CTOR

VHDL allows model parametrization using generics. The methodology uses class constructor, as shown in Line-04, instead of SC_CTOR, to use parameters as VHDL generics.

C. Virtual Destructors

As it is also discussed in the guideline 7 in Effective C++, if the SystemC model has any virtual function then it should have a virtual destructor.

D. Mapping VHDL and SystemC port types

Both VHDL and SystemC have similar types of ports. As in Line-05 and Line-07, sc_in and sc_out for VHDL in and out ports respectively. Additionally, VHDL *buffer* port can be mapped to *sc_inout* port.

E. Naming the SystemC process

SystemC process names are required, whereas optional in VHDL. The methodology uses VHDL process name, if it exists, or derives it from the process line-number, as shown in Line-10.

F. Exploiting native C++ data-types for faster simulation

This is conventional wisdom to use C++ datatypes. As shown in Line-13, *bool* type is used instead of SystemC defined SC_*bit* data type. However, this approach must be used with care, to make sure that the left-out values (e.g. 'X'/'Z' in this case) are not used anywhere in the code.

G. Writing SystemC module constructor on-the-fly

SystemC module has a constructor with information, like sensitivity list, that is spread across VHDL code. Information for the SystemC constructor has to be gathered while parsing the VHDL module, accumulated on-the-fly and finally written to the SystemC file. This is shown in Line-20.

H. Translating the VHDL process

SystemC has SC_METHOD and SC_THREAD, similar to a VHDL process. A VHDL process with a wait statement is implemented as SC_THREAD, otherwise SC_METHOD might be preferred. As shown in Line-21, SC_METHOD is used, and the sensitivity of the VHDL-process becomes the sensitivity of SystemC translation, as shown in Line-22.

I. Preventing invocation of SystemC process at start-up

SystemC scheduler invokes every process (SC_METHOD or SC_THREAD) at the start of simulation, even in absence of any event! Use dont_initialize() to stop this default invocation, as shown in Line-23.

J. Using port-methods for clarity

SystemC uses same operator '=' for variables and ports. To remove ambiguity in understanding the translated source-code, use .read() and .write() methods for ports and '=' for variable assignments. As in Line-29, port is read using its .read() method.

K. Handling clock-edge sensitivity

VHDL and SystemC use varying notations to describe clock edge sensitivity. Line-36 shows the translation for positive-edge clock sensitivity. But the process is invoked on both the edges of clock. Efficient and recommended, but elaborate approach is to analyze the VHDL to determine the clock sensitivity of interest, and then use it in SystemC. Such scheme is better for an event-based simulator like SystemC.

L. Translating switch-cases

VHDL allows variables, logic-types as well as ports in the switch-case construct, whereas SystemC allows only integers. Hence, VHDL switch literal must be converted to an integer, if possible. Another approach is to use if-then-else constructs, as shown in Line-38 and Line-43, where the VHDL switch literal

- uses a range of values
- cannot be converted to an integer, e.g. strings

M. Chain of SystemC library calls

Sometimes it is necessary in practice to employ a chain of SystemC library calls to achieve the desired behavior, e.g. comparing a single-bit value from a port, as shown in Line-45.

In addition to the above details highlighted in Fig. 1, few other rules are followed for translation, described below:

N. Operator precedence differences

This consideration is extremely important to follow. VHDL and SystemC have different precedence for certain operators. As is common, use parenthesis for clarity and overwriting precedence.

O. Handling concurrent VHDL statements

Concurrent VHDL statements can be handled as:

a) Single SC_METHOD for all concurrent statements, sensitive to all the source RHS operations in these statements. Drawback is that all the statements are executed whenever any RHS operation changes, affecting the simulation performance.

b) Separate SC_METHOD for each statement, sensitive to only this statement's source variable. This reduces the simulation overhead, but increases the code-size.

P. Executing the SystemC model constructor

SystemC models instantiated inside a top-level module need their constructors to be executed, e.g. implementing a 4-bit adder from 4 instances of 1-bit adder.

SystemC code generated by the zamiaCAD implementation has following highly desirable qualities:

Human readability: This is an important consideration about the usage of the generated SystemC code: whether the SystemC code is only to be fed to a compiler or is it going to be maintained by human developers. Many VHDL to SystemC converters lack this feature and generate an obscure code which is not fit for human use. It goes a long way to decide the human relationship with the generated SystemC code.

Correspondence of the translated SystemC to VHDL: If a team of designers needs to maintain both the VHDL and SystemC code-bases, then it is appropriate to have a consistent view between the two. The translation mechanism must decide about this early on and take care of. Usage of the same module names, variables, constructs, etc. must be adhered too, unless SystemC does not support a feature inherently.

Table 1. Plasma VHDL simulation-profile

Module	Simulation time (%)	Remarks
ALU	31.4	Highest sim-time
Register-bank	0.9	Lowest sim-time

4. OPTIMIZATIONS FOR SIMULATION-SPEED

The SystemC-model available from the previous section is obtained by converting the VHDL model on a line-by-line basis. This plain translation approach does not take into account the simulation characteristics as well as implementation style of the SystemC model. This section explores various implementation aspects to improve the simulation-speed and discover potential optimization possibilities.

A. Optimization for Co-simulation

This section analyzes the performance of co-simulating VHDL and SystemC models, as in [19]. The aim of this analysis is to find possible optimizations in VHDL and SystemC models, as well as to recommend an effective co-simulation approach.

Cosimulation plays an important role in SoC system-level-design (SLD)[2]. Architects initialize system-design at a higher abstraction level, e.g. Matlab, C, etc, then refine each component in a step-wise manner, while cosimulating with the rest of the system still at higher level. HW designers cosimulate their designs under development in a low level language e.g. VHDL/Verilog while the rest of the test environment, e.g. CPU, memory, bus, etc., is still in higher level language. SW developers typically cosimulate the instruction-set-simulator (ISS) at a higher level while the HW for which driver is being developed is simulated at a lower level. Hence co-simulation is important all over the SoC design-cycle.

Simulation profiling: Cosimulation performance-analysis begins with profiling the Plasma VHDL simulation, to get the simulation-time and its %-age taken by each VHDL module. Table-1 shows that the ALU module takes the highest (31.4%) simulation-time, whereas register-bank takes the lowest (0.9%) simulation-time.

Base performance analysis: VHDL-only simulation is taken as the base for performance analysis. The term VHDL-only implies the original Plasma-core in VHDL, without any module in SystemC for co-simulation. The base performance will be used to compare against the cosimulation performance. As shown in Figure-2, the time-taken varies linearly with the simulation-time, ensuring reliable and consistent results for varying simulation times.

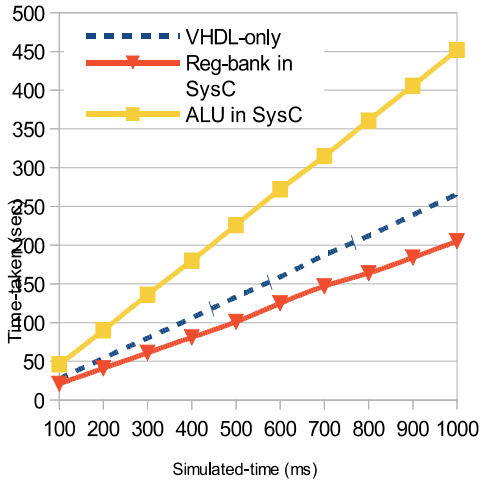


Figure-2. VHDL-SystemC co-simulation analysis

Co-simulating ALU in SystemC: As the ALU module is profiled in Table-1 taking the highest %-age of simulation-time, it is a good candidate for co-simulation performance analysis. The ALU module is translated to SystemC, replacing the ALU module in VHDL, and co-simulated with the rest of the Plasma-core still in VHDL. Figure-2 shows the actual time-taken against the simulation-time. As expected, the variation of time-taken is linear. The co-simulation with ALU is taking more time than the VHDL-only simulation. The reason for lower performance of co-simulation is that both VHDL and SystemC simulation- engines are now being invoked to perform the co-simulation, increasing the simulation time.

Co-simulating register-bank in SystemC: The least %-age of simulation-time is spent in the register-bank. Hence, it is considered next for co-simulation performance-analysis. As earlier, the co-simulation setup consists of the register-bank translated to SystemC while the rest of the Plasma design is in VHDL. The ALU module is also reverted back to VHDL from SystemC. Figure-2 shows the actual time-taken against the simulation-time, for both register-bank in SystemC as well as in VHDL.

Significant and interesting observation in this case is that the register-bank in SystemC has better performance than the VHDL-only base-performance.

Table 2. Single SC_METHOD

VHDL syntax	SystemC syntax
architecture behav of E is begin X <= A and B; Y <= not A; Z <= X or Y; end behav;	SC_METHOD(P1); sensitive << A << B << X << Y; void P1(void) { X = A && B; Y = !A; Z = X || Y; }

Table 3. Separate SC_METHOD's

VHDL syntax	SystemC syntax
architecture behav of E is begin X <= A and B; Y <= not A; Z <= X or Y; end behav;	SC_METHOD(P_X); sensitive << A << B; void P_X(void) { X = A && B; } SC_METHOD(P_Y); sensitive << A; void P_Y(void) { Y = !A; } SC_METHOD(P_Z); sensitive << X << Y; void P_Z(void) { Z = X || Y; }

The reason for better co-simulation performance of register-bank in SystemC is due to the event-based design of the SystemC kernel. Event-based design invokes the SystemC kernel only when there is an event to execute the register-bank functionality. As earlier shown in Table-1, the register-bank takes negligible simulation-time, implying that there are almost no events for the register-bank module. Hence the SystemC kernel is almost not invoked in this case, leading to a better performance.

B. Optimization for Combinational-statements

Combinational statements in VHDL are implemented outside of any VHDL process and are executed concurrently, in parallel. SystemC does not have this feature of concurrent statements, as all statements are executed within a SystemC process, either SC_METHOD or SC_THREAD. Since a VHDL combinational method does not have a 'wait' feature, only SC_METHOD is to be considered for converting combinational statements to SystemC.

Multiple combinational VHDL statements can be converted to SystemC SC_METHOD in the following 2 alternatives:

a) Single SC_METHOD for all combinational statements

This approach implements all VHDL combinational statements within a single SC_METHOD. The single SC_METHOD is made sensitive to all the source RHS elements in these statements. Table-2 shows an example of this approach. Advantage of this implementation is its simplicity and ease of implementation. Drawback is that all the statements in the SC_METHOD are executed even if only 1 RHS element changes.

b) Separate SC_METHOD for each combinational statement

In this approach, each combinational statement is implemented in a separate SC_METHOD. This SC_METHOD is made sensitive only to the RHS elements appearing in source VHDL statement. Table-3 shows an example of this approach. Advantage of this approach is that only the relevant SystemC statements are executed, within a particular SC_METHOD. Limitation is the increase in the source-code size due to an SC_METHOD implemented for each combinational statement.

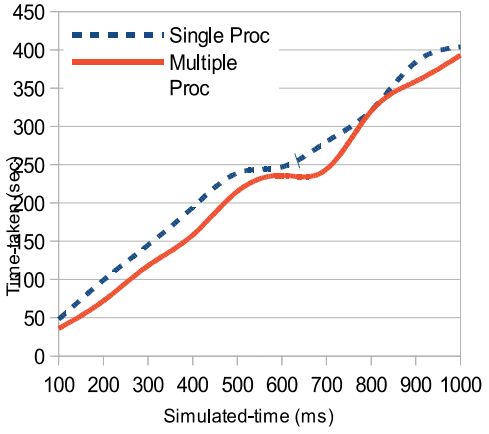


Figure-3. Simulation-time for single and multiple processes

Results:

To experiment the 2 alternative methods of translating combinational-statements in VHDL to SystemC, the 'shifter' module in the Plasma core is selected. The 'shifter' module has 12 combinational-statements, making it a good candidate to observe the differences in simulation-time. As is seen from the Figure-3, the multiple process approach takes about 25% lesser simulation-time, resulting in faster simulation-speed. This can be attributed to the fact that in multiple-process implementation, only a single statement is executed, resulting in optimized simulation-model.

C. Optimization for Events

Events are used in SystemC to synchronize actions among processes. A SystemC process can wait for multiple events, suspending its task while in wait-state. Another process triggers the event at some time during simulation. The waiting process wakes-up when it receives the notification of event-trigger and resumes its task. A process can be made to wait on a single-event or on multiple-events, as shown in Table-4. On the other hand, the event notification can be done immediately or in the next delta-cycle, as shown in Table-5.

Results:

The modules ALU, DDR, DMA and MUX from the Plasma-core are implemented with the earlier alternatives; e.g. ALU implementation is referred to as:

- ALU SE: Single-event waiting, immediate notification

Table 4. Events-waiting in SystemC

Single-event waiting	Multiple-event waiting
`SC_THREAD(th_single_ev);` `void th_single_ev(void)` `{` `  wait(ev_single);` `  statement-1;` `  statement-2;` `  ...` `}`	`SC_THREAD(th_mult_ev);` `void th_mult_ev(void)` `{` `  wait(ev_mult_1);` `  statement-1;` `  wait(ev_mult_2);` `  statement-2;` `  ...` `}`

Table 5. Events-notification in SystemC

Immediate notification	Delta-cycle notification
`sc_event ev_do;` `void pr_imm()` `{` `  statement-1;` `  . . .` `  ev_do.notify();` `  . . .` `}`	`sc_event ev_do;` `void pr_delta()` `{` `  statement-1;` `  . . .` `ev_do.notify(SC_ZERO_TIME)` `;` `  . . .` `}`

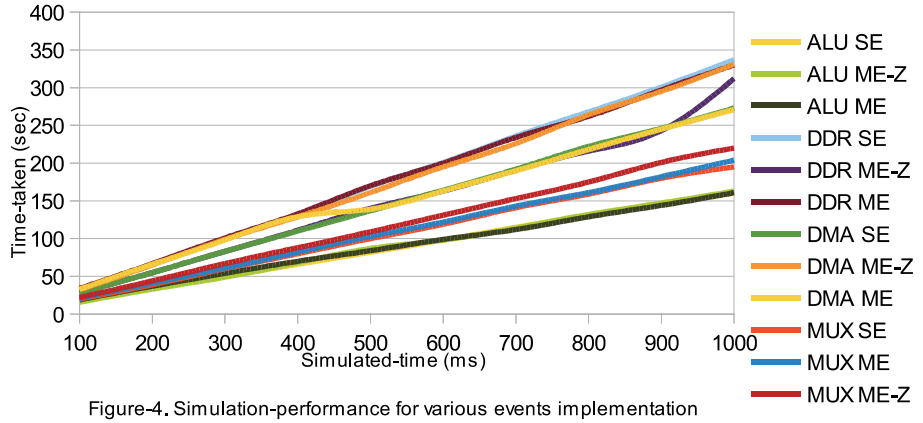


Figure-4. Simulation-performance for various events implementation

- ALU ME: Multiple-event waiting, immediate notification
- ALU ME-Z: Multiple-event waiting, delta-cycle notification

Figure-4 shows the simulation-performance for ALU, DDR, DMA and MUX modules, for the 3 alternatives. As seen in Figure-4, there is no significant variation in the simulation-performance for a given module.

The single-event implementation is simpler in terms of coding, understanding and debugging. Hence, this might be a preferred implementation alternative, though based on coding-aspect rather than the simulation-speed.

5. CONCLUSIONS

This research has discussed a methodology for translating RTL VHDL IPs to cycle-accurate SystemC models, and experimented with various optimization methodologies for the generated SystemC model. (1) Performance-analysis of co-simulating VHDL with SystemC-module taking the highest simulation-time lowers the simulation-performance. On the other hand, co-simulation with the module taking least simulation-time is 30% faster than VHDL-only simulation. (2) Implementation of VHDL combinational-statements using multiple SystemC-processes takes about 25% lesser simulation-time than implementation with a single SystemC-process. (3) SystemC synchronization strategies using single and multiple events take similar simulation-time, although single-event implementation is easier to develop and maintain. The optimization results obtained in this paper are relevant to a wide SystemC community, including architects, designers, verifiers as well as IP design-houses and EDA-vendors.

ACKNOWLEDGEMENTS

The work has been supported in part by the Estonian ICT project FUSETEST, by CEBE through the European Structural Funds, by Estonian SF grants 8478 and 9429 and by the Tiger University Program of the Information Technology Foundation for Education (HITSA).

REFERENCES

[1] IEEE 1666 Standard SystemC Language Reference Manual, 2011.

[2] ESL Models and their Application: Electronic System Level Design and Verification in Practice(Embedded Systems) B Bailey, G Martin. 2012.

[3] Carbon Design Systems, http://carbondesignsystems.com

[4] HIFSuite, EDA-Lab, http://hifsuite.edalab.it, 2012

[5] N. Bombieri, M. Ferrari, F. Fummi, et al., "HIFSuite: Tools for HDL Code Conversion and Manipulation," in EURASIP Journal on Embedded Systems, vol. 1155, no. 10, 2010.

[6] VHDL-to-SystemC-Converter, Eberhard-Karls-University of Tübingen, http://www-ti.informatik.uni-tuebingen.de/~systemc/, 2012

[7] VH2SC, HT-Lab, http://www.ht-lab.com, 2012

[8] R. Görgen, J.H. Oetjens, W. Nebel, Automatic integration of hardware descriptions into system-level models, Proc. of IEEE DDECS 2012, Tallinn, pp.105-110

[9] R. Görgen, "VHDL-to-SystemC Transformation," 2012, http://vhome.offis.de/ralphg/vhdl2sc.pdf

[10] W. Snyder, Verilator, Veripool, http://www.veripool.org/wiki/verilator

[11] Edwin Naroska, "FreeHDL," http://www.freehdl.seul.org/, 2012

[12] OStatic, "VHDLC," http://ostatic.com/vhdlc, 2012

[13] zamiaCAD Open-source HW Design Framework, http://zamiacad.sf.net

[14] Legacy SystemC co-simulation of multi-processor systems-on-chip. Benini, L. ; Bertozzi, D. ; Bruni, D. ; Drago, N. ; Fummi, F. ; Poncino, M. IEEE Int'l Conf Computer Design, 2002.

[15] An automated approach to SystemC/Simulink co-simulation. Mendoza, F.; Kollner, C.; Becker, J.; Muller-Glaser, K.D. IEEE Int'l Symp. System Prototyping (RSP), 2011

[16] Uniform SystemC Co-Simulation Methodology for System-on-Chip Designs. IEEE CyberC, 2012

[17] An automated approach to SystemC/Simulink co-simulation. Mendoza, F. ; Kollner, C. ; Becker, J. ; Muller-Glaser, K.D. IEEE Int'l Symp. On Rapid System Prototyping (RSP), 2011

[18] Extensible Open-Source Framework for Translating RTL VHDL IP Cores to SystemC, Syed Saif Abrar, Maksim Jenihhin, Jaan Raik; IEEE Int'l Symp. Design and Diagnostics of Electronic Circuits & Systems (DDECS), 2013

[19] Performance Analysis of Cosimulating Processor Core in VHDL and SystemC. Syed Saif Abrar, Shyam Kiran A., Maksim Jenihhin, Jaan Raik, C. Babu; IEEE Int'l Conf. Advances in Computing, Communications and Informatics (*ICACCI*), 2013.

Appendix F

Research paper [F]

Syed, Saif Abrar; Jenihhin, Maksim; Raik, Jaan. “Extensible Open-Source Framework for Translating RTL VHDL IP Cores to SystemC.” IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems, Karlovy Vary, Czech Republic, 2013. pp 1-4.

Extensible Open-Source Framework for Translating RTL VHDL IP Cores to SystemC

Syed Saif Abrar[1,2], Maksim Jenihhin[2], Jaan Raik[2]
[1]IBM, Bangalore, INDIA
[2]Tallinn University of Technology, Tallinn, ESTONIA
{saif | maksim | jaan}@ati.ttu.ee

***Abstract*— SystemC has gained wide acceptance in the design of VLSI SoCs. At the same time there exists a large number of legacy IP cores described in VHDL whose reuse and integration into SystemC ecosystem is highly demanded. However, there is a lack of any standard approach in this regard. This paper proposes an open-source framework and methodology to convert RTL VHDL IP cores to cycle-accurate SystemC designs. The SystemC output is emphasized to be human-readable and providing for clear correspondence to the source VHDL code, thus allowing further manual code changes and debug. The described framework has been implemented based on an open-source zamiaCAD platform and has been successfully applied to translate various VHDL benchmark designs.**

Keywords—VHDL, SystemC, RTL

I. INTRODUCTION

Consumer electronic devices are becoming increasingly complex, presenting lots of hardware and software design challenges. Hardware designers need performance evaluation to select proper architecture. Software developers need to develop and debug the software much before the availability of the physical hardware. The traditional design approaches at Register Transfer Level (RTL) using VHDL or Verilog are no longer suitable, resulting in the search for a newer abstraction-level that enables unified development of the System-on-a-Chip (SoC) designs. This level has been called *system level*, *behavioral level*, *C-level*, *algorithmic level*, *Electronic System Level (ESL)*, or just *a higher level* (i.e. higher than RTL).

Various languages and tools were proposed and tried out to enable different parts of the system level design. *Behavioral Verilog or VHDL*, *C*, *Java*, *C++ class libraries*, C derivatives like *SpecC* [1] and *HandelC* [2], *SystemVerilog*, and special-purpose languages like *Bluespec* [3] and *Esterel* [4]. Among these, *SystemC* has emerged as the dominant language and was standardized as IEEE-1666 [5].

Product development companies are focusing their methodologies around SystemC. Architects, hardware designers, software developers, etc. have started using SystemC for their project tasks. Either complete or partial systems are now being modeled in SystemC, proliferating its usage and acceptance. Thus, the IP providers are deeply pressed to make available the SystemC models of their legacy VHDL or Verilog IPs, to remain competitive and relevant in today's market. The manual conversion approach is error prone and extremely time consuming. A preferred way is to automate VHDL/Verilog conversion to SystemC, as this approach is faster and maintains consistency between source and SystemC implementations.

The abstraction level of the converted SystemC models can range from cycle-accurate RTL to untimed TLM (Transaction-Level Modeling). Availability of cycle-accurate SystemC model, instead of VHDL, has advantages across the product development, namely:

- Architects integrate and update their C/C++ algorithms directly in the hardware model
- Hardware designers, transitioning from VHDL to SystemC, learn relationship between VHDL and SystemC.
- Verification engineers directly use SystemC testbenches and can avoid expensive co-simulation tools
- Firmware developers working with assembly/C/C++ find familiar development environment for their tasks.

This paper presents a framework and methodology to convert a VHDL model to cycle-accurate SystemC, with the following benefits

- Extensible framework
- Open-source development
- Human-readable SystemC output
- Correspondence of the output to the source VHDL code

The methodology has been verified by converting various benchmark VHDL codes, as well as partially converting and co-simulating with a processor implementation in VHDL.

The rest of the paper is organized as follows. Section 2 gives an overview of the state-of-the-art. Section 3 describes the proposed methodology Sections 4 and 5 demonstrate experimental results and conclude the paper.

II. RELATED WORK

Among the commercial solutions there is *Carbon Model Studio* [6] that allows creating configurable SystemC models from RTL VHDL or Verilog descriptions. It is targeted mainly at simulation speedup and does not intend to create human-readable output. *HIFSuite* [7], [8] is a design and verification framework addressing manipulation and integration of heterogeneous design parts. Similarly, it allows dumping out RTL VHDL decryptions into SystemC. Before the dump out the design models can be manipulated on the internal HIF model. The output result also does not consider human-readability and correspondence to the source VHDL. The

VHDL

```vhdl
entity b09 is port (
 reset,clock: in bit;
 <snip>
 y: out bit
);
end b09;

architecture BEHAV of b09 is
 constant Bit_start : bit := '1';
 <snip>
 constant Zero_8 : bit_vector ( 7 downto 0)   := "00000000";
 <snip>
 signal d_in: bit_vector ( 8 downto 0);
 <snip>

begin
 process(clock,reset)
  variable stato: integer range 3 downto 0;
  <snip>
 begin
  if    reset = '1' then
  stato := INIT;
  <snip>
  elsif clock'event and clock='1' then
   case stato is
   when INIT
    =>
     stato := RECEIVE;
     <snip>
   when LOAD_OLD
    =>
     if  d_in(0) = Bit_start
     then
      if  d_in(8 downto 1) = old
      then
       d_in <= Zero_9;
       <snip>
      else
       <snip>
      end if;
       <snip>
     else
      d_in <= x & d_in(8 downto 1);
     end if;
    end case;
   end if;
 end process;
end BEHAV;
```

SystemC

```systemc
class sc_WORK_B09_BEHAV : public sc_module
{
 public:
 sc_WORK_B09_BEHAV(sc_module_name mn_);
 sc_in < bool > RESET;
 <snip>
 sc_out < bool > Y;
 private:
  sc_signal< sc_bv < 9 >  > D_IN;
  void process_line_23(void);
  <snip>
};
const bool BIT_START = 1;
<snip>
const sc_bv  < 8 >  ZERO_8 = "00000000";

sc_WORK_B09_BEHAV ::
sc_WORK_B09_BEHAV(sc_module_name mn_)
: sc_module(mn_)
{
 SC_METHOD(process_line_23);
 sensitive  << CLOCK << RESET;
 dont_initialize();
}
void sc_WORK_B09_BEHAV::process_line_23(void)
{
 sc_int < 4 > STATO;
 <snip>
if ( RESET.read() == 1 )
{
 STATO = INIT;
 <snip>
}
else
{
 if ( (CLOCK.read() == 1)  )
 {
  if ( STATO == INIT )
  {
   STATO = RECEIVE;
   <snip>
  }
  if ( STATO == LOAD_OLD )
  {
   if (D_IN.read().bit(0).to_bool() == BIT_START )
   {
    if ( D_IN.read().range(8,1) == OLD.read() )
    {
     D_IN = ZERO_9;
     <snip>
    }
    else
    {
     <snip>
    }
   }
   else
   {
    D_IN=concat(X.read(),D_IN.read().range(8,1));
    <snip>
   }
  }
 }
}
} //end sc_WORK_B09_BEHAV::process_line_23()
```

Fig. 1. ITC b09 translated from VHDL to SystemC using the proposed approach

approach supports equivalence checking to prove the correctness of the result. However, the major advantage of this tool is ability to raise the level of abstraction of the design from RTL to TLM. We had access only to the demo version of the tool in our experiments. We were able to find only the following two non-commercial and free solutions to generate SystemC from VDHL. *VHDLParser* by University of Tuebingen [9] and *VH2SC* by HT-Lab [10]. Both approaches consider mapping of the source VHDL to a limited set of SystemC constructs. These tools have demonstrated significant limitations, do not guarantee equivalence and they are not maintained anymore. VHDLParser dates to 2001 and addresses SystemC 1.0. Closed sources of the tools do not allow engineers to extend them to their needs.

The most relevant approach is published by OFFIS in [11]. It assumes creation of readable SystemC representations from VHDL that are targeted to be wrapped and simulated in the Simulink environment. The approach is claimed to support industrial designs, however only an illustrative example details are available [12]. This practical work also does not provide for equivalence checking mechanisms or results. There are known approaches for creating SystemC models from Verilog [13] and tools targeting creation of other C++ subsets [14], [15].

Different from the existing works we address an open-source extensible framework for fully automated generation of standard SystemC descriptions from legacy RTL VHDL IP cores targeting their reuse in industrial SystemC environments. The framework assumes future extension of the current implementation to support automated RTL to TLM abstraction.

III. VHDL TO SYSTEMC TRANSLATION METHODOLOGY

The proposed methodology has been implemented in zamiaCAD, a scalable model based open-source RTL design, verification and analysis platform for VHDL [16]. Current implementation exploits the platform infrastructure with the graphical user interface based on Eclipse IDE plugin. It exploits the abstract syntax tree (AST) generated by the platform's front-end while assuming usage of the semantically resolved scalable instantiation graph data structures [17] for complex design constructs. The initial considerations about the idea were presented in [18].

Without any loss of generality, Fig. 1 shows the ITC99 benchmark design *b09* translated from VHDL to SystemC using our methodology. For the sake of clarity and explanation, only the code of interest is shown in Fig. 1, and the omitted code is marked with a <*snip*> tag. Translation methodology is described below by set of rules, referring to the Line-numbers in the SystemC column to aid the explanations.

1) Handling multiple architecture definitions

VHDL enables multiple architectures for a single entity. The methodology names a SystemC module by concatenating VHDL entity and architecture names, as in Line-01.

2) Using constructors rather than SC_CTOR

VHDL allows model parameterization using generics. The methodology uses class constructor, as shown in Line-04, instead of SC_CTOR, to use parameters as VHDL generics.

3) Mapping VHDL and SystemC port types

Both VHDL and SystemC have similar types of ports. As in Line-05 and Line-07, `sc_in` and `sc_out` for VHDL in and out ports respectively. Additionally, VHDL *buffer* port type can be mapped to SystemC *sc_inout* port.

4) Naming the SystemC process

SystemC process names are required, whereas optional in VHDL. The methodology uses VHDL process name, if there, or derives from the process line-number, as shown in Line-10.

5) Exploiting native C++ data-types for faster simulation

This is conventional wisdom to use C++ datatypes. Using lesser state types might suffice instead of higher state types. As shown in Line-13, *bool* type is used instead of SystemC defined *sc_bit* data type. However, this approach must be used with care, to make sure that the left-out values (e.g. 'X'/'Z' in this case) are not used anywhere in the code.

6) Writing SystemC module constructor on-the-fly

SystemC module has a constructor with information, like sensitivity list, that is spread across VHDL code. Information for the SystemC constructor has to be gathered while parsing the VHDL module, accumulated on-the-fly and finally written to the SystemC file. This is shown in Line-20.

7) Translating the VHDL process

SystemC has SC_METHOD and SC_THREAD, similar to a VHDL process. A VHDL process with a wait statement is implemented as SC_THREAD, otherwise SC_METHOD might be preferred. As shown in Line-21, SC_METHOD is used, and the sensitivity of the VHDL-process becomes the sensitivity of SystemC translation, as shown in Line-22.

8) Preventing invocation of SystemC process at start-up

SystemC scheduler invokes every process (SC_METHOD or SC_THREAD) at the start of simulation, even in absence of any event! Use `dont_initialize()` to stop this default invocation, as shown in Line-23.

9) Using port-methods for clarity

SystemC uses same operator '=' for variables and ports. To remove ambiguity in understanding the translated source-code, use methods for ports and '=' for variable assignments. As in Line-29, port is read using `.read()` method.

10) Handling clock-edge sensitivity

VHDL and SystemC use varying notations to describe clock edge sensitivity. Line-36 shows the translation for positive-edge clock sensitivity. But the process is invoked on both the edges of clock. Efficient and recommended, but elaborate approach is to analyze the VHDL to determine the clock sensitivity of interest, and then use it in SystemC. Such scheme is better for an event-based simulator like SystemC.

11) Translating switch-cases

VHDL allows variables, logic-types as well as ports in the switch-case construct, whereas SystemC allows only integers. Hence, VHDL switch literal must be converted to an integer, if possible. Another approach is to use if-then-else constructs, as shown in Line-38 and Line-43, where the VHDL switch literal

- uses a range of values
- cannot be converted to an integer, e.g. string types.

12) Chain of SystemC library calls

Sometimes it is necessary in practice to employ a chain of SystemC library calls to achieve the desired behavior, e.g. comparing a single-bit value from a port, as shown in Line-45.

In addition to the above details highlighted in Fig. 1, few other rules are followed for translation, described below:

13) Considering operator precedence differences between VHDL and SystemC

This consideration is extremely important to follow. VHDL and SystemC have different precedence for certain operators. As is common, use parenthesis for clarity and overwriting precedence.

14) Handling concurrent VHDL statements in SystemC

Concurrent VHDL statements can be handled as follows:

a) Single SC_METHOD for all concurrent statements, sensitive to all the source RHS operations in these statements. Drawback is that all the statements are executed whenever any RHS operation changes, affecting the simulation performance.

b) Separate SC_METHOD for each statement, sensitive to only this statement's source variable. This reduces the simulation overhead, but increases the code-size.

15) Executing the SystemC model constructor

SystemC models instantiated inside a top-level module need their constructors to be executed, e.g. implementing a 4-bit adder from 4 instances of 1-bit adder.

Following the above mentioned translation methodology results in the SystemC code that has the following highly desirable code qualities:

Human readability. This is an important consideration about the usage of the generated SystemC code: whether the SystemC code is only to be fed to a compiler or is it going to be maintained by human developers. Many VHDL to SystemC converters lack this feature and generate an obscure code which is not fit for human use. It goes a long way to decide the human relationship with the generated SystemC code.

Correspondence of the translated SystemC to VHDL. If a team of designers needs to maintain both the VHDL and SystemC code-bases, then it is appropriate to have a consistent view between the two. The translation mechanism must decide about this early on and take care of. Usage of the same module names, variables, constructs, etc. must be adhered too, unless SystemC does not support a feature inherently.

IV. Experimental Results

We have performed experiments with a number of available tools described in Section 2. Table 1 demonstrates the results of translating a set of ITC99 benchmarks [19] and a greatest common divisor implementation *gcd*. The EKUT's tool [9] was able to translate *gcd* and the translation results were successfully compiled by a Microsoft Visual C++ based setup of SystemC. Translation of the ITC99 benchmarks contained errors. [10] was able to translate most of the designs but the compilation of them did not succeed. We had access only to the demo version of the HIFSuite v2012.10 tool [7] that implies very strict design size limits. The largest design allowed by this version we have successfully translated was a 3-bit adder implementation. However, the user manual of this tool reports successful translation of all the ITC99 benchmarks listed in the table.

TABLE I. Experimental results for VHDL to SystemC translation

Design	*gcd*	*b01*	*b02*	*b03*	*b04*	*b05*	*b06*	*b07*	*b08*	*b09*	*b10*	*b11*
EKUT	●	-	-	-	-	-	-	-	-	-	-	-
HT-Lab	●	○	○	○	-	-	○	-	○	○	○	-
HIFSuite[1]	N/A	m	m	m	m	m	m	m	m	m	m	m
zamiaCAD	●	●	●	●	●	-	●	-	-	●	-	●

- – the design was not translated
○ – the design was translated but not compiled
m – according to the tool manual data
● – the design was translated and compiled

[1] We have used the demo version of the tool that implies very strict design size limits.

The current implementation of the proposed framework based on zamiaCAD was able to successfully and fully automatically translate into a compilable SystemC output most of the exercised designs. As a separate case-study we have also automatically translated the RTL VHDL *ram* core of the OpenCores.org *plasma* processor design [20]. The resulted cycle-accurate SystemC module was successfully co-simulated in the Mentor Graphics ModelSim environment with the original testbench producing an equivalent system behavior.

V. V. Conclusions and Future Work

With the acceptance of SystemC in the SoC design, IP providers are hard pressed to provide SystemC views of their legacy IPs. At the same time manual conversion of VHDL or Verilog IPs to SystemC is time consuming and error prone. This paper has presented a methodology and implementation as an open-source framework for translating RTL VHDL IPs to cycle-accurate SystemC models. The methodology ensures the generated SystemC code to be human readable with close correspondence to the source VHDL for easy maintainability.

The correctness of the proposed methodology was verified by comprehensive simulation targeting functional and code coverages. Formal equivalence checking of the translated designs approach is under development and will be integrated as an automated step into the framework. Future work also aims at developing a methodology to automatically raise the abstraction level of SystemC from RTL to TLM.

Acknowledgments

The work has been supported in part by EU through the European Regional Development Fund, by Estonian SF grants 8478 and 9429 and by the Tiger University Program of the Estonian Information Technology Foundation (EITSA).

References

[1] SpecC, University of California, http://www.cecs.uci.edu/~specc/

[2] Handel-C Synthesis Methodology, Mentor Graphics, http://www.mentor.com/products/fpga/handel-c/, 2012

[3] Bluespec, Bluespec, Inc. http://www.bluespec.com/, 2012

[4] G.Berry, G.Gonthier, The Esterel synchronous programming language: Design, semantics, implementation. Science of Computer Programming, 19(2), pp. 87-152, 1992.

[5] IEEE 1666 Standard SystemC Language Reference Manual, 2011.

[6] Model studio, Carbon Design Systems, http://carbondesignsystems.com

[7] HIFSuite, EDA-Lab, http://hifsuite.edalab.it, 2012

[8] N. Bombieri, M. Ferrari, F. Fummi, et al., "HIFSuite: Tools for HDL Code Conversion and Manipulation," in EURASIP Journal on Embedded Systems, vol. 1155, no. 10, 2010, pp. 1–20.

[9] VHDL-to-SystemC-Converter, Eberhard-Karls-University of Tübingen, http://www-ti.informatik.uni-tuebingen.de/~systemc/, 2012

[10] VH2SC, HT-Lab, http://www.ht-lab.com, 2012

[11] R. Görgen, J.H. Oetjens, W. Nebel, Automatic integration of hardware descriptions into system-level models, Proc. of IEEE DDECS 2012, Tallinn, pp.105-110

[12] R. Görgen, "VHDL-to-SystemC Transformation," 2012, http://vhome.offis.de/ralphg/vhdl2sc.pdf

[13] W. Snyder, Verilator, Veripool, http://www.veripool.org/wiki/verilator

[14] Edwin Naroska, "FreeHDL," http://www.freehdl.seul.org/, 2012

[15] OStatic, "VHDLC," http://ostatic.com/vhdlc, 2012

[16] zamiaCAD Open-source HW Design Framework, http://zamiacad.sf.net

[17] A. Tšepurov, G. Bartsch, R. Dorsch, M. Jenihhin, J. Raik, V. Tihhomirov, A Scalable Model Based RTL Framework zamiaCAD for Static Analysis, Proc. IEEE VLSI-SOC, pp. 1-6, 2012.

[18] S. A.Syed, M.Jenihhin, J.Raik, Open-source Framework and Practical Considerations for Translating RTL VHDL to SystemC, Pub. by IP-SOC 2012, Design & Reuse, pp. 1-6.

[19] Politecnico di Torino, "ITC-99 Benchmarks," 1999, http://www.cad.polito.it/tools/itc99.html

[20] Plasma CPU a small synthesizable 32-bit RISC microprocessor, http://opencores.org/project,plasma , 2012

CURRICULUM VITAE

Personal data

Name: Syed, Saif Abrar

Date of birth: 18 December 1977

Place of birth: Uttar Pradesh, India

Citizenship: Indian

Contact data

Address: IBM India Pvt Ltd, Nagawara, Bangalore 560045, India

Phone: +91 98865 12327

E-mail: syedsaifabrar@gmail.com

Education

2012 – 2016 PhD, Tallinn University of Technology

1998 – 2000 Masters of Technology, Indian Institute of Technology, Delhi

1994 – 1998 Bachelors of Technology, Aligarh Muslim University, India

1984 – 1994 High school, Aligarh, India

Language competence

English Fluent

Professional employment

2012 – IBM, India

2010 – 2012 Intel Mobile Communications, India

2000 – 2010 Philips Semiconductors, India

ELULOOKIRJELDUS

Isikuandmed

Nimi: Syed, Saif Abrar

Sünniaeg: 18. Detsember 1977

Sünnikoht: Uttar Pradesh, India

Kodakondsus: India

Kontaktandmed

Aadress: IBM India Pvt Ltd, Nagawara, Bangalore 560045, India

Telefon: +91 98865 12327

E-mail: syedsaifabrar@gmail.com

Hariduskäik

2012 – 2016 Tallinna Tehnikaülikool, doktorantuur

1998 – 2000 Indian Institute of Technology, Delhi, magistratuur

1994 – 1998 Aligarh Muslim University, India, bakalaureuseõpe

1984 – 1994 Aligarh, India, keskkool

Keelteoskus

Inglise keel Kõrgtase

Teenistuskäik

2012 – IBM, India

2010 – 2012 Intel Mobile Communications, India

2000 – 2010 Philips Semiconductors, India

DISSERTATIONS DEFENDED AT TALLINN UNIVERSITY OF TECHNOLOGY ON *INFORMATICS AND SYSTEM ENGINEERING*

1. **Lea Elmik**. Informational Modelling of a Communication Office. 1992.

2. **Kalle Tammemäe**. Control Intensive Digital System Synthesis. 1997.

3. **Eerik Lossmann**. Complex Signal Classification Algorithms, Based on the Third-Order Statistical Models. 1999.

4. **Kaido Kikkas**. Using the Internet in Rehabilitation of People with Mobility Impairments – Case Studies and Views from Estonia. 1999.

5. **Nazmun Nahar**. Global Electronic Commerce Process: Business-to-Business. 1999.

6. **Jevgeni Riipulk**. Microwave Radiometry for Medical Applications. 2000.

7. **Alar Kuusik**. Compact Smart Home Systems: Design and Verification of Cost Effective Hardware Solutions. 2001.

8. **Jaan Raik**. Hierarchical Test Generation for Digital Circuits Represented by Decision Diagrams. 2001.

9. **Andri Riid**. Transparent Fuzzy Systems: Model and Control. 2002.

10. **Marina Brik**. Investigation and Development of Test Generation Methods for Control Part of Digital Systems. 2002.

11. **Raul Land**. Synchronous Approximation and Processing of Sampled Data Signals. 2002.

12. **Ants Ronk**. An Extended Block-Adaptive Fourier Analyser for Analysis and Reproduction of Periodic Components of Band-Limited Discrete-Time Signals. 2002.

13. **Toivo Paavle**. System Level Modeling of the Phase Locked Loops: Behavioral Analysis and Parameterization. 2003.

14. **Irina Astrova**. On Integration of Object-Oriented Applications with Relational Databases. 2003.

15. **Kuldar Taveter**. A Multi-Perspective Methodology for Agent-Oriented Business Modelling and Simulation. 2004.

16. **Taivo Kangilaski**. Eesti Energia käiduhaldussüsteem. 2004.

17. **Artur Jutman**. Selected Issues of Modeling, Verification and Testing of Digital Systems. 2004.

18. **Ander Tenno**. Simulation and Estimation of Electro-Chemical Processes in Maintenance-Free Batteries with Fixed Electrolyte. 2004.

19. **Oleg Korolkov**. Formation of Diffusion Welded Al Contacts to Semiconductor Silicon. 2004.

20. **Risto Vaarandi**. Tools and Techniques for Event Log Analysis. 2005.

21. **Marko Koort**. Transmitter Power Control in Wireless Communication Systems. 2005.

22. **Raul Savimaa**. Modelling Emergent Behaviour of Organizations. Time-Aware, UML and Agent Based Approach. 2005.

23. **Raido Kurel**. Investigation of Electrical Characteristics of SiC Based Complementary JBS Structures. 2005.

24. **Rainer Taniloo**. Ökonoomsete negatiivse diferentsiaaltakistusega astmete ja elementide disainimine ja optimeerimine. 2005.

25. **Pauli Lallo**. Adaptive Secure Data Transmission Method for OSI Level I. 2005.

26. **Deniss Kumlander**. Some Practical Algorithms to Solve the Maximum Clique Problem. 2005.

27. **Tarmo Veskioja**. Stable Marriage Problem and College Admission. 2005.

28. **Elena Fomina**. Low Power Finite State Machine Synthesis. 2005.

29. **Eero Ivask**. Digital Test in WEB-Based Environment 2006.

30. **Виктор Войтович**. Разработка технологий выращивания из жидкой фазы эпитаксиальных структур арсенида галлия с высоковольтным p-n переходом и изготовления диодов на их основе. 2006.

31. **Tanel Alumäe**. Methods for Estonian Large Vocabulary Speech Recognition. 2006.

32. **Erki Eessaar**. Relational and Object-Relational Database Management Systems as Platforms for Managing Softwareengineering Artefacts. 2006.

33. **Rauno Gordon**. Modelling of Cardiac Dynamics and Intracardiac Bio-impedance. 2007.

34. **Madis Listak**. A Task-Oriented Design of a Biologically Inspired Underwater Robot. 2007.

35. **Elmet Orasson**. Hybrid Built-in Self-Test. Methods and Tools for Analysis and Optimization of BIST. 2007.

36. **Eduard Petlenkov**. Neural Networks Based Identification and Control of Nonlinear Systems: ANARX Model Based Approach. 2007.

37. **Toomas Kirt**. Concept Formation in Exploratory Data Analysis: Case Studies of Linguistic and Banking Data. 2007.

38. **Juhan-Peep Ernits**. Two State Space Reduction Techniques for Explicit State Model Checking. 2007.

39. **Innar Liiv**. Pattern Discovery Using Seriation and Matrix Reordering: A Unified View, Extensions and an Application to Inventory Management. 2008.

40. **Andrei Pokatilov**. Development of National Standard for Voltage Unit Based on Solid-State References. 2008.

41. **Karin Lindroos**. Mapping Social Structures by Formal Non-Linear Information Processing Methods: Case Studies of Estonian Islands Environments. 2008.

42. **Maksim Jenihhin**. Simulation-Based Hardware Verification with High-Level Decision Diagrams. 2008.

43. **Ando Saabas**. Logics for Low-Level Code and Proof-Preserving Program Transformations. 2008.

44. **Ilja Tšahhirov**. Security Protocols Analysis in the Computational Model – Dependency Flow Graphs-Based Approach. 2008.

45. **Toomas Ruuben**. Wideband Digital Beamforming in Sonar Systems. 2009.

46. **Sergei Devadze**. Fault Simulation of Digital Systems. 2009.

47. **Andrei Krivošei**. Model Based Method for Adaptive Decomposition of the Thoracic Bio-Impedance Variations into Cardiac and Respiratory Components. 2009.

48. **Vineeth Govind**. DfT-Based External Test and Diagnosis of Mesh-like Networks on Chips. 2009.

49. **Andres Kull**. Model-Based Testing of Reactive Systems. 2009.

50. **Ants Torim**. Formal Concepts in the Theory of Monotone Systems. 2009.

51. **Erika Matsak**. Discovering Logical Constructs from Estonian Children Language. 2009.

52. **Paul Annus**. Multichannel Bioimpedance Spectroscopy: Instrumentation Methods and Design Principles. 2009.

53. **Maris Tõnso**. Computer Algebra Tools for Modelling, Analysis and Synthesis for Nonlinear Control Systems. 2010.

54. **Aivo Jürgenson**. Efficient Semantics of Parallel and Serial Models of Attack Trees. 2010.

55. **Erkki Joasoon**. The Tactile Feedback Device for Multi-Touch User Interfaces. 2010.

56. **Jürgo-Sören Preden**. Enhancing Situation – Awareness Cognition and Reasoning of Ad-Hoc Network Agents. 2010.

57. **Pavel Grigorenko**. Higher-Order Attribute Semantics of Flat Languages. 2010.

58. **Anna Rannaste**. Hierarcical Test Pattern Generation and Untestability Identification Techniques for Synchronous Sequential Circuits. 2010.

59. **Sergei Strik**. Battery Charging and Full-Featured Battery Charger Integrated Circuit for Portable Applications. 2011.

60. **Rain Ottis**. A Systematic Approach to Offensive Volunteer Cyber Militia. 2011.

61. **Natalja Sleptšuk**. Investigation of the Intermediate Layer in the Metal-Silicon Carbide Contact Obtained by Diffusion Welding. 2011.

62. **Martin Jaanus**. The Interactive Learning Environment for Mobile Laboratories. 2011.

63. **Argo Kasemaa**. Analog Front End Components for Bio-Impedance Measurement: Current Source Design and Implementation. 2011.

64. **Kenneth Geers**. Strategic Cyber Security: Evaluating Nation-State Cyber Attack Mitigation Strategies. 2011.

65. **Riina Maigre**. Composition of Web Services on Large Service Models. 2011.

66. **Helena Kruus**. Optimization of Built-in Self-Test in Digital Systems. 2011.

67. **Gunnar Piho**. Archetypes Based Techniques for Development of Domains, Requirements and Sofware. 2011.

68. **Juri Gavšin**. Intrinsic Robot Safety Through Reversibility of Actions. 2011.

69. **Dmitri Mihhailov**. Hardware Implementation of Recursive Sorting Algorithms Using Tree-like Structures and HFSM Models. 2012.

70. **Anton Tšertov**. System Modeling for Processor-Centric Test Automation. 2012.

71. **Sergei Kostin**. Self-Diagnosis in Digital Systems. 2012.

72. **Mihkel Tagel**. System-Level Design of Timing-Sensitive Network-on-Chip Based Dependable Systems. 2012.

73. **Juri Belikov**. Polynomial Methods for Nonlinear Control Systems. 2012.

74. **Kristina Vassiljeva**. Restricted Connectivity Neural Networks based Identification for Control. 2012.

75. **Tarmo Robal**. Towards Adaptive Web – Analysing and Recommending Web Users` Behaviour. 2012.

76. **Anton Karputkin**. Formal Verification and Error Correction on High-Level Decision Diagrams. 2012.

77. **Vadim Kimlaychuk**. Simulations in Multi-Agent Communication System. 2012.

78. **Taavi Viilukas**. Constraints Solving Based Hierarchical Test Generation for Synchronous Sequential Circuits. 2012.

79. **Marko Kääramees**. A Symbolic Approach to Model-based Online Testing. 2012.

80. **Enar Reilent**. Whiteboard Architecture for the Multi-agent Sensor Systems. 2012.

81. **Jaan Ojarand**. Wideband Excitation Signals for Fast Impedance Spectroscopy of Biological Objects. 2012.

82. **Igor Aleksejev**. FPGA-based Embedded Virtual Instrumentation. 2013.

83. **Juri Mihhailov**. Accurate Flexible Current Measurement Method and its Realization in Power and Battery Management Integrated Circuits for Portable Applications. 2013.

84. **Tõnis Saar**. The Piezo-Electric Impedance Spectroscopy: Solutions and Applications. 2013.

85. **Ermo Täks**. An Automated Legal Content Capture and Visualisation Method. 2013.

86. **Uljana Reinsalu**. Fault Simulation and Code Coverage Analysis of RTL Designs Using High-Level Decision Diagrams. 2013.

87. **Anton Tšepurov**. Hardware Modeling for Design Verification and Debug. 2013.

88. **Ivo Müürsepp**. Robust Detectors for Cognitive Radio. 2013.

89. **Jaas Ježov**. Pressure sensitive lateral line for underwater robot. 2013.

90. **Vadim Kaparin**. Transformation of Nonlinear State Equations into Observer Form. 2013.

92. **Reeno Reeder**. Development and Optimisation of Modelling Methods and Algorithms for Terahertz Range Radiation Sources Based on Quantum Well Heterostructures. 2014.

93. **Ants Koel**. GaAs and SiC Semiconductor Materials Based Power Structures: Static and Dynamic Behavior Analysis. 2014.

94. **Jaan Übi**. Methods for Coopetition and Retention Analysis: An Application to University Management. 2014.

95. **Innokenti Sobolev**. Hyperspectral Data Processing and Interpretation in Remote Sensing Based on Laser-Induced Fluorescence Method. 2014.

96. **Jana Toompuu**. Investigation of the Specific Deep Levels in *p*-, *i*- and *n*-Regions of GaAs p^+-*pin*-n^+ Structures. 2014.

97. **Taavi Salumäe**. Flow-Sensitive Robotic Fish: From Concept to Experiments. 2015.

98. **Yar Muhammad**. A Parametric Framework for Modelling of Bioelectrical Signals. 2015.

99. **Ago Mõlder**. Image Processing Solutions for Precise Road Profile Measurement Systems. 2015.

100. **Kairit Sirts**. Non-Parametric Bayesian Models for Computational Morphology. 2015.

101. **Alina Gavrijaševa**. Coin Validation by Electromagnetic, Acoustic and Visual Features. 2015.

102. **Emiliano Pastorelli**. Analysis and 3D Visualisation of Microstructured Materials on Custom-Built Virtual Reality Environment. 2015.

103. **Asko Ristolainen**. Phantom Organs and their Applications in Robotic Surgery and Radiology Training. 2015.

104. **Aleksei Tepljakov**. Fractional-order Modeling and Control of Dynamic Systems. 2015.

105. **Ahti Lohk**. A System of Test Patterns to Check and Validate the Semantic Hierarchies of Wordnet-type Dictionaries. 2015.

106. **Hanno Hantson**. Mutation-Based Verification and Error Correction in High-Level Designs. 2015.

107. **Lin Li**. Statistical Methods for Ultrasound Image Segmentation. 2015.

108. **Aleksandr Lenin**. Reliable and Efficient Determination of the Likelihood of Rational Attacks. 2015.

109. **Maksim Gorev**. At-Speed Testing and Test Quality Evaluation for High-Performance Pipelined Systems. 2016.

110. **Mari-Anne Meister**. Electromagnetic Environment and Propagation Factors of Short-Wave Range in Estonia. 2016.