a thesis submitted for the degree of Dr.-Ing.

SystemC Through the Looking Glass: Non-Intrusive Analysis of Electronic System Level Designs in SystemC

by

Jannis Stoppe

Supervisor:

Prof. Dr. Rolf Drechsler

Second referee:

Prof. Dr. Stefan Edelkamp

CONTENTS

DISCLAIMER

I hereby declare that

- this dissertation is my own original work,
- it has been completed without claiming any illegitimate assistance and
- I have acknowledged all sources used (both, verbatim and regarding their content).

Jannis Stoppe, February 27, 2017

I would like to thank Robert Wille
for the assistance he offered whenever I needed it
and the time he put into me

and Rolf Drechsler for enabling me
to present my ideas to the world
and giving me the opportunity to work in a beautiful environment.

Your support has been invaluable.

I would also like to thank my wife and children
for being a patient, loving support during stressful times.

You give meaning and make me endure,
which is more than I ever could have hoped for.

1 INTRODUCTION AND MOTIVATION

We can only see a short distance ahead, but we can see plenty there that needs to be done.

Alan Turing [113]

The complexity of digital systems is increasing.

Ever since Gordon E. Moore projected that the amount of components in integrated circuits would double every year in 1965 [75] (and later revised that to every two years in 1975 [76]), his prediction would hold. This exponential growth not only leads to faster and more complex systems, it also results in serious issues concerning their design: A current 18-core Intel Haswell Xeon chip consists of more than 5 billion transistors [79] – nearly twice as many as there are base pairs in the human DNA [115].

Being able to manufacture such large systems, while being an impressive achievement by itself, still requires them to be designed in the first place. This simple aspect is a serious problem for the development of modern chips as the design tools must enable the designers to handle such large systems: if these large systems cannot be designed, they cannot be manufactured.

Classic Hardware Description Languages (HDLs) that have been designed to model hardware are – by definition – close to the hardware they describe. This also means that systems that are written using these HDLs cannot easily be run and/or tested if the description is still incomplete. Thus, the development process becomes a bottom-up approach, requiring designers to first build the system as a whole and then start testing it. Issues with the overall architecture can therefore only be found once the system has been fully implemented, making decisions that need to revise architectural properties expensive and thus large systems tough to be designed.

Traditional HDLs hence struggle with the complexity of modern hardware. In order to keep up with the manufacturing advances, the designs are following a modular approach and the distinct parts are re-used. Especially in conjunction with well-scaling parts such as memory (which can simply be "made larger" in order to gain the according benefits), these larger designs can still be achieved – at least to a certain degree. Still, complexity remains a key issue, especially for designs that cannot be scaled easily or do not benefit from such a scaling. HDLs are therefore increasingly facing the issue of the *design gap* [35], i.e. the problem that systems cannot be designed in accordance to the available manufacturing capabilities due to their complexity.

In order to handle this complexity, more abstract description languages have been introduced. These so-called system level descriptions are used to model the coarse structure using established features (such as modules and signals) but use an underlying high-level programming language (such as C++) to describe the detailed behaviour of these parts. This means that a system can be prototyped more quickly and the system as a whole may be tested before the actual hardware (or software) parts have been implemented.

The current de-facto standard for system level design is SystemC [7]. Implemented as a C++ library and specified in an open standard [81], it combines the benefits of an established high-level programming language (such as tool chains, libraries, Integrated Development Environments (IDEs) etc.) with the ability to describe hardware designs on an abstract level and simulate them despite being implemented in this abstract way.

Although these system level languages strive to reduce the amount of complexity needed to create a prototypical system design, the designs still are considerably complex. To appropriately model all parts of a system and their interactions, the design usually becomes quite large, even when working on higher levels of abstraction. The sheer size of a design may make it difficult for new designers to join a project. Acquiring knowledge about a design, either to introduce new members into a team or to keep other people working on the project up to date with recent changes, is therefore a critical part of the design process.

This *Design Understanding* can be approached in a variety of ways, depending on the available sources of information. Reading the source code of the design is usually inefficient and not an appropriate way of starting to understand a design, especially for larger designs. While a proper (manually created) documentation is usually the best way to introduce designers to certain parts of the work, the creation and especially maintenance of a manual is a time-consuming (and therefore expensive) process with no immediate benefit for a project, making it sometimes infeasible to keep the documentation in an appropriate condition.

Methods to automatically generate or retrieve more abstract representations of hardware designs to allow designers to quickly grasp important structures are one way to approach the problem of imperfect, manually created meta documents. The automatic generation of e.g. visual representations of system designs is a common way to quickly allow designers to take a glance at a system's general structure. While this approach of course needs to address issues such as the complexity of the resulting images [45] or the performance of the visualisation [58, 128], deriving the structure to be displayed from the source code has been a straightforward procedure for classic HDLs.

System level descriptions in general and SystemC implementations in particular level descriptions are, however, harder to analyse. The hardware structures are not described statically as in classic HDLs but instead are *created dynamically at run-time*, making a static analysis approach to extract the structure more difficult or even impossible. Additionally, C++ is an older language with lots of different dialects available that is not even considered context-free. This means that writing a single parser that could serve as a front end to analyse a design is an equally futile task. Lastly, compilers are used to translate the system descriptions into an executable form – in case of C++ binary code which can directly be executed on a given computer. This means that once the program has been processed to be executable, it is even harder or downright impossible to analyse.

This work proposes new methods and approaches to analyse SystemC designs despite these issues and without shifting the burden to the designer (by e.g. requiring the source code to adhere to certain standards). More precisely, the research question of this thesis is

> *How can the desired information from a given SystemC design be extracted without assuming restricted language means and without modifying the existing infrastructure like parsers or compilers?*

This means that there are two core requirements to any solution. The first is that the source code should be considered untouchable. If the solution to the analysis problem is anything that requires major source code alterations, the designers may as well simply add the required tracing functionality to their code without utilizing another solution. As the time required to use a solution can be directly translated to the money that an engineer needs to be paid, the solution must be easily added in order to prevent large costs for something like an analysis tool. The second requirement is that the resulting method should be compatible to at least all major C++ compilers (i.e. gcc, clang and Microsoft Visual C++ Compiler (MSVC++)). As SystemC adheres strictly to the C++ standard, it can be used in any type of project, regardless of the code elements being used in them. This means that a SystemC project may use any C++ dialect to describe the system. Therefore, while an analysis tool may not support each and every exotic framework available, any method to analyse SystemC should at least be able to handle the source code used on the major platforms.

These points are summarized as the trait of *non-intrusiveness*, describing that an existing project (encompassing both, infrastructure such as compilers being used and the source code of the design itself) should not be altered by any analysis approach. It is an important issue concerning the methods that can be used to analyse a system description and thus represents a cornerstone of design decisions throughout this thesis.

The remainder of this thesis first gives an overview of existing HDLs in general and SystemC in particular in Chapter 2. Afterwards, there are four Chapters that provide the core contributions of this work.

- Chapter 3 illustrates how a static model describing the design can be extracted from a SystemC description.
- Chapter 4 then shows how these techniques can be further refined to be applied to running designs, allowing their behaviour to be traced as well.
- Chapter 5 explains how information that was not retrieved using the previous steps can be learned using approaches from the machine learning domain.
- Chapter 6 finally illustrates how these approaches can be applied and shows the according use-cases.

Chapter 7 contains a brief summary of the thesis. It proposes several research questions for potential future work as well.

The approaches that are proposed in this thesis have been previously published on various occasions:

- Martin Ring, Jannis Stoppe, Christoph Lüth, and Rolf Drechsler. Change impact analysis for hardware designs. In *Forum on specification & Design Languages (FDL)*, 2016
- Jannis Stoppe, Arved Friedemann, and Rolf Drechsler. SystemCDG – AI based coverage driven stimuli generation for SystemC. In *International Workshop on Logic & Synthesis (IWLS)*, 2016
- Martin Ring, Jannis Stoppe, Christoph Lüth, and Rolf Drechsler. Change management for hardware designers. In *Workshop on Design Automation for Understanding Hardware Designs (DUHDe)*. IEEE, 2016
- Rolf Drechsler and Jannis Stoppe. Hardware/software co-visualization on the electronic system level using SystemC. In *International Conference on VLSI Design (VLSID)*, pages 44–49, 2016
- Jannis Stoppe and Rolf Drechsler. Analyzing SystemC designs: SystemC analysis approaches for varying applications. *Sensors*, 15(5):10399 – 10421, 2015
- Nils Przigoda, Jannis Stoppe, Julia Seiter, Robert Wille, and Rolf Drechsler. Verification-driven design across abstraction levels: A case study. In *Euromicro Conference on Digital System Design (DSD)*, pages 375 – 382. IEEE, 2015
- Jannis Stoppe, Robert Wille, and Rolf Drechsler. Automated feature localization for dynamically generated SystemC designs. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 277 – 280. EDA Consortium, 2015
- Jannis Stoppe and Rolf Drechsler. Ecore model generation from SystemC/C++ implementations. In *Workshop on Design Automation for Understanding Hardware Designs (DUHDe)*. IEEE, 2015
- Jannis Stoppe and Rolf Drechsler. KI-Unterstützung im Systementwurf. *Industrie Management – Zeitschrift für industrielle Geschäftsprozesse (IM)*, 1:21 – 24, 2015
- Jannis Stoppe, Robert Wille, and Rolf Drechsler. Validating SystemC implementations against their formal specifications. In *Symposium on Integrated Circuits and Systems Design (SBCCI)*. ACM, 2014
- Jannis Stoppe, Marc Michael, Mathias Soeken, Robert Wille, and Rolf Drechsler. Towards a multi-dimensional and dynamic visualization for ESL designs. In *Workshop on Design Automation for Understanding Hardware Designs (DUHDe)*. IEEE, 2014
- Jannis Stoppe, Robert Wille, and Rolf Drechsler. Cone of influence analysis at the Electronic System Level using machine learning. In *Euromicro Conference on Digital System Design (DSD)*, pages 582 – 587. IEEE, 2013
- Jannis Stoppe, Robert Wille, and Rolf Drechsler. Data extraction from SystemC designs using debug symbols and the SystemC API. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 26 – 31. IEEE, 2013

Additionally, further papers were published after the editorial deadline. While these are directly or indirectly related to the work presented here, they are not covered in this thesis.

- Rolf Drechsler and Jannis Stoppe. Hardware/software co-visualization: The lost world. In *International Workshop on Boolean Problems (IWSBP)*, 2016
- Mehran Goli, Jannis Stoppe, and Rolf Drechsler. AIBA: an automated intra-cycle behavioral analysis for SystemC-based design exploration. In *International Conference on Computer Design (ICCD)*, 2016
- Mehran Goli, Jannis Stoppe, and Rolf Drechsler. Automatic equivalence checking for SystemC-TLM 2.0 models against their formal specifications. In *Design, Automation and Test in Europe (DATE)*, 2017
- Kevin Leonard Schneider, Oliver Keszöcze, Jannis Stoppe, and Rolf Drechsler. Effects of cell shapes on the routability of digital microfluidic biochips. In *Design, Automation and Test in Europe (DATE)*, 2017

2 PRELIMINARIES

Simplicity does not precede complexity, but follows it.

Alan Jay Perlis [86]

The physical elements that constitute a computer system are its Hardware – as opposed to the Software, which is made up of the programs and data which are executed and stored on a given hardware.

While the term hardware encompasses all physical parts of a computer system [21] (i.e. including parts such as cases, keyboards, mounts etc.), this work focuses on aspects of the design of Integrated Circuits (ICs) (also referred to as microchips). These ICs are of electronic circuits on a small plate of semiconductor material onto which the required parts are applied [75].

Just like software (or any other system), hardware needs to be designed. For small systems (as they were common during the early days of computer systems), the hardware can be designed manually. In this case, the elements and parts that are available in hardware (i.e. transistors and connections) are set up and laid out until the system is ready to be built. Figure 1 illustrates how this classic design on paper looks like: parts of the design were simply drawn and described. While the approach of doing this using pen and paper worked well for early, small designs, modern ones consist of billions of transistors, each merely a few nanometres in size.

Obviously, designing such systems manually, i.e. by simply laying out hardware parts, is infeasible. To be able to handle designs of such scales, several different hardware description languages were developed, each with its own features and issues. The most distinct differences, however, can be seen regarding different abstraction layers. Hardware can be described using various means. The available components that are the obvious choice, leading to a fine-grained description of a hardware design that can easily be manufactured. Another way is to use formulas that can be mapped to these components, allowing designers to use mathematical constructs to describe a design. Various other abstraction levels are also available, up to natural language: specifications written in English are still hardware descriptions, albeit they may be inaccurate or incomplete. This variety of description means allows designers to work on systems using different granularities, depending on the precision, speed and features being required for the process.

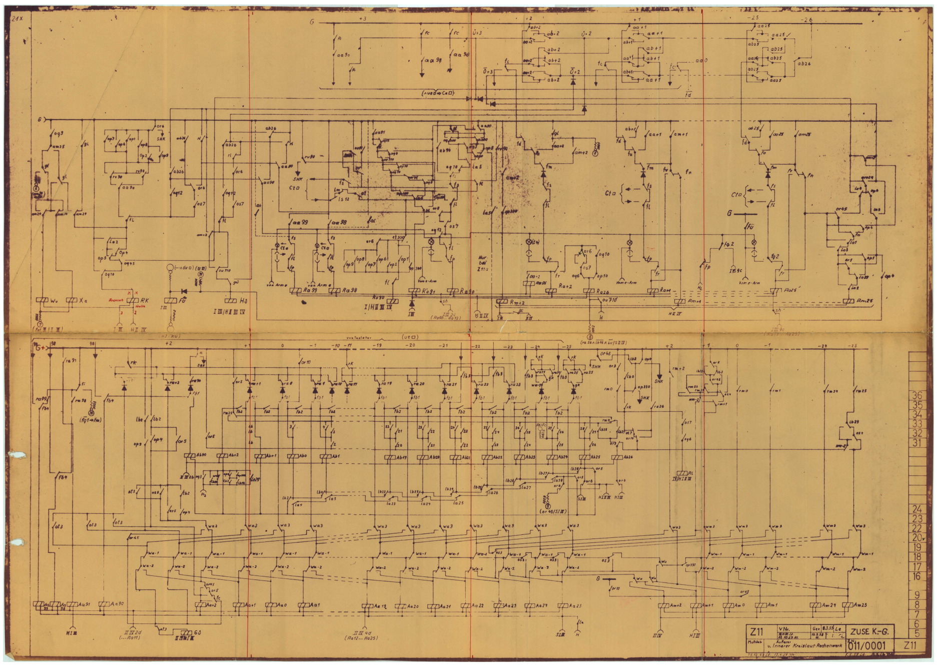


Figure 1: The plan of a part of the Arithmetic and Logical Unit (ALU) of Conrad Zuse's Z11 [130].

This chapter gives an overview on hardware design paradigms, outlining different approaches to hardware design in Section 2.1. As this work specifically focuses on aspects of designing hardware on the Electronic System Level (ESL) using SystemC, Section 2.2 then focuses on this level of abstraction and the SystemC library in particular.

HARDWARE DESIGN APPROACHES

Today's approach to hardware design is layered, going from abstract towards more detailed system descriptions. Each of these layers serves a particular purpose, allowing designers to handle the respective design tasks at the abstraction level that fits best a given goal.

Different abstraction levels to describe hardware

TRANSISTORS are the fundamental elements of current ICs. A transistor can be used to amplify or switch an electronic current. Combined with the ability to manufacture them on small scales using modern photolithography (i.e. "printing" them onto semiconducting material), the transistor has become *the* basic building block for all parts in an IC. The *Transistor Level* is describing how these parts need to be connected for a given circuit design, making it the lowest abstraction level.

When designing hardware systems, working on this lowest of all abstraction levels leads to large descriptions as even mundane systems tend to require a large amount of transistors. Figure 1 illustrates the complexity of these layouts. With even early computing machines suf-

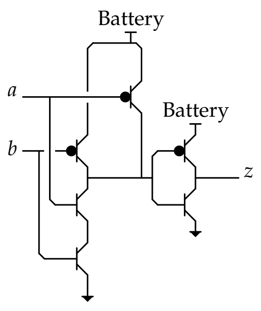


Figure 2: AND gate built from transistors

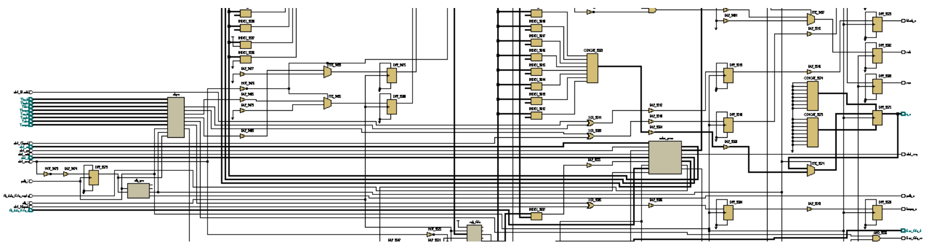


Figure 3: Visualization of a hardware design at the gate level.

fering from the complexity of descriptions that relied on transistors as their primary means of specification, it is obvious that designing systems on this level is not a sustainable approach to hardware design. Instead, while the translation to transistors (and their layout) is required for a system to be built, this is usually done automatically in order to let designers work on more abstract descriptions that allow them to handle a given system's complexity more easily.

THE GATE LEVEL is the first abstraction to the description of the transistors and their connections. As transistors can be used to model certain boolean functions, systems can be described using these and then be directly mapped to their respective transistor counterparts. Figure 2 illustrates this relation, showing how an exemplary boolean AND can be modelled using transistors. Figure 3 shows a visualization of a hardware design at the gate level.

The advantage of this abstraction level is that there exists a direct correspondence from a mathematical set of tools to a hardware design. Thus, not only can formulas that are used to describe a system be translated into a hardware description but the methods developed in the field of boolean algebra can directly be applied to improve a given design.

One approach to this abstraction level is the description of a combinatorial circuit using a truth table or Binary Decision Diagram (BDD), both of which can easily be translated to a boolean function. Once this is done, methods such as e.g. the Quine-McCluskey algorithm [67] can be used to minimize this function. The resulting function is minimal, which means it uses fewer boolean operators and variables in order to describe the same result. With boolean operations being implemented as gates that can directly be translated to transistor descriptions, this also means that the resulting circuit will use fewer transistors and thus require less space (and thus be cheaper to produce). This illustrates how the abstraction of using gates to describe a circuit can improve the design process.

No registers in boolean logic

Unfortunately though, using boolean functions to describe circuits not only provides a rather small step concerning the abstraction of the description (and thus still results in large descriptions), it also practically disregards the notion of persistent states. ICs can be divided into the combinatorial logic (which can easily be described as a boolean function) and its registers that retain their state until specifically being set to a new state. In order to make this defining feature of computer hardware a core feature of the description as well (and thus be able to efficiently design circuits that actually use it), another, even more abstract layer was introduced.

THE REGISTER TRANSFER LEVEL (RTL) focuses on the description of registers and the logic that interconnects them (i.e. that transfers information between registers – hence the name). This core idea of hardware design on the Register Transfer Level (RTL) is therefore already close to traditional (software) programming paradigms that rely on data and operations. However, due to their intended usage as HDLs, languages that describe hardware on the RTL have very distinct features that separate them from modern high-level software programming languages. The most prominent of these is their inherent parallelism. As hardware describes a set of transistors and intermediate wires, elements described in these HDLs are not evaluated sequentially (as it is the case for software) but instead represent a system where assignments happen in parallel.

Verilog and VHDL

The most famous examples of RTL HDLs are the VHSIC Hardware Description Language (VHDL) [80] and Verilog [112]. Both of these were regarded as "breakthroughs" [20] concerning the development of digital circuits after they were introduced as they finally enabled designers to abstract from gate level designs. They both have been standardized by the Institute of Electrical and Electronics Engineers (IEEE) [41, 42]. Figure 4 shows the visualization of an RTL hardware description.

Notice that these two languages have been extended over the years and also include non-synthesizeable language constructs that allow designers to more rapidly describe a circuit's behaviour but require this description to be altered or rewritten before it can actually translated into hardware. While this seems to contradict the idea of RTL descriptions being *hardware* descriptions, it illustrates the issue that hardware design on the RTL faces: the level of abstraction no longer suffices to describe hardware on the current level of complexity. However, larger

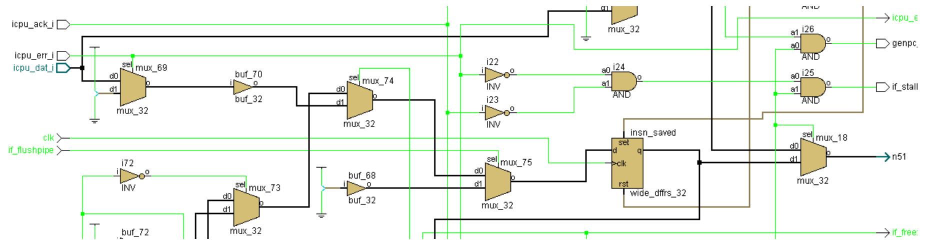


Figure 4: Visualization of a hardware design at the RTL.

abstractions necessarily result in descriptions that may be executable but not snythesizeable. Even simple means such as a loop that *may* be unbounded (i.e. has no well-defined upper bound concerning how often it is executed) or a second `wait` statement in a process that lets a given description wait for another signal in two different "locations" of its description are easy to execute but cannot be synthesized. This means that the HDLs being used today offer designers the choice of (temporarily) adding more abstract elements that limit the description to a simulation for the sake of more quickly having a working prototype of a given system.

The Electronic System Level (ESL) basically inverts this line of thought. Instead of enriching RTL HDLs with abstract features, ESL descriptions use already-abstract high-level programming languages and add hardware-description capabilities to these.

This results in a much more "top-down" approach. As a system description can use all high-level constructs that are offered by its underlying programming language, a first prototype is usually established quickly on the ESL. Especially the abundance of libraries for modern software programming language helps speeding up this process: instead of e.g. having to implement an image processing algorithm in a language that is synthesizeable (or close to it), any existing library (such as e.g. OpenCV [10] for an image processing problem) can be used to add this functionality to a given ESL description. A "module", i.e. a block of the hardware design, is thus regarded as a black box – its internal workings remain hidden to the surrounding framework (and maybe even the designer).

While certain parameters (such as the time that is required to calculate a result in the final system) may need to be guessed or ignored for the time being, this approach results in the ability to very rapidly see if the architecture of a design works as intended. The detailed hardware implementation is thus no longer part of the hardware description. Instead, ESL designs give designers the ability to see if the anticipated behaviour of the given parts of the design works as intended – and from that point, gradually decrease the abstraction of the description until it can be synthesized again. Generally, the ESL can thus be regarded as an

ESL descriptions are executable specifications executable specification: while it (mostly) cannot be synthesized, it can simulate the desired behaviour and thus specifies in detail the interfaces the final implementation needs to comply with.

THE FORMAL SPECIFICATION LEVEL (FSL) adds another layer of abstraction to even more rapidly design the cornerstones of a system. Instead of providing an ESL-like executable specification, the Formal Specification Level (FSL) focuses on the specifications of the system itself, sacrificing the executability to gain more abstraction and thus allow designers to work more rapidly.

The Unified Modeling Language (UML) [9, 117] is the most prominent example of a formal specification language. Focused on software development, it allows designers to plan the structure of their application, describe the desired behaviour or constrain the input and output of functions using Object Constraint Language (OCL) constraints.

An adaption of the UML that instead focuses on the development of hardware designs is the Systems Modeling Language (SysML). The software-centric view of the UML is replaced with description means for hardware designs, allowing designers to formally describe a system before starting with an implementation on the ESL.

While these cannot be executed (and thus clearly offer fewer details than the ESL designs), they enable designers to quickly develop unambiguous schematics for a given design. These can be used to e.g. communicate the design to other people who are part of the development process (such as other designers, customers, management etc.) in an understandable fashion or even (due to their formal nature) locate errors (such as deadlocks or oxymorons).

THE INFORMAL SPECIFICATION LEVEL (ISL) finally is the most abstract way to describe a system. Using natural language and without any further restrictions, the Informal Specification Level (ISL) is used to draft out a system description as a first step.

While there are approaches to automatically analyse and process these natural language specification [43], these approaches are neither complete nor flawless and can in general only be considered an effort to assist a manual translation to a less abstract description. Instead, any natural language description on the ISL should be seen as a way to communicate between people, describing the system in the level of detail that is required to achieve whatever aim the description serves.

The description means for hardware thus form a hierarchy from the most abstract description to the least abstract one. Starting with the ISL that has virtually no restrictions over the FSL that provides a formal specification and the ESL that provides an executable specification, the design is usually finalized on the RTL that provides a synthesizeable description and can thus be automatically translated into hardware

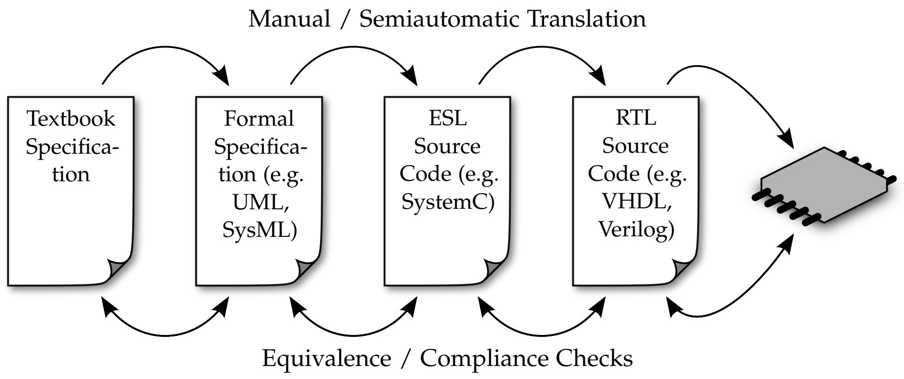

Manual / Semiautomatic Translation
Textbook Specification
Formal Specification (e.g. UML, SysML)
ESL Source Code (e.g. SystemC)
RTL Source Code (e.g. VHDL, Verilog)
Equivalence / Compliance Checks

Figure 5: Today's hardware design flow

schematics. Figure 5 illustrates these connections between the different layers of hardware design approaches.

SYSTEMC

SystemC is a C++ library for modelling and simulating system designs on the ESL. Like other ESL modelling frameworks, it provides means to model hardware structures and a simulation kernel that provides the needed logic to simulate the parallel behaviour that is inherent to hardware designs. By providing descriptions means for both, hardware concepts (like modules, signals, ports, etc.) and software concepts (like class instantiations, function calls, memory allocation, etc.), it allows to model and to execute hardware and software systems on various levels of abstraction. While modules and their connections (both representing parts of a hardware system) are instantiated, the logic behind those can be both, made up of simulated hardware elements down to gate level or just a software simulation of the behavior that is supposed to be realized in hardware or software later on.

While a variety of ESL description libraries exist that extend most major high-level languages with hardware description means (such as Esys.net for developing in .net/mono [51, 52], Hardware Join Java for developing in Java [46], Chisel for developing in Scala [2], and more), SystemC has emerged as the "top of the pack". By now, SystemC is considered an "industry standard" [96], has been standardized [81] and is widely used to prototype the structure of hardware/software systems as well as their behaviour in both, academia and industry.

Example 1 *Figure 6 shows a SystemC program that realizes a simple carry-ripple adder. The bit-width of the adder is not statically defined, but will be provided by the user when executing the program. This is realized by iteratively instantiating new one-bit full adders.*

When writing hardware designs in SystemC, designers have the full spectrum of the underlying language at their disposal. Modules are class

```
#include <systemc.h>

SC_MODULE(fullAdder) {
  sc_in<bool> a, b, carryIn;
  sc_out<bool> result, carryOut;

  void calculate() {
    carryOut.write((a && carryIn) || (b && carryIn) || (a && b));
    result.write((a && !b && !carryIn) || (!a && b && !carryIn) ||
      (!a && !b && carryIn) || (a && b && carryIn));
  }

  SC_CTOR(fullAdder): a("a"), b("b"), result("result"), carryOut("
      carryOut") {
    SC_METHOD(calculate);
    sensitive << a << b;
  }
};

int sc_main (int argc , char *argv[]) {
  int bits = 2;
  if (argc > 1)
    bits = max(0, min(16, atoi(argv[1])));
  fullAdder* previous;
  for (int i = 0; i < bits; i++) {
    fullAdder* fa = new fullAdder("fullAdder");
    if (i > 0) {
      sc_signal<bool>* sig = new sc_signal<bool>("carrySignal");
      previous->carryIn(*sig);
      fa->carryOut(*sig);
  }
  previous = fa;
  }
  return 0;
}
```

Figure 6: SystemC program

instances, their internal states (registers, flip-flops etc.) can be modelled using field variables and the internal logic is written using arbitrary C++ code.

This leads to a design paradigm that detaches the behaviour from its implementation. Where RTL languages such as VHDL provide designers with the ability to specify *several* implementations to choose from, SystemC abstracts further from this concept and does not require anything as long as the behaviour has been described properly. In VHDL, designers e.g. need to decide which kind of adder should be used if two numbers should be added – e.g. whether or not a carry lookahead should be used and which one. SystemC on the other hand is focused on the efficient simulation of these systems, leaving the question of implementation details for later in the design process: adding two numbers is done, and while the designer may specify how much time the final implementation supposedly will take, this is not necessary.

Hardware / Software Co-Design

The designer does not even need to decide whether a given functionality should be implemented in software or hardware. Instead, the behaviour that is specified may later be translated to hardware, software or a mixture of both. This principle is called *Hardware/Software Co-Design* [18] and allows designers to use SystemC to quickly develop working prototypes of hardware/software systems without being required to implement all the details that would be needed in order to have a working e.g. VHDL design.

This term reflects how the design process encompasses the development of both, software and hardware parts, that later interact closely. The SystemC design process may start with an arbitrary, abstract C++ implementation which is then iteratively translated into a combination of RTL desriptions (which may be embedded into the SystemC simulation) and C programs that are executed on the RTL parts until the whole system becomes a set of synthesizeable parts and can thus be translated into a single hardware / software implementation.

SystemC simulation kernel

Beyond the classes that represent hardware structures, SystemC offers a simulation kernel that manages the execution of the methods that represent a module's internal logic . Two concepts of the simulation kernel are of special interest for this work:

- The simulation kernel splits the execution of a SystemC program into two phases, the *elaboration phase* and the *simulation phase*. The former is meant to create the system that is to be simulated, i.e. all (virtual) hardware parts such as modules and signals are initialized. The latter then simulates this system and actively *prohibits* the creation of further SystemC objects.

- The simulation kernel manages an *event queue* that contains all processes and threads that need to be executed at a later point in simulation time. This means that the SystemC kernel runs an event

> driven simulation, waking up those parts of the design that need to be updated and letting all other elements of the design lay dormant, thus saving computational power. This way, the simulation kernel simulates the parallelism that is inherent to the hardware parts that are modelled by SystemC. Note that the kernel, while it simulates the parallelism of the simulated system, is not itself running in parallel but is instead a strictly sequential, single-threaded software, thus avoiding issues that may arise from parallel execution such as race conditions at the cost of decreased performance.

Its design allows SystemC to separate the functionality (that needs to be built in order to end up with an executable system) from the implementation. In this case, the latter means either the hardware that implements the desired behaviour or the combination of the hardware platform and the software that is executed on it in order to end up with a module that implements the given behaviour. The C++ implementation that describes the behaviour for SystemC designs should be considered an executable specification instead of an implementation though. As the given source code that describes a thread's behaviour can neither be translated into software nor hardware automatically, it does not qualify for being an implementation in the sense of embedded systems. However, while it is much more abstract than a hardware implementation or the combination of an IC and the embedded software running on it (and may ignore several other traits of an implementation such as timing), it is of course still a C++ implementation that is executed to emulate the module's behaviour.

While these concepts are already focusing on shorter design cycles by letting designers create executable prototypes as early as possible, the concept of arbitrary C++ code includes libraries, which results in even faster prototyping of certain modules. As there is a vast amount of C++ libraries available, they may all be used to quickly develop solutions for a given problem. This means that development cycles are much shorter, allowing for designers to quickly iterate design prototypes with customers, start to write tests early on and validate design concepts at a lower cost than using the traditional hardware design approach.

SystemC performance

While this approach results in shorter development processes by allowing designers to quickly build working (i.e. executable) prototypes, it also has advantages when running the simulation itself. As the module logic is implemented in arbitrary C++ code, no simulation kernel call is required to calculate the result of an operation. Instead, after the kernel wakes up e.g. a given method, the results are calculated at once and the designer may instruct the simulation kernel to wait for a given time frame in order to emulate how the module would require some time to finish the given calculations. For the simulation, this means that much less overhead is required in order to perform calculations as the simulation kernel needs to be invoked less often.

TLM

Transaction Level Modeling (TLM) is the next step in this manner, altering the way modules communicate in order to increase performance by reducing the kernel activity associated with these operations. While TLM had started as an optional extension to SystemC, it is now part of the standard and serves the purpose of further increasing simulation performance and unifying transactions between modules. It does so by introducing two core concepts to the existing foundation of the SystemC library:

1. *Temporal Decoupling* is the idea of letting different parts of the SystemC model advance at different speeds. The traditional, event-driven model relies on a single simulation time value that is advanced for all components. If anything is supposed to take time in the simulation, the processes need to be stopped and the required events to wake them up again communicated to the kernel. These are then added to the event queue (and later have the kernel invoke them again). This process, if done repeatedly, takes time and thus slows down the simulation speed.

   TLM instead give designers the possibility to use functions as interfaces that model a transaction and simply store the time that is supposed to pass in a thread-specific variable. That means that instead of invoking the scheduler that manages the consecutive execution of SystemC threads, functions in different modules call each other and invoke the required actions, each time annotating that simulation time should have passed (but did not yet) and at some point wait for the simulation kernel to "catch up". This results in an even better performance, although certain concepts (such as pipelining) cannot be translated into these transactions.

2. *The Generic Payload* is a generic transaction class that serves as a common foundation to transfer information. Instead of having different types for each port (such as `boolean` or `integer`), the `tlm_generic_payload` class enables designers to use a single type for various transactions, storing both data and meta information for a given transaction. This results in better interoperability, allowing modules to be interconnected easier – thus eliminating the need for wrapper modules in a lot of cases and again speeding up both, development and simulation time.

The advantages of the design approach on the ESL and the features of SystemC in particular have turned SystemC into a "de-facto standard" [7]. SystemC is by now widely used, with the reference implementation being available under an open source license. While other ESL solutions are available as well, the adaptation of SystemC is far ahead of any contenders. This is the main reason for the focus of this work: in order to ensure a scientific (and maybe industrial) relevance, this work focuses on SystemC. While other solutions for design on the

ESL may not share the same set of issues that need to be solved, the degree of usage of SystemC justifies solving these using the given platform.

3 EXTRACTION OF SYSTEMC META DATA

Names and attributes must be accommodated to the essence of things, and not the essence to the names, since things come first and names afterwards.

Galileo Galilei

SystemC is implemented as a C++ library. This means that a design implementation may use any C++ statements to describe both, the system's structure and behaviour. Any C++ libraries can be used to add functionality to a design, vastly simplifying the design process especially for complex tasks that have been implemented in software before. However, relying purely on C++ comes at a price as well: High level software languages such as Java or C# include frameworks for introspection and reflection of running programs. The former allows the designer to write code that inspects the structures that were created by the running program, the latter allows the designer to write code that modifies those structures. E.g. in Java, retrieving an object's type, getting a list of its field variables and invoking one of its methods can be achieved using a few lines of code that utilize the reflection framework. In C++, this behaviour is not supported.

When compiling a C++ program, the compiler translates the program into machine code, stripping it of all unnecessary information (such as information about an object's fields' names and types), thereby making it impossible to simply retrieve any information about the structures that are present in a running program. However, as SystemC relies on C++ to create the simulated system in the first place, this run-time retrieval would be needed for e.g. a generic visualization tool.

Simply put, the lack of native reflection/introspection in C++ renders the extraction of any meta information about a SystemC design a far-from-trivial problem. This chapter focuses on approaches to bypass this shortcoming, presenting an approach to gather as much data as possible to enable design understanding techniques for SystemC.

The remainder of this chapter gives an overview of existing approaches to the extraction of SystemC features in Section 3.1, introduces a novel and unintrusive approach in Section 3.2 and gives details about its implementation in Sections 3.3 and 3.4 before concluding in Section 3.5.

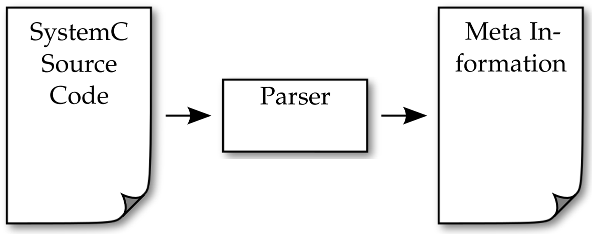


Figure 7: Parsing SystemC

STATE OF THE ART

Due to the lack of native support for reflection and introspection, researchers developed several approaches concerning the extraction of information from a given SystemC design.

A common approach to the extraction issue was to parse the given SystemC source code in order to extract the needed information. Figure 7 illustrates the corresponding procedure: As the source code does not have to be executed, the analysis builds a model from reading the source code alone. Several dedicated SystemC parsers have been implemented, most of them being open and available for usage.

Approaches using parsers to extract SystemC design information

- ParSyC [31] is a SystemC parser based on the Purdue Compiler Construction Tool Set (PCCTS) [85], the predecessor to ANother Tool for Language Recognition (ANTLR) [84]. ParSyC has been developed at the University of Bremen, the source code has not been released publicly yet. ParSyC translates the SystemC description into an Abstract Syntax Tree (AST), builds an intermediate representation from this AST which can be checked for semantic consistency and finally synthesizes this intermediate result to a netlist. This last step limits the parser to a synthesizeable subset of C++/SystemC, limiting the available code elements to the intersection of the C++ constructs the parser can understand and those that can be synthesized.

- The Karlsruhe SystemC Parser Suite (KaSCPar) [48] is another SystemC parser, created using the Java Compiler Compiler (JavaCC) and the according preprocessor *JJTree*. As being a user-created parser for SystemC/C++, it has the same problems as ParSyC concerning portability: KaSCPar does not support the whole C++ standard, so compiler-specific additions to the standard library are usually unsupported. The same is true for libraries or anything else that does not have its source code embedded into the project. This renders several patterns for SystemC development unavailable.

- SystemC to Verilog Synthesizable Subset Translator (sc2v) [14] is a tool that does not primarily analyse a SystemC design but focuses on the translation from SystemC to Verilog, but as a detailed SystemC analysis is a prerequisite to properly translate a given design to Verilog, it fits well into this list. As the primary purpose of this tool is the translation into Verilog, it limits itself to the analysis of a synthesizeable subset of SystemC. While this is a reasonable decision for a translation tool, it is limiting the analysis capabilities.

- SystemCXML [4] uses doxygen [114] as a foundation for its code analysis. Using an existing solution instead of writing a custom C++ parser certainly helps this approach in the analysis process as C++ constructs are not as limited as a custom solution (that may be incomplete). However, SystemCXML is limited to extracting the behavioural properties of the code from the generated doxygen files, basically limiting it to e.g. a list of module instanciations. The interpretation of conditional statements such as loops only results in an according syntax tree. SystemCXML does not generate the according structures, limiting its application to e.g. visualizing the source code instead of the system itself.
- SystemPerl [98] is a collection of Perl scripts that parse SystemC using regular expressions. In order to properly parse the given source code, SystemPerl requires the designer "to provide hints in the program for the preprocessor to identify the constructs to be expanded" [63], meaning that the original source code needs to be modified and prepared for the tool to be able to properly interpret it.
- Another (unnamed) parser, based on using Flex/Bison for lexical and syntactical analysis was introduced in [11]. This tool relies on processing the code with a C parser after its analysis though, which will make it difficult to adopt it to handling the full C++ language.
- Scoot [8] focuses on generating a formal model from SystemC designs. Just like the other parsers, it relies on a purely static analysis of a given design. In order to make the parsing process more feasible, scoot requires the usage of a customized collection of SystemC header files, which "declare only relevant aspects of the API" [8]. Scoot also uses a modified scheduler which performs better than the standard scheduler that comes with the SystemC kernel. Interestingly, it also provides a way to translate the analysed design back into a C++ project that does not depend on the SystemC library anymore, resulting (together with the scheduler modifications) in a better simulation performance. However, as scoot has certain limitations concerning the available language constructs and even modifies the SystemC kernel, utilizing it for an existing

project may require some heavy refactoring of the given source code, which might not always be possible.

All these parsers have in common that, by definition, they parse the source code and use this result to extract the system that was described in there.

Advantages of parsers

The advantage of this approach is that the whole analysis process remains in the hand of a single tool. Also, the static approach keeps the architecture simple as no further execution of a given design is needed in order to extract a synthesizeable description.

However, while the static parsing approach works well for designs that describe one single, static system, this approach has some serious drawbacks when it comes to more dynamic structures. Parsers, by definition, do not execute a given system. This quickly becomes a problem when taking SystemC's notion of design setup into account where the design itself is created by executing the according C++ code during the elaboration phase. Structures may be created by simply lining up calls to module constructors which can easily be parsed. However, elaborate designs can be realized by creating structures in loops or recursions, creating lots of instances with just a few lines of code. Creation can also be nested, with submodules being created within other modules, maybe even hidden within macros or other code elements that may or may not be active depending on how the compiler's preprocessor modifies the code. A design may even be created using e.g. user-defined input values, letting the designer specify e.g. the number of cores of a simulated CPU or the amount of cache available to a system before running the previously compiled executable file. Naturally, parsers are unable to retrieve the correct design in this case.

Limitations of parsers.

Overall, parsers have serious issues when it comes to analysing complex designs. At least the elaboration phase needs to be executed in order to extract a particular system design from a SystemC implementation.

Hybrid approaches

Hybrid approaches expand the parsing approach to extract a SystemC design from its source code. First, just like the parsers discussed above, they apply a static analysis of the source code to extract e.g. the structure of a given module. Second, however, they solve the problem of analysing the system's elaboration phase by executing the program at least until the start of the simulation.

However, while the general idea of this approach seems straightforward, extracting information about the structures inside a running C++ program is not easy. Compilers usually discard all the information that is not needed to execute a given program. This includes information about the created objects and their structure – in order to execute a C++ program, the program itself does not need the information that an object contains a field called `m_internal_state`, it just needs to know that

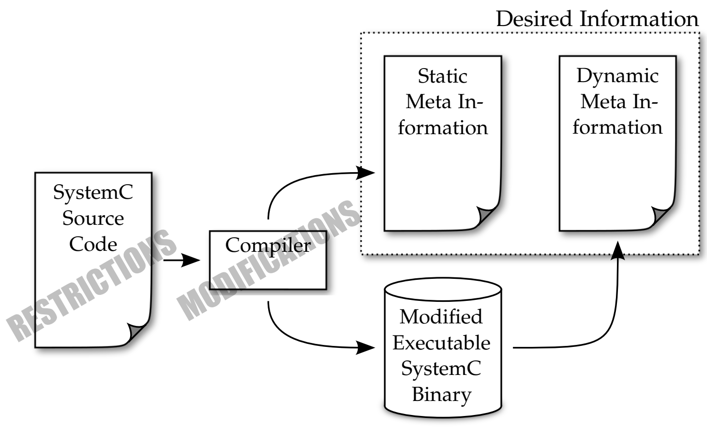


Figure 8: Hybrid approaches

a value at address *n* needs to be changed, even disregarding the value's type.

As the compiler removes the unnecessary information during compilation, it is the one element in the workflow that has access to the information in the first place. Hybrid approaches use this fact to extract the needed data before it is discarded by the compiler by modifying the compilation workflow in a way that lets them extract the information or prevent it from being removed in the first place. Basically, the idea is simple: If the compiler gets rid of the information that is needed, it needs to be modified not to do this anymore.

Separation of static and dynamic information extraction

The compiler needs to analyse the program's static structure anyway in order to compile it. This means that any structural information about the source code is present in the compiler and just needs to be extracted. Any constructs that are usually hard to parse (such as preprocessor directives) have been properly processed by the time the compiler has analysed the program structure. Problems therefore either do not arise from the language constructs being used or would also lead to compilation problems when trying to compile the program in order to simulate the system.

The dynamic information (e.g. what modules have been created in the elaboration phase) cannot be extracted at compile-time as the elaboration phase needs to be executed for this information to be present at all. However, as the compiler is the one that translates the program into an executable binary, it can just as well modify this binary to track and later store the information about the system being set up.

Figure 8 shows how this static analysis of the program's source code combined with the ability to track the dynamic creation of objects at runtime by modifying the resulting binary file results in a detailed model of the given design. First, the SystemC source code is processed by the compiler. Usually, this compiler would just translate the code into an executable file that could then be run. However, the compiler may be configured or altered to emit and store intermediate results of processing the code. The results of this static code analysis (i.e. the static meta information) are the first part of the information needed to retrieve the information present in a SystemC design. Second, the executable file the compiler builds is modified in order to emit and store information that is crucial for retrieving the structures from the running application that cannot be retrieved statically. These two sets of retrieved information represent a model of the design, which is the information that was desired in the first place. There are a few implementations that follow this idea:

Hybrid extraction approaches

- Pinapa [77] relies on the GNU Compiler Collection (gcc) to analyse SystemC designs. The main advantage of the approach taken is that the limitations of traditional parsers using their own C++ grammar are overcome by utilizing an off-the-shelf front end instead. This idea, combined with the approach of executing the compiled elaboration phase to retrieve the dynamic instances that describe the actual system, makes Pinapa a much more robust tool than the parsing approaches outlined above. However, in order to extract the data, it uses a "slightly modified version of SystemC" and requires "a patch to the GNU C++ compiler" [78]. As both, the SystemC kernel *and* the compiler need to be modified for Pinapa to work, this solution is not portable: projects either need to use the given compiler or are simply unsupported. The SystemC kernel modifications restrict projects to the SystemC reference implementation and even then problems may arise if the given project needed some modifications to the given SystemC kernel by itself, requiring the two SystemC kernels to be merged. Also, as SystemC itself may be updated, this approach requires a constant maintenance to keep up with the changes of the underlying SystemC library.

- A development that improves the approach taken by ParSyC with a dynamic execution of the elaboration phase was also suggested [36]. While this approach solves the parser's problem of not being able to fully extract the elaboration phase's result, the issue of relying on custom parsers for the code interpretation is still present in this approach. Unlike the other dynamic approaches, it does not use an off-the-shelf front end for C++, thus limiting the available language features – which in turn might result in the need for serious re-writing of the code.

- PinaVM [62] is the successor of Pinapa. Instead of using a patched gcc, PinaVM relies on using the LLVM compiler framework via its Application Programming Interface (API). Instead of using e.g. a modified version of gcc (like Pinapa), PinaVM relies on using

LLVM like a library to handle the given code base. While this has the advantage of being more compatible with version changes ("Although its API is not fully stable, it is clean, and the migration from a version of LLVM to another is a painless task" [62]), it comes at the price of being dependent on what this API offers. As a result of working on LLVM, PinaVM heavily relies on handling the intermediate representation LLVM builds from the source code, the so-called bitcode. This assembler-like language is much less abstract than the original C++ code, resulting in each original statement usually being translated into several new statements, which makes the mapping from the used representation to the original source code a non-trivial task. Also, PinaVM, just like Pinapa and SHaBE (see below) relies on one single front end. As the method itself cannot be transferred to other front ends, this makes this a powerful but restricted solution: projects that use code that are incompatible to LLVM are not supported by PinaVM either.

- SystemC Hierarchy and Behavior Extractor (SHaBE) [12] uses a different approach that relies on a debugger instead. While both, Pinapa and PinaVM are modifying the compilation process itself, SHaBE instead uses a combination of the gcc and the corresponding GNU Debugger (gdb), utilizing the latter to stop the execution of a running SystemC program and extract the dynamic information from the debugger. Using a predefined set of breakpoints, SHaBE tracks e.g. the creation of SystemC modules by stopping the execution in the constructor of the `sc_module` class and inspecting the call stack to get the inheritance hierarchy. At this point, an object's fields can also be retrieved, allowing SHaBE to also extract an object's static features. To retrieve information about the system's behaviour, SHaBE goes a way similar to Pinapa. It hooks into the compiler to retrieve the program's abstract syntax tree, which contains detailed information about the system's functions and their interaction. Instead of modifying the gcc, a plugin is used though, allowing SHaBE to be more robust concerning changes of the underlying compiler, assuming that the plugin API does not change frequently. Just like Pinapa and PinaVM, this ties this approach to one specific compiler though. Not only is the plugin written for gcc but other compilers simply do not offer a corresponding architecture. Thus, this approach is not portable and relies heavily on the project being interoperable with the chosen architecture. This even excludes projects that rely on workflows based on older versions of the gcc, with the plugin API being a rather recent addition to the project.

- Another approach that uses a debugger is presented in [93]. Although the focus of this approach lies on the actual debugging of the system, it still handles the same issues to inspect the system in the first place (i.e. the extraction of the given system's properties). Like SHaBE, this approach uses gdb to extract the data from the running program and therefore suffers the same lock-in issues as the former. While this tool comes with its own visualization to control the debugging environment, it is using a proprietary engine by Concept Engineering (a company based in Freiburg) to do so, thus limiting the availability of the system.

While these approaches differ slightly, they share the same features of a dynamic extraction.

Basically, the idea that a compiler has access to all structures and can be used to modify the output in any way is sound. A design that is handled using either of these tools can be analysed well: both, static and dynamic structures are accessible to the tool and may be extracted, so the output of these methods is thorough.

Hybrid approaches are tightly linked to a specific compiler

However, this information retrieval approach does still come with a trade-off. The solutions are tightly intertwined with the compiler that they are based on. Either because the compiler itself is modified (as in Pinapa [77]) or because a plugin specific to that compiler needs to be used during compilation (as in SHaBE [12]). This implies that no other setup may be used in order for the respective implementation to be applied.

The impact of this fact differs depending on the code base. Projects that strictly stick to the C++ standard [81] should be portable enough. As the build process differs from compiler to compiler, setting up the build environment for a new compiler is usually a cumbersome but still manageable task.

Language dialects therefore are an issue

However, C++ comes with dialects and libraries that are not standardized and which jeopardize the application of this approach in other build environments. Different environments have access to different libraries and tools. Microsoft e.g. offers several extensions to the standard C++ library that cannot simply be ported to other build environments [73]. The problem of dialects even exists within the same environment: updating a compiler may break the compilation for some source constructs.

This is a problem for this approach, as the number of available compilers is currently limited to those that can be modified through plug-ins or source modifications, with closed-source compilers probably remaining unsupported due to the missing ability to add features at will. The solution to port an existing project to a supported build environment implies that a potentially large part of the code base needs to be rewritten. This is a time-consuming and therefore expensive task that should be avoided if possible, especially considering that data extraction methods are usually supposed to assist the designer instead of causing him even more work.

The hybrid approach therefore comes with the most promising, yet also quite limiting notion of either supporting a given SystemC project and being able to export the full design down to the abstract syntax tree or not supporting it at all.

Overall, existing approaches either suffer from

- being focused on static aspects only,
- the need to use a customized adaption of SystemC and, therefore, an inapplicability e.g. for future releases of SystemC and/or combination with other methods that modify SystemC as well, or
- a dependency on customized compilers and/or parsers which limits the usable language constructs.

DEBUG SYMBOL PARSING

In order to address the research question, the data and interfaces that are available in SystemC/C++ designs are exerted as much as possible. In particular,

- debug symbols (that are generated by practically all compilers, albeit using differing standards) from which relevant (static) information of the considered design can be obtained and
- the SystemC API that allows for an extraction e.g. of values from data-structures during the execution of a program

are being used.

In this Section, an approach is proposed which exploits these existing interfaces for a generic and flexible information extraction of SystemC designs. More precisely, the debug symbols that are generated during compilation anyway are used to extract the static meta information of SystemC programs. Following this approach, modifying a compiler (e.g. to dump the accumulated information) can be avoided as the desired information can also be extracted from these symbols. Figure 9 illustrates the proposed workflow.

Dynamic information can obviously be retrieved only during the execution of a program as some values might not be known at compile-time. But instead of trying to retrieve this information by modifying the given design (e.g. to dump values currently assigned to signals) or by deduction from the static information, the existing infrastructure is used. In fact, the SystemC API is exploited to extract the desired information during run-time, making the solution for dynamic extraction independent from the platform or the compiler being used.

Instead of modifying the compiler and working with a binary extended by additional information needed for data extraction only, the

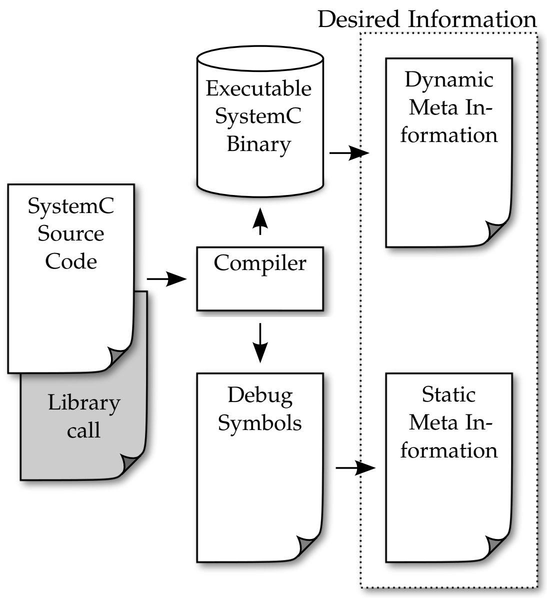


Figure 9: The SystemC build process.

existing data structures and interfaces are exploited (namely the debug symbols and the SystemC API). By relying on this existing infrastructure, the proposed solution is flexible, quite independent from compiler versions, and fully supports the whole range of SystemC.

In the next section, the proposed approach is described in detail. Two different modules implement the ideas proposed above: The first module reads the compiler-generated debug symbols and extracts static information of the design from it. The second module is a C++ library which can be called during run-time to export SystemC objects that are currently residing in memory. This allows for an extraction of the dynamic information. Afterwards, the extracted dynamic information is matched with its static counterpart.

EXTRACTION OF STATIC INFORMATION VIA DEBUG SYMBOLS

Existing approaches for the extraction of static information rely on parsing the code using a custom program or interpreting the intermediate language of a given compiler. In the proposed solution, debug symbols of the compiled program are exploited instead. Such debug symbols are created by almost every modern compiler and contain meta information of the code. This data is usually used to aid the designer in developing and debugging his/her implementation. For this purpose, the compilers collect and build extensive data about the program which, after the compilation, is written to the disk so that the debugger can use

```
.stabs "fullAdder:Tt(0,4872)=s5332!1,020,(0,1927);a:(0,1979),736,448;b
    :(0,1979),1184,448;carryIn:(0,1979),1632,448;result:(0,3933)
    ,2080,480;carryOut:(0,3933),2560,480;  ...
```

Figure 10: Debug symbols in the STABS format

them. However, in a similar fashion, the desired static information can be extracted from these symbols.

Example 2 *Figure 10 shows some debug symbols in the STABS format as they have been generated by the GNU compiler using the example program introduced in Figure 6. As can been seen, relevant static information of the design (e.g. the* fullAdder *class and its fields) can be recognized. This kind of information is available to a very deep level, including access modifiers of fields, a function's lines in the source code, function parameter types, base class information, size of types, etc. Furthermore, many debug symbols are arranged in a hierarchical manner, i.e. a debug symbol may be composed of several subsymbols.*

The information available through these debug symbols can be exploited to extract the desired static information of a given system.

Two tools have been developed to illustrate how the debug symbols can serve as a source of information:

Reading debug data files

The first tool uses the Microsoft VC++ compiler and the *Program Database*-files (PDB-files) generated by it [72]. Although the format itself is proprietary, using the *Debug Interface Access* (DIA) SDK [71], the debug symbols that are collected in those files can be accessed.

The second tool is based on the DWARF debug symbols [27] and the libdwarf/dwarfdump tools to read them. These open tools provide a more streamlined access to the DWARF debug symbols that can be generated by both, gcc [101] and clang [53]. More specifically, the output generated by dwarfdump is passed on to the tool that translates the structures into the needed format.

Using these tools, the data from debug symbols can be obtained on all platforms and all major compilers. To the best of my knowledge, there is no compiler that does not generate the needed information in the form of debug symbols and stores them for later usage, making this approach unversally applicable.

Figure 11 shows the procedure of the extraction for the Microsoft data. Given a PDB-file created by the VC++ compiler from a SystemC design, all topmost debug symbols are loaded first (line 2). Afterwards, each symbol is separately considered and analyzed (lines 3-5). The desired information is thereby dumped into an XML-data structure representing the static information of the system. Since the debug symbols are hierarchically structured, the analysis is recursively conducted through the function *analyzeSymbol* (line 8). Here, all the desired information of the

```
1   function analyzeDebugData(filename) begin
2           symbolTable = loadDataFromDebugFile(filename)
3           for each symbol in symbolTable
4                   analyzeSymbol(symbol)
5           end for
6   end function
7
8   function analyzeSymbol(currentSymbol) begin
9           dumpAllData(currentSymbol)
10          if currentSymbol has typeInformation then
11                  analyzeSymbol(typeInformation)
12          end if
13          if currentSymbol has subSymbols AND
14                  currentSymbol is NOT baseClass AND
15                  currentSymbol is NOT typedef then
16                  for each subsymbol in subsymbols
17                          analyzeSymbol(subsymbol)
18                  end for
19          end if
20  end function
```

Figure 11: Pseudo code of debug information extraction

currently considered debug symbol are dumped to the XML-data structure first (line 9). Afterwards, it is checked whether further hierarchical information is available (lines 10-19). If this is the case, the corresponding sub-symbols are analyzed by recursively calling *analyzeSymbol* for them. To avoid redundancies, lines 14 and 15 stop the recursion if types are found that are also part of the *symbolTable* and would otherwise be extracted several times.

Example 3 *Consider again the SystemC code from Figure 6 to be analyzed. Using the PDB-file generated by the VC++ compiler, the analysis of the first debug symbol (line 8) results in an XML-tag like*

```
<userDefinedType name="fullAdder" addressOffset="0"
addressSection="0" constType="0" length="428">
```

stating that the considered system contains the class fullAdder *(an instance of which occupies 428 bytes in memory). More information about this class can be gained by the analysis of the corresponding sub-symbols through the recursive calls (line 10-18). One of the* fullAdder *class's sub-symbols contains e.g. information about its field "a" (i.e. the full adder's first input bit):*

```
<data name="a" [...] >
  <type>
    <userDefinedType name="sc_core::sc_in\&lt;bool\&gt;"}[...]>
```

Note the field's name ("a") and the field's type (`sc_core::sc_in<bool>` *with "<" being replaced by "&lt;" and ">" by "&gt;" respectively to keep the XML structure valid) in the description. This information is contained in its own debug symbol that is part of the former symbol. The hierarchy is encompassed by the recursion. More information could be gained by searching for the description of the given type itself. Other sub-symbols provide information e.g. on inheritance, functions, their parameters, etc.*

The DWARF format, on the other hand, does not need the recursion step that is required for the PDB file analysis (and shown in Figure 11). Where the interface to read the PDB data is handing out references to a given sub-symbol, the DWARF data is rather flat, with links between symbols being stored in the form of unique identifiers that can just be ported either as such or in the form of target names.

Applicability

Using any of these procedures allows for an exploitation of debug symbols, which are generated anyway, for the purpose of static information extraction. Compared to previously proposed solutions with their respective constraints and requirements, this results in a near-to-none setup as all the information is retrieved from existing compilers and tools. The whole extraction step can be embedded transparently into the compilation workflow and does not require any human interaction after its initial setup.

Limited to the information the compiler stores

While the advantages of this method (being applicable to any setup, being portable to any platform and being usable, as it doesn't require any interaction beyond the initial setup) are clear, it is limited to the information that is stored by the compiler in the debug symbols. This means that usually, the static information does not contain the full abstract syntax tree. In most cases, the information is structural only, meaning that e.g. the information that a certain class has a function is stored but not the contents of said function.

This means that the advantages come with a price tag and the retrieved information, while being vast, is not complete. In the end, this is a tradeoff that the designer needs to consider when chosing a method to analyse SystemC code.

Especially for the use case of Design Understanding, the advantages of the proposed approach far outweigh the information that cannot be retrieved though. As this use case does not require complete information but has its focus on the quick and simple retrieval of more abstract information present in a design, this new approach suits this aim well by being simple and unversal to apply. Basically, the usability is the point the given approach excels in, as the required steps to use it are kept to a minimum, making it suitable for a quick glance into a given system on a more abstract level than the source code could provide.

EXTRACTION OF DYNAMIC INFORMATION

The SystemC library comes with an API that provides not only means of virtually creating and simulating systems, but also allows for accessing and inspecting the created instances of a system during run-time. That is, SystemC itself is, in principle, able to deliver an overview of the dynamic information of the instantiated system. Existing approaches exploiting this API for the extraction of dynamic information still rely on

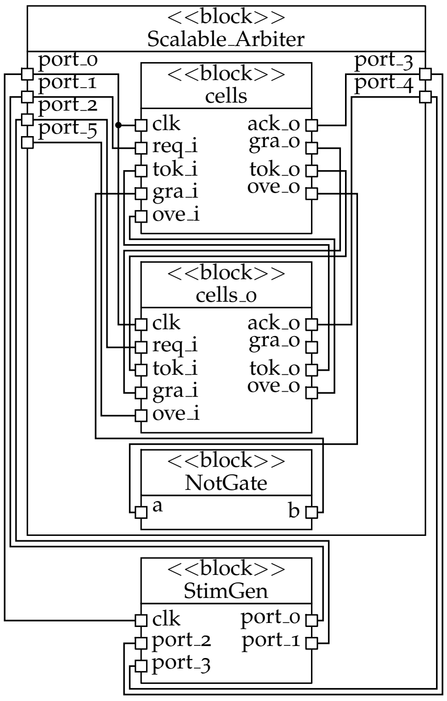


Figure 12: Diagram extracted using the SystemC API from the arbiter example [23].

modifying the SystemC library [40] and, hence, only provide a limited and restricted solution.

The proposed solution requires as few changes as possible to the existing setups by performing the following steps each time dynamic information should be retrieved:

ACCESSING THE INSTANTIATED OBJECTS The SystemC API provides a function to get access to the simulation context (via `sc_get_curr_simcontext()`) through which an `object_manager` instance can be retrieved (via `context->get_object_manager()`). The `object_manager` in turn provides access to all instantiated SystemC-objects that are being used in the current run of the SystemC program. This means that objects that do not implement the `sc_object` class (and are thus not managed by the SystemC simulation kernel) cannot be retrieved via this method, but all objects that are functioning as stand-ins e.g. for a hardware module or a signal can be. Extracting the objects that are available using this method returns the information illustrated in Figure 12: Instanciated modules, their ports and signals are all retrieved.

NAMING THE RETRIEVED OBJECTS To name the retrieved objects, [40] named the instances based on the fields' names. That is, an object created by the SystemC line `fullAdder faField("faName")` would be named *faField*. While this approach seems obvious at first sight, this leads to serious problems as

- field names may be used more than once (at different locations in the program) and
- a single instance may be assigned to several fields and, hence, a single instance might be referred to by several different field names.

In order to overcome these problems, the proposed solution uses the respective SystemC name field for naming an instantiated object. That is, an object created by the SystemC line `fullAdder faField("faName")` would be named *faName*. As SystemC automatically renames duplicates and assigns names to unnamed objects, this solves the above mentioned problems. Moreover, as the name is chosen by the designer and is not required to comply with the rules of C++ variable names, the naming of the respective objects becomes more intuitive for the designer. Finally, this solution makes any parsing of the SystemC source code obsolete. Limiting modifications in the SystemC library (as done e.g. in [40]) can be dropped.

MAPPING INSTANCES In order to retrieve a complete model of the given design, in a last step the extracted dynamic information is mapped to the static information gathered by the debug symbols[1] and/or other objects that are being used in the design. To comply with the non-invasiveness principle, the C++ and SystemC APIs are exploited:

- The types of the modules are retrieved by *Run Time Type Inspection* (RTTI) using the `typeid` operator (via `typeid(*module).name()`). The RTTI information is only available for objects that have virtual methods and therefore need a virtual function table (`vtable`) that stores the according information. However, as SystemC objects already have virtual methods, all SystemC objects provide a `vtable` during run-time, making the use of RTTI reasonable. This type name is the same as in the debug symbols, mapping the instances to their classes.
- Channels are differentiated from modules using an attempted cast to `sc_interface*` (via `dynamic_cast<sc_interface*>(module) != 0`). This allows for channels to be marked as such during extraction albeit they usually behave like modules.
- Ports are matched either by the name of their channel (when being connected to one) (via `(dynamic_cast<sc_channel*>(chan))->name()`) or the memory address of the respective signal (via `reinterpret_cast<int>(chan)`). All ports that share the same

1 Note that e.g. [40] did not retrieve any static information and, hence, no such mapping at all was considered there.

signal address are considered to be connected depending on being either input, output, or both. In contrast, ports that share the same channel instance's name are considered to be connected to the given channel.

The type names that are extracted by the SystemC API in this fashion exactly match those that are extracted from the debug symbols. Hence, the remaining mapping between the extracted static information and the extracted dynamic information is a simple comparison operation.

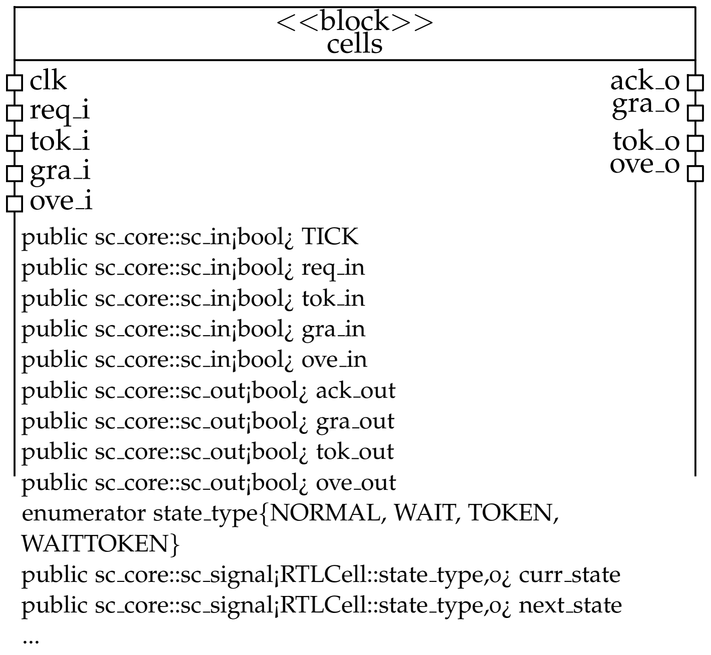
Incorporating these steps, the designer only has to specify at which point during the execution of the project the dynamic information should be extracted. This is realized by providing a function `dumpModulesToFile(string filename)`. Whenever this function is called during the execution of a SystemC program, the respective steps from above are performed and, similar to the extraction of the static analysis, the determined information is stored in an XML-data-structure.

Example 4 *Consider again the SystemC code from Figure 6. Assuming the designer is interested in all dynamic information available after the full adder has completely been instantiated. Then the designer only has to add the command* `dumpModulesToFile(string filename)` *after line 28 in the code from Figure 6. Then, during the execution of the program, the respective API calls are conducted, eventually leading to the XML-tags as partially shown in Figure 14. From this, the designer can obtain the desired information, e.g. that there is an object called "fullAdder_0" of the type "fullAdder" with two inputs "a" and "b", each of which is of type* `sc_core::sc_in<bool>`*.*

With the methods outlined above, a single, coherent model of the structure and the behavior of the ESL design can be extracted. However, SystemC models do not necessarily consist only of objects that inherit SystemC objects and represent e.g. modules or signals. Retrieving these non-SystemC objects is not possible using the SystemC API as they obviously are not handled by the SystemC kernel.

Due to the shortcomings of introspection and reflection in SystemC (as discussed above), the idea of using debug symbols proposed in chapters 3.3 and 3.4 needs to be expanded further to be able to extract these generic objects. More precisely, an information extraction flow as sketched in Figure 15 is applied.

First, static information on the SystemC implementation is determined exploiting the compiler-generated debug symbols (1). Afterwards, the original code is compiled so that the resulting binary can be executed (2). Both steps do not assume any changes and restrictions on the compiler. As discussed above, SystemC allows then to retrieve information of the instantiated instances at run-time. By additionally exploiting the static information, which is now available



<<block>>
cells
clk
req_i
tok_i
gra_i
ove_i
ack_o
gra_o
tok_o
ove_o
public sc_core::sc_in¡bool¿ TICK
public sc_core::sc_in¡bool¿ req_in
public sc_core::sc_in¡bool¿ tok_in
public sc_core::sc_in¡bool¿ gra_in
public sc_core::sc_in¡bool¿ ove_in
public sc_core::sc_out¡bool¿ ack_out
public sc_core::sc_out¡bool¿ gra_out
public sc_core::sc_out¡bool¿ tok_out
public sc_core::sc_out¡bool¿ ove_out
enumerator state_type{NORMAL, WAIT, TOKEN, WAITTOKEN}
public sc_core::sc_signal¡RTLCell::state_type,0¿ curr_state
public sc_core::sc_signal¡RTLCell::state_type,0¿ next_state
...

Figure 13: Retrieved arbiter cell with static information mapped to its dynamic instance.

```
<SystemCDesign>
 <module name="fullAdder_0" type="struct fullAdder">
  <In name="fullAdder.a"
    type="class sc_core::sc_in<bool>"></In>
  <In name="fullAdder.b"
    type="class sc_core::sc_in<bool>"></In>
   ...
```

Figure 14: Result of the dynamic design extraction of SystemC for the full adder.

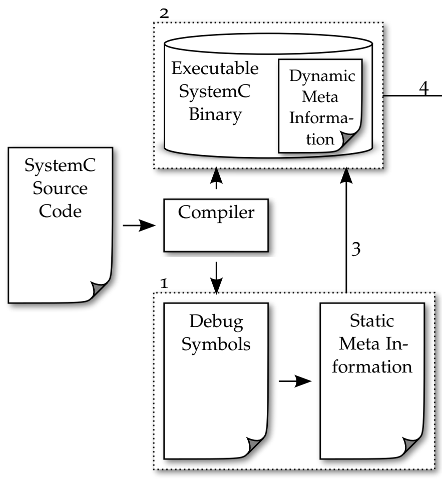


Figure 15: Proposed methodology.

through the debug symbols (3), the structure and current state of the executed system can be derived (4). This does not only include which classes/modules have been instantiated, but also the members these classes are composed of (including referenced objects) and which values they have been assigned with. During the execution of the SystemC implementation, several such "snapshots" of the system states can be conducted. From those, corresponding FSL descriptions (e.g. a block definition diagram of the structure and a sequence diagram including the respective object diagrams of the behavior) can be derived and eventually compared to the originally given FSL specification (5). For this purpose, existing modeling frameworks such as e.g. the *Eclipse Modeling Framework* can be utilized as they offer corresponding model checking capabilities out of the box. For this work, a simple checker was implemented as a proof-of-concept that asserts all parts of the specification are present in the implementation.

Especially step (4) expands upon and mixes the static/dynamic approaches outlined before.

In order to complete the "snapshot" of a currently considered program state, the respective assignments to each member have to be derived. This is accomplished by traversing the memory allocated by the considered SystemC execution. A breadth-first-search along a running program's references is applied, starting with the objects that were retrieved via the API. The information contained in the debug symbols is used to determine an individual object's memory layout. All fields are located and separated into

- base types (like integer, float, etc.) to extract their values and
- compound types and pointers, both of which are again enqueued for later analysis.

However, this traversal of allocated memory does have the following issues:

- Bad Pointers

  Null pointers are not analyzed further. A more serious problem are bad pointers. For example, Figure 16a shows an instance with a bad pointer. Although a field indicates that this pointer should not be used, this semantic distinction cannot be recognized automatically. Hence, the implementation assumes that all pointers are valid and catches memory access violations at run-time. Due to the read-only nature of the data extraction, the information stored in the memory is not compromised by this.

- Unreferenced Memory

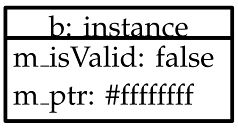
  There might be parts in memory that are allocated and used but not addressed. A common example is an array that is created on the heap and addressed using a pointer while a second value stores its size. As an example, see Figure 16b showing an instance with an array of size 4. In this case, only the referred memory is considered, i.e. only the graph that is formed by elements in memory and their connections is traversed. Elements that are separated from this graph cannot be reached by the traversal algorithm and, hence, are ignored. That is, in the example of Figure 16b `arr[0]` is extracted as a single int value (as both, pointer and type, are known), but `arr[1]` to `arr[3]` are ignored.

- Incorrectly Typed References

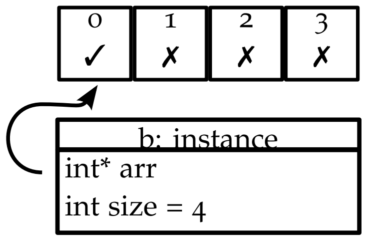
  Objects referenced using an erroneous type also pose a problem for the proposed extraction method. The approach assumes that the type of a pointer corresponds to the type of data that is actually stored at the given address. Due to C++'s lack of type safety, an instance of type a may be stored in a part of the memory that is referenced as being of a certain unrelated type b. The analysis algorithm will then assume that the address contains an instance of this type b and, therefore, extracts the data as if it would be of that type b. This may result in missing or incorrectly extracted data. For example, Figure 16c shows a pointer of type `char` that actually points to an instance that contains much more data. However, the approach treats this as a `char` and, hence only retrieves the first byte.

- Multiple Type References

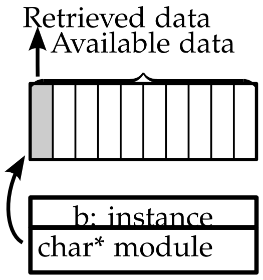
  Although similar to the incorrectly typed references, this is more a problem concerning established modeling languages: If a single address in memory is addressed by two or more differently typed



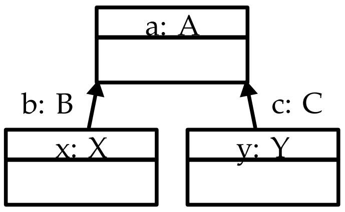
(a) Bad pointer: Although the pointer will not be accessed in the running program by checking for the *m_isValid* field first, the algorithm is unable to make this connection and will try to access the memory address currently stored in *m_ptr*.



(b) Unreferenced memory: While there are four integers allocated, the algorithm is unable to guess that the *size* variable specifies the length of an array located behind the address of the given integer pointer and will only retrieve the first element.



(c) Incorrectly typed references: A pointer might be typed correctly. In this example, although a pointer is allegedly referencing a `char`, it in fact references a `module`. However, as the algorithm assumes that the type specified is correct, it retrieves the first byte (depending on the char size of the given system), ignoring the rest of the allocated data.



(d) Multiple type references: Two references may be addressing the same area in memory but do so with different types (that may or may not be related). As the type defines how the data is to be interpreted, this leads to an ambiguous extraction. Currently, both interpretations are saved – which of the extractions is the "correct" interpretation cannot be determined though.

Figure 16: Special issues during program state extraction.

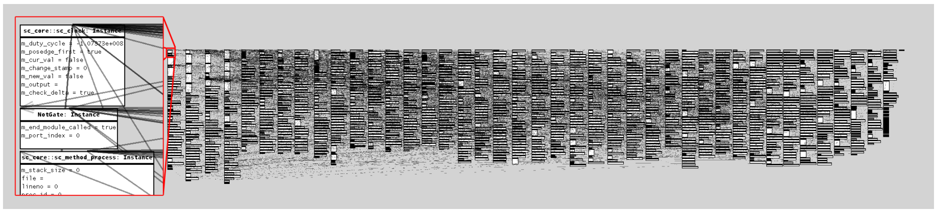


Figure 17: Resulting program state. The area on the left is a close-up to the upper-left corner of the diagram. While a manual comparison would be hard due to the amount of data that is retrieved, this data can be used as a foundation for an automated model-checking approach.

variables (as illustrated in Figure 16d), a corresponding representation is usually not available in established modeling languages. Currently, the approach just exports all interpretations of a certain variable. Depending on the desired modeling language, these features might need additional tweaking.

The result of this traversal is indicated in Figure 17 for the running example, i.e. the arbiter implementation[2]. The resulting model does not only contain the instances of the original, but also all instances of any object that is referenced through these (directly or indirectly) and the values that are currently assigned to the according member variables. Although this is much more data than actually needed, it now provides a precise "snapshot" of the current system state in terms of an FSL description. By sequentially applying this scheme for each system state which shall be compared against an FSL specification, all the information needed for this comparison is retrieved.

CONCLUSION

In this chapter, a new method to extract SystemC designs from their descriptions was proposed.

In contrast to previous approaches, this new method relies solely on data that can be extracted from

- debug symbols (which are generated by compilers anyway) and
- the unmodified SystemC kernel during runtime (which needs to be called anyway).

While other methods are limiting the available code elements in order to parse the code or can only be applied for specific (sometimes modified) compilers and/or SystemC implementations, the proposed method

2 Please note that this figure is not supposed to provide detailed information but serves as an illustration of the magnitude of the obtained information.

has been developed with applicability in mind and can be used on all major platforms without further restrictions. Additionally, the source code does not have to be altered at all, allowing the presented approach to be applied easily as it merely has to be integrated into the build process. It thereby provides a straightforward way to take a look at SystemC designs at virtually no cost.

Beyond, the method is able to extract field variable values at runtime for generic C++ instances. It may therefore also server as a substitute for C++'s missing introspection capabilities, with the potential of being applied outside the SystemC scope of application.

4 EXTRACTION OF SYSTEMC BEHAVIOUR

The only way to make sense out of change is to plunge into it, move with it, and join the dance.

Alan Watts [118]

The method described in Chapter 3 extracts information about a SystemC design not only from its source code but as well from a point during execution. As the elaboration phase is used to build the system, the intuitive approach is to wait for it to finish and then extract the design to be able to extract all modules. The result of this approach is a single model of the design, much like a description of a static hardware system.

However, during its execution, a SystemC program may be performing quite complex tasks. This especially includes the simulation phase where the system is being run, but also concerns the elaboration phase where arbitrarily complex schemes may be used to construct the design in the first place.

Due to SystemC's C++ foundation, analysing this program behaviour can be done using debuggers. These are used to stop the program at certain points in order to allow the designer to inspect the program's state and e.g. step through the execution of certain statements. Another approach, which is closer to the hardware aspect of SystemC, is to collect traces of SystemC's signals which can be stored in value change dumpfiles. These then contain a list of timestamps and signals that were altered at the respective times. However, the former's use case is limited to stopping the execution at one certain point to let the designer inspect the system and then continue while the latter is limited to simple variables and signals.

This Chapter illustrates methods that are able to retrieve data about a system's behaviour, allowing the designer to inspect his system's runtime properties in more detail. Section 4.1 illustrates approaches that utilize existing software analysis tools and highlights the issues that arise when these are used. Section 4.2 then outlines how the techniques illustrated in Chapter 3 can be exploited to extract behavioural information as well. Section 4.4 finally shows how *Aspect Oriented Programming* (*AOP*), an approach that refactors the source code before compilation to modify the behaviour of a program, can be used to insert more detailed extraction measures into existing projects before the Chapter is concluded in Section 4.5.

SOFTWARE-BASED APPROACHES

Due to SystemC being "merely" a C++ library, techniques and approaches that are used to analyse the behaviour of ordinary C++ programs can be applied to SystemC designs as well.

While these approaches usually have the disadvantage of ignoring certain cases that are special to SystemC designs (such as the division into elaboration and simulation phase and the rather static nature of `sc_object` instances that can neither be removed nor created after the simulation started), they do offer readily-available means to analyse a program. Two major approaches are outlined in this Section: *Debuggers* that are used to inspect the state of running programs and *Coverage* tools that are used to collect information about what parts of a program were executed in order to afterwards present this information to the designer.

Behaviour analysis using debuggers

DEBUGGERS are usually utilized to locate errors ("bugs") in a given program (hence the name). Programs compiled specifically in Debug-mode have their executable data mapped to the corresponding locations in the original source code file. This is the information that is exploited in the approach in Section 3.3 to retrieve the static program information. When such a program is executed by a debugger, it can pause the execution, use the embedded mapping information to locate the program's current location in the source code and display information about e.g. variable assignment or call stack.

A debugger is usually fit to perform its primary task though – it is supposed to aid a designer in locating the source of an error. The major use case is therefore to define a break point at some manually-set location in the program (that usually indicates that an error is about to occur or has already occured) and then manually see how this might have happened. While the designer may then step trough the program, he cannot go back in time, leaving him with the options of either being able to tell when an error is about to happen and place the breakpoints accordingly (which usually means that there is already a strong indication towards a source of a given error) or try and deduce how that error occured based purely on the current (erroneous) state of the program.

Some debuggers also support watching certain variables and breaking on changes (e.g. in *gdb*, designers can set watchpoints for certain variables or expressions, telling the debugger to stop the execution as soon as certain values change or are set to a certain value) or even executing arbitrary code upon certain conditions (which is usually used in tracepoints that e.g. log certain values as soon as certain conditions are met). *Gdb* in particular is scriptable to a large degree. It not only allows writing complex statements to conditionally stop, resume and supervise the execution of a program, it can load a predefined set of these commands as a script, allowing arbitrary programs to be analysed in a particular way during execution.

SHaBE, which was described in Section 3.1, is an example of this approach: while it does not analyse the behaviour of SystemC files but instead focuses on the extraction of the static design, it still uses this approach successfully. In SHaBE's case, the creation of new modules and other SystemC objects are recorded and traced. Conceptually, continuing to trace the program's execution and also focus on e.g. the modules' field assignments would be one way to use this approach to retrieve data about the system's behaviour.

Using debuggers is not a portable approach

Still, this comes with the drawback of technology lock-in. While SystemC has been implemented standard-compliantly, allowing it to be used on all major C++ platforms, *gdb* is limited to *gcc* and *clang* compiled programs. Even worse, the approach cannot even be transferred to all platforms as e.g. the Visual Studio debugger cannot be automated. Thus, using debuggers to retrieve information from running SystemC models cannot even be ported to at least one major platform conceptually. This is a serious drawback to this method as it would severely limit the applicability of any such approach.

All in all, debuggers are mainly tools to assist the designer in a *manual* process of analysing the behaviour or the given program. Their purpose is to offer a user a way to interact with the design at hand, not to give an interface for automatic extraction or interaction to other tools. Those debuggers that offer sophisticated interfaces to automate e.g. extraction and analysis methods are not portable.

In order to analyse the behaviour beyond giving a designer the ability to manually inspect what is going on at a specific point in time during the execution of a program, debuggers therefore do not seem to be the right choice. Instead, code-analysis tools that are not focused on single, specific points in time during the execution but instead focus on retrieving traces of whole programs should be considered in order to give the designers a broad access to the execution data of their SystemC designs.

Source code coverage tools to trace a design's behaviour

COVERAGE TOOLS focus on just that. Instead of stopping at specific points or tracing certain values of specific variables, they collect data concerning the execution of the whole program. The program is divided into *coverage items* (*CI*s) which are then instrumented during execution.

The point of coverage metrics is usually to provide a measurement as to how much of the program was actually run. This data is then used to locate elements that have not yet been executed and thus adapt existing or create new tests that cover all parts of the program.

There are different code coverage metrics available for C++ which can directly be applied to SystemC designs:

Existing coverage metrics for C++

- *Line Coverage* counts how often each individual line of source code has been executed. While it is readily available (gcc, clang and

MSVC++ all support instrumenting the code to retrieve line coverage), it suffers from the quite fundamental flaw that lines of code do not necessarily have any fixed relation to the underlying logic.

Even a simple for-loop such as `for (int i = 0; i < 4; i++)` contains three statements on a single line, each of which is executed a different number of times (or maybe even not at all, as a condition statement like `i < 0` would lead to the increment `i++` not being executed even once).

- *Statement Coverage* remedies this flaw by instead tracing the execution of each statement. However, while this would help concerning the example above by reporting `int i = 0` to be executed once, `i < 4` five times and `i++` four times, this does not mean that a program is error-free.

  Especially, there may be parts of the program that only cause errors if they are *not* executed as illustrated in Example 5.

  Example 5 *Consider the following pseudo-code program:*

  ```
  usertype* t;
  if (condition)
          t = new usertype();
  t.do();
  ```

  The variable `t` *is a pointer that is not set during initialization, thus pointing at an arbitrary point in memory that might still contain data. Only if the* `condition` *holds, it is initialized and set to a new instance, thus pointing at a valid address in space (instead of 0 as before). Hence, running the program with a setup that lets* `condition` *hold, this part will get 100 % statement coverage as all statements are executed while still containing the error that* `t.do();` *will result in undefined behaviour if* `condition` *should not hold.*

- *Branch Coverage* again remedies this particular issue by instead tracing whether or not (or how often) a particular *branch* was executed. In Example 5, two branches exist: one including the statement that initializes `t` and one that skips directly to the final method call `t.do()`. By tracing if each of these was executed, errors such as the one above can be located as getting 100 % branch coverage would mean that the erroneous branch needs to be taken.

  However, just as a single branch may be the source of a certain behaviour (such as an error), it can also be a combination of branches that need to be taken. Additionally, any error that does not result from a specific path that is taken through the program but from the data being passed through is inherently hard to discover: errors resulting e.g. from an integer overflow are not easily located using these coverage metrics. This means that even branch coverage is in no way indicating whether or not a program is error-free.

- While there are several other coverage metrics available such as *path coverage* that counts whether or not all different *combinations of branches* were taken (which is a problem with loops, as there may be an infinite number of different paths through a program), *loop coverage* that handles loops differently, *condition coverage* that tests for different outcomes on conditions instead of branches and others, the ones explained above are available in compiler suites such as *gcc*.

Basically, all available coverage metrics have the advantage on focusing on the analysis of a running program, which is needed for an analysis of a given SystemC design as well. However, they all suffer from the flaw of being very software-centric and usually being tied to the source code in one way or another. The applicability for a detailed run-time analysis of a SystemC design is hence limited for several reasons:

Issues with applying coverage metrics to SystemC designs

- *The results are aggregated.* Instead of tracing what happened when during the simulation, the gathered data is accumulated and reported as a single value per coverage item (regardless of it being a line, a statement or anything else). Instead of giving the designer the information that a certain part of the design was triggered at a particular point during the simulation, the final information is that a certain coverage item was triggered *n* times as a whole, which may not be helpful considering how a simulation may run for a quite long time with all kinds of scenarios being run.

  Coverage metrics always focus on giving the designer a single value that indicates how "complete" the test cases are, which may be helpful for fulfilling certain criteria but does not necessarily provide any insight about the structure and behaviour of a system. This foundational difference between coverage metrics and the goal of design understanding is a large issue for the application of coverage tools in the context of this work.

- *Value assignments are disregarded.* Especially for a simulated hardware environment, the information which parts were active at a certain point in time is just as important as *what kind of data* was handled by those parts. The established software coverage tools do not collect any information concerning which variables were set to which values, disregarding particular states of the running system for the sake of boiling down the runtime behaviour to a single completeness value.

- *They disregard ESL-specific features.* Coverage tools are based in the software domain and hence focus on the smallest common denominator of software projects, which is the source code. Dividing a system into control structures such as statements, branches or

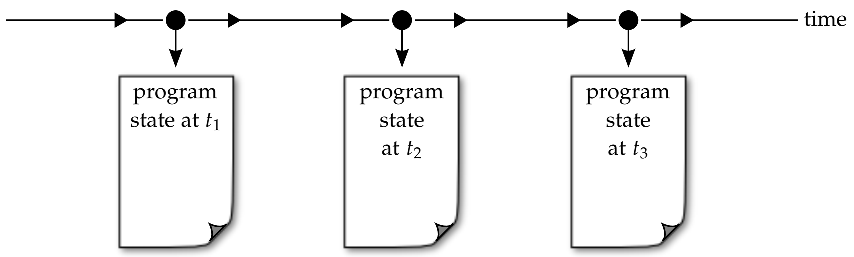


Figure 18: Extracting consecutive program states allows analysis of the system's behaviour.

paths may be a valid choice for defining CIs for plain software projects. SystemC on the other hand e.g. has a set of very static objects which are created during the elaboration phase and then required to persist until the end of the simulation. While these are a static part of the simulated hardware design, they are a dynamic part of the program and not treated any differently.

In conclusion, existing tools for software behaviour analysis have shortcomings that are limiting the usefulness of the given approaches for ESL designs. While their availability makes them a straightforward choice for a quick glance at the behaviour of a SystemC design, other methods that focus on the application on SystemC implementations and their peculiarities would be desirable.

The following Sections therefore propose two different approaches that are focusing on the extraction of more detailed SystemC run-time data in order to give the designer a better understanding of a given design.

CONSECUTIVE SNAPSHOT EXTRACTION

Extract multiple consecutive states

As the extraction approaches that are described in Chapter 3 offer various ways of extracting a system's state, there is a simple approach to behaviour extraction. The consecutive application of the method extracts a set of consecutive states that describe altering system states over time.

This approach, illustrated in Figure 18, works well.

The resulting list of snapshots reflects how the system changes over time. Due to the approach extracting not only SystemC-related data (such as signal assignments) but information about any other datatypes as well, these snapshots offer significantly more information than e.g. a signal trace as offered by the SystemC implementation. Also, the approach remains just as applicable, allowing the snapshots to be extracted without restrictions concerning the compiler or dialect being used.

The major issue of extracting these snapshots is not only the extraction itself once it has been triggered, it is (once the extraction has been implemented) the question of *when* to trigger this extraction. As the designer should know which part of a system needs supervision, invoking the extraction at the crucial points can be regarded as a manual task. This way, the designer would simply add the required code to arbitrary points in the source code, hence extracting the data at just the right points in time. However, if the designer does not want to specify the points in time at which the states should be extracted, an automatic method should be available. This basic requirement does not have an equally trivial solution though. The point during execution is neither trivial to pick nor easy to use as an extraction point, assuming that the approach should work automatically.

Triggering the extraction suffers from the same problem as the extraction itself: C++ does not offer an easy way to work with the program's implementation from within the program itself. Java e.g. offers not only its reflection package to gather information about the program and modify the state at run-time, it also has a well-defined intermediate representation, the Java Byte Code [5]. This common foundation can be used to alter programs on a fundamental level, rewriting the program's instructions to include features that have not been present in the original source code. C++ does not offer anything similar. The compiler's responsibility is the translation of source code into machine code, with the result differing depending on the target platform and no constraints concerning any intermediate steps to take. Hence, if the process should not remain a manual one, the question of how to pick a valid extraction point *and* how to automatically add the according logic to be executed at that point, is a a crucial one.

An intuitive approach would be to extract the snapshots after clock cycles. This way, the system's distinct timing scheme could be exploited to extract system states. Additionally, this would very well reflect the traditional way of tracing waveforms during the simulation of a system – simulating clock ticks and then logging how values change throughout the system is a method that has been used on RT level designs for quite some time. However, while this approach does seem obvious, it comes with several pitfalls that should be considered before applying such a pattern.

- SystemC designs do not neccessarily use clocks to synchronize their modules, so assuming that clock signals are available for triggering the extraction process might be presumptuous. SystemC designs may as well rely on manually set stimuli, with the needed time in between being hard-coded into the source.

- SystemC designs may contain an arbitrary amount of behaviour contained in a single clock cycle that may be of interest for analysing the given design.

  A SystemC module may contain any amount of logic. The code inside a module may be as complex as needed, with its execution not only covering what might later be a large part of a given design but a considerable amount of time as well. As a module may calculate a result and may be coded to e.g. wait a certain amount of time to pass it on to simulate that the calculation took longer than a single clock tick, this is not even a shortcoming of the framework itself. Instead, modules doing complex calculations in an instant (as far as the simulation kernel is concerned) is a result of how SystemC works and can be considered a feature of the framework. These calculations, however, could not be extracted in detail by a tickwise snapshot extraction.

- Depending on the simulation parameters, there might be long timeframes without any activity in the system. While such inactive phases could be handled by the appropriate diffing algorithm, executing the according code consecutively without any further information gain for an arbitrary amount of clock cycles may be considered an unneccessary performance hit.

With no definite point in time to extract the data, the issue would have to be directed at the designer, who could then manually insert extraction markers into the source code that would trigger the according method. However, this approach would contradict the core idea of a non-invasive analysis – after all, adding the according methods manually to the required points could be a major amount of work.

Generally, the extraction should be triggered automatically. As the executing program itself cannot log its own behaviour in detail though, the solution should be an external one. One approach could be to use the compilers to be able to access the program: modifying the program during compilation or using existing methods to alter the resulting executable in order to analyse its behaviour may be a way to insert the extraction methods into the running program at arbitrary points. Another viable approach could be the modification of the source code *before* compilation, altering it in a way that makes the final program to be compiled behave in the desired way.

COMPILER-BASED BEHAVIOUR MODIFICATION

As making a program analyse itself from within is an issue, one solution may be to instead insert the required tracing functionality on the compiler level. As the code has to be processed by the compiler anyway in order to be translated into an executable application, making the compiler add the according functionality at the appropriate insertion points is a straightforward way.

This approach requires some way to specify how the behaviour should actually be modified. Figure 19 illustrates the architecture of this approach: in addition to the SystemC source code, the compiler is also

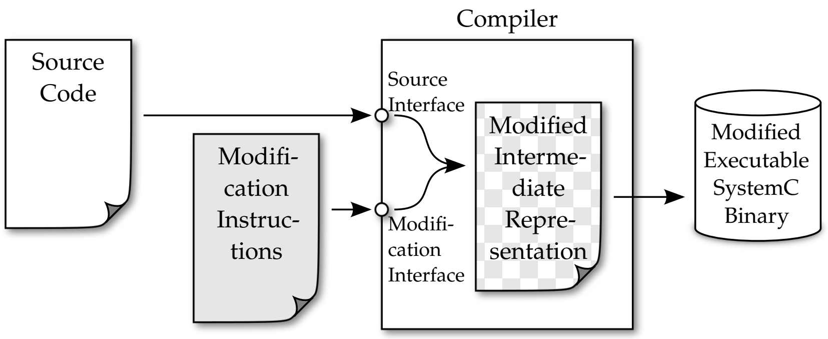

Compiler
Source Code
Modification Instructions
Source Interface
Modification Interface
Modified Intermediate Representation
Modified Executable SystemC Binary

Figure 19: Using the compiler to insert the analysis functionality requires some way to specify the required modifications.

instructed to somehow alter the resulting code, which then executes the required analysis while the program is running.

Avoid self-analysis of the application by modifying the behaviour at compile-time

The most important issue this approach fixes is that the program no longer needs to analyse itself using the means available via the standard C++ features. While C++ itself does not offer the ability to e.g. execute arbitrary code upon the change of a variable or the execution of a function, simply inserting the required functionality around these points is less of an issue. Being able to describe e.g. that with every function call inside a module, the call should be logged for later analysis would make the compiler insert the tracing functionality at each of these functions, resulting in extensive modifications during compilation but with no further requirements concerning the original source code.

Still, non-invasiveness in this case would not be an issue concerning the source code modifications but of the available platforms. Not all compilers offer an interface that allows altering the steps performed during compilation. While e.g. clang offers extensive plugin capabilities, gcc is more limited in this concern but is still an open-source solution, which allows the modification of the compiler's behaviour by patching its sources. However, any closed-source compiler that does offer any such plugin interfaces (such as e.g. the MSVC++ or Borland compilers) would be excluded from an approach that relies on the modification of the compilation workflow.

The similarity of this approach to the hybrid extraction approaches introduced in Chapter 3 is obvious, basically the major difference is the application: instead of extracting the static system models, the behaviour is supposed to be logged. As extracting the design already requires the analysis tool to analyse the run-time behaviour of the program (as modules are only created when it is running), extending these methods to somehow extract behavioural features is a logical next step. Still, the portability issues remain for any approaches that extend these ideas: something that requires or modifies a particular compiler requires the designs to be compatible to that compiler and the compilation workflow to use it. This may be a deal-breaker for projects that rely on different architectures and use their particularities.

In order to comply with the idea of non-invasiveness, these compiler-based approaches are therefore unsuitable. Thus, a different approach that has less requirements concerning the build environment of a given project is inspected in detail.

ASPECT-ORIENTED ANALYSIS INSERTION

Instead of raising awareness for a specific approache's issues, overcoming them should be a priority though. This section introduces the concept of Aspect Oriented Programming (AOP) and how it can be used to provide a solution to the issues of the previous approach.

The main goal is to overcome the limitations of previously proposed solutions, i.e. the static focus and, hence, missing support for dynamic descriptions. At the same time, the aim is keeping the proposed solution as non-intrusive as possible, preventing the designer from having to incorporate significant changes into their implementation just in order to analyse their design.

Differentiation between the extraction method and its application required

The general solution in this case is a division of labour between the method that analyses a design and its application. If existing solutions such as the statement extraction explained above or other generic software coverage methods provide a sufficient metric, it may still be an issue to actually use this metric for a given design. For the statement extraction approach, the issue was already outlined in section 4.2: while the retrieval method itself is sound, the question of where to insert the according code fragments is important and may come down to a manual process if certain constraints are not satisfied in the design. For software coverage metrics, despite them being readily available and applicable the question remains the same, especially for system designs that are merely written in software: assuming that the behaviour of a design should be analysed, describing not only how the data itself needs to be extracted (e.g. line coverage) but also at which intervals and when it should be done (e.g. for each function call, each clock cycle etc.) is a crucial part of the analysis.

Use AOP to define the points when the extraction method should be used

This division is accomplished by combining analysis metrics such as the extraction schemes from Chapter 3.3 or e.g. the *gcov* coverage tool for line coverage with the scheme of AOP [99]. The former provide the extraction method, i.e. they implement a certain means of extracting the required information from a running program. The latter is then used for that method's application, i.e. it determines the exact points in the running program when the chosen method should be invoked to actually retrieve the required data.

Existing coverage tools represent a valid foundation for behaviour analysis tools for SystemC, being directly applicable to its C++ foundation. Line coverage e.g. provides a per-line-of-source listing of parts of the program which have been executed in a given run. However, as outlined in section 4.1, simply applying corresponding methods for line coverage analysis such as *gcov* does not always provide a satisfactory result. Corresponding analyses are static, providing merely the information which coverage items (CIs) were triggered in a full run of a given design. Therefore, while indeed pinpointing to respective lines of code, e.g. the instance that is currently active and calling a given function (and thus the precise component of the design dynamically generated in the elaboration phase) remains unknown when using these tools, as CIs are limited to the static program elements, which excludes the dynamic structures that are created during the execution. As no further specification concerning the application of the given coverage metric is given (except that it is collected at all), a single, large dataset is generated for the elaboration and simulation phases of a system execution.

In order to address this, the insertion of another dimension when performing line coverage analysis is proposed. A valid extraction goal that is pursued as a proof-of-concept in this chapter is the collection of CIs *per module instance*. Instead of only providing the designer with the total number of times a single line has been executed during a run, the aim is to provide the designer with the total number of executions of a line for each module of a given system.

Application goal: extract coverage information per instance

Example 6 *Let's assume a system in which two components a and b are instantiated from a common base class CLS. Furthermore, let's assume that, during simulation, only component a is triggered. A static analysis would only unveil that the respective lines of the common code basis in CLS have been triggered n times. In contrast, having the total number of executions per instance would unveil that component a has been triggered n times while component b has not been triggered at all.*

Such a more elaborated result would lead to a much more accurate analysis of the system's behaviour. In the example, it gets obvious that component *a* triggered the coverage item (and might include the feature the designer is looking for) while component *b* seems irrelevant in this case.

Generally, the information is available in the methods that are being called: C++ functions that are members of a given class (such as an `sc_object`) are automatically supplied with a `this` parameter that can be accessed from within the given function and points at the object that is currently executing the function. This reference allows access to the instance of the module which actually executes the respective function. Using the `this`-pointer, one can track a unique identifier – e.g. the `name`-field of the respective SystemC-object – and thus keep track of which instance is currently executing a particular part of code.

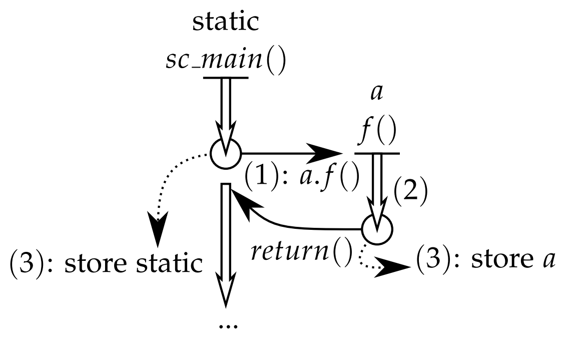


Figure 20: The proposed coverage analysis scheme.

Relying on these ideas, a scheme to extract the desired features is outlined in Figure 20. Whenever a function is called (1), a common code line analysis tool (e.g. *gcov*) is additionally invoked. This keeps track of which lines are triggered when running the function (2) – as in any existing feature localization tools. However, the tracked information is not stored globally, but with respect to a unique identifier of the instance which runs the function. For this purpose, the execution of the analyzer is terminated together with the currently considered function (3), i.e. either before calling another function or when exiting a function. In order to determine the unique identifier, the `this`-pointer is read just before the execution of the function is terminated.

Invoking a classic coverage tool at each of those points and mapping the information from this tool to the currently active `this` reference records the full line-based coverage and adds the information which object instance the according lines were called from.

By this, dynamically generated components are explicitly considered during the execution and tracking of coverage items.

AOP is applied to avoid the workload that a naive realization of this solution would result in. A straight-forward, manual implementation of the proposed scheme would require designers to perform significant changes in their existing project. For each function, the respective additions for step (1) and step (3) from Figure 20 would have to be added – something which should be avoided as it would introduce a lot of modifications to the code that do not add any functionality and take a considerable amount of time to write. AOP, on the other hand, can be used to add this functionality to a given code base automatically, restructuring the source code before compilation and adding the given scheme without requiring further work from the designer.

AOP's programming scheme is motivated by the following scenario frequently occurring in system design: a new functionality shall be implemented into an existing system which, however, would require an extensive re-factoring of the existing implementation (or even an entire re-development from scratch).

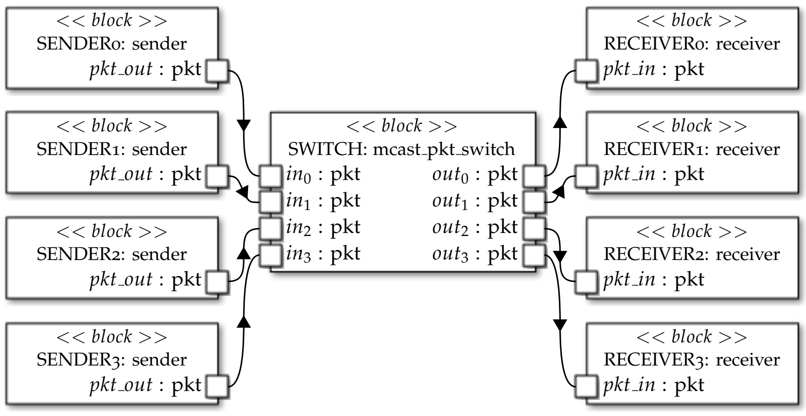


Figure 21: Simplified representation of an initialized *pkt_switch* system.

Example 7 *Consider the `pkt_switch`-system from Figure 21 which is taken from the SystemC reference implementation. Each of the classes used in the implementation contains an `entry()`-method that updates the state and output of the respective module. Let's assume that this design shall be enriched with a tracing functionality which keeps track of each call of the respective `entry()`-methods. Although all modules of that example share this method, they do not inherit it from a common base class. Hence, there is no single location to add the respective extensions to. The designer is left with the option of either*

- *enriching the entire system e.g. by an inheritance structure (which might be a lot of work and may result in other designers having to adapt their code as well) or*
- *adding the respective code into all classes (which results in redundant structures or global methods that are supposed to be called only from a certain context – both considered bad style which usually reduces maintainability).*

For those cases, AOP provides the designer with an additional layer which allows him/her to describe the new behaviour (almost) independently from the existing implementation. Following this scheme, designers avoid huge re-factorings but need to provide the implementation of the newly added behaviour *and* a description of the position it is supposed to be executed at. More precisely, AOP distinguishes between separate *component code* (which represents the existing object-oriented programming scheme that is used to describe a certain structure and basic functionality) and a so-called *aspect code* (which describes additional functionality that may be shared by several components and does not fit into the existing structure). These two kinds of code are written independently and, before compilation, are merged by the so-called *aspect weaver*. The aspect weaver takes the aspect code and inserts it into the specified positions (so-called *join points*) in the original source code.

Component code and aspect code are merged using the aspect weaver

```
#include <iostream>

aspect Tracer
{
        advice execution("%...::%(...)") : before()
        {std::cout << "before␣" << tjp->signature() <<"\n";}

        advice execution("%...::%(...)") : after()
        {std::cout << "after␣" << tjp->signature() <<"\n";}
};
```

Figure 22: Aspect code

Fig. 22 shows a simple aspect that illustrates how aspect code is to be used and written.

- An *aspect* is named (in this case "Tracer") and extends the concept of classes. While an aspect may thus contain field variables and functions, it may also contain
- *advices* that define where and how the existing project should be altered during the weaving process. An advice is declared using several constructs:
  - The `advice` keyword.
  - The *pointcut* that defines the *join points* where the code is supposed to be altered. In this case, this is the predefined function `execution` and a pattern to be matched ("%...::%(...)", which matches all function calls).
  - The `before` or `after` keyword, defining *how* the code is to be altered – in this case, the aspect code is inserted either before or after the matching pointcuts.
  - The aspect code that is inserted. Notice that join points introduce a `tjp` object (of type `JoinPoint`) that allows accessing information about the join point itself. In this case, this is used to print the method signature before and after all function calls.

Fig. 23 illustrates a minimal SystemC program that merely writes a single line to the console. When woven with the aspect code presented in Fig. 22, however, its output is enriched with information about the function calls, as seen in Fig. 24. Notice that the original SystemC program remains unchanged but the resulting program's behaviour is altered.

Basically, the AOP workflow can be understood as a series of code refacturing instructions that can be applied to any given code base. The aspect code contains the instructions concerning how the given code base should be refactored and running the weaver gives the designer the result of applying the given rules.

```
#include <stdlib.h>
#include <systemc.h>

SC_MODULE (hello_world) {
        SC_CTOR (hello_world) {  }
        void say_hello() {
                cout << "Hello_World.\n";
        }
};

int sc_main(int argc, char* argv[]) {
        hello_world hello("hi");
        hello.say_hello();
        return(0);
}
```

Figure 23: Minimal SystemC program

```
        SystemC 2.3.1-Accellera --- Sep 18 2016 13:54:52
        Copyright (c) 1996-2014 by all Contributors,
        ALL RIGHTS RESERVED
before int sc_main(int,char **)
before void hello_world::say_hello()
Hello World.
after void hello_world::say_hello()
after int sc_main(int,char **)
```

Figure 24: Aspect code output

Example 8 *Using AOP, the desired tracing functionality from Example 7 can be realized by leaving the existing code as it is. Instead, the tracing functionality is separately realized in an aspect. This aspect code also specifies that all classes inheriting the `sc_module`-class and containing an `entry()`-method (i.e. sender, receiver, and switch) should execute the newly added tracing implementation whenever `entry()` is called. The resulting code consists of the original SystemC model as well as a description of additional functionality that describes where in the design (i.e. "after calling the `entry`-method") which functionality (i.e. "trace the execution") should be added.*

The ability to transparently "weave in" aspect code just before compilation allows for the insertion of additional functionality to large source bases without further interaction. Other designers of a given system do not even need to know about new features being added: functionality that is needed at some other point in the workflow can remain hidden from them. This results in less complexity as designers are only presented with implementations related to their respective tasks – a clear separation of concerns.

In order to apply AOP in SystemC, or C++ in general, *AspectC++* [99] provides an implementation of the respective programming scheme. AspectC++ performs thereby two steps (which are illustrated in Figure 25):

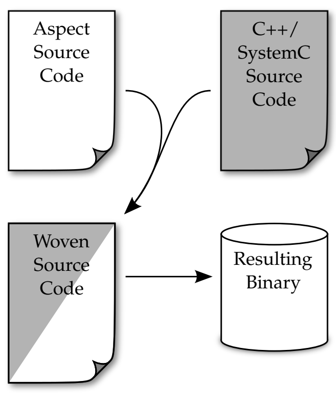


Figure 25: The workflow for AOP.

- First, *weaving* is applied, i.e. the original source code as well as the respective aspect source code (which contains its own keywords to specify the aspects but largely sticks to the C++'s syntax) is taken and a corresponding woven source is created. This code includes both, the original functionality and the parts introduced in the aspects.
- Afterwards, the resulting source code (which is not necessarily meant to be read or edited) is *compiled*; directly leading to a binary executable which allows to perform the new functionality.

Existing approaches that utilize AOP for ESL design analysis

Due to the promising traits of AOP, some approaches have already utilized this approach for various aims.

- Although not combined in a single tool, [19] outlines the application of AOP for metrics collection, functional verification, communication and the more specific use cases of a cache replacement policy and the separation of control and data streams for given examples.
- As SystemC designs are usually focussing on a system's structure and behaviour (especially in their early stages of development), concerns such as power estimation are usually ignored at first and hard to add later on. AOP provides an easy way to add the needed code for power estimation throughout a design, adding the code that implements the corresponding power model once the structural and functional description is done without the need for any heavy refactoring [55].
- Usually, the resulting source code that is generated during the weaving process is quite complex. This is not only a problem for its readability and debugging but also for the synthesization of hardware from a given SystemC design if the original design was

composed from SystemC's synthesizeable subset and meant to be translated into hardware. ASystemC [30] is an AOP implementation that, unlike AspectC++, focuses on SystemC and its ties to hardware design.

Although the focus on hardware development and synthesizeability seems fitting, ASystemC relies on a custom parser for the interpretation of the original SystemC code, sacrificing portability in order to implement its own engine.

In order to comply with the notion of non-intrusiveness, ASystemC was therefore not considered to be used for the sake of applicability.

- CHIMP (*CHIMP Handles Instrumentation for Monitoring of Properties*) [111] is a tool that monitors temporal SystemC properties. The basic approach in this case is that certain properties are hard to describe with standard C++ assertions (especially properties relying on temporal relations) and custom property checkers are hard to embed into existing code bases. AOP, in this case, provides a framework to embed this sophisticated monitor into existing SystemC projects.

  CHIMP adds another layer of abstraction above the usual writing of AOP advice code, taking instead properties and locations in a custom language and generating the according aspect code. Apart from providing a more abstract way to exploit AOP's features, CHIMP also adds functionality beyond the mere translation of its own property expressions into aspects: in order to overcome AspectC++'s limitation of being unable to match arbitrary syntax, CHIMP allows matching any piece of code via regular expressions as well. While this does not allow matching syntax on a semantic basis (which might be a problem for statements that should be matched but contain some unexpected character), it is a solution that should work fine for most cases.

Hence, AOP represents a promising approach to integrate static metrics into existing system designs according to the scheme illustrated in Figure 20.

Shortcomings of the AspectC++ implementation

However, the API that is provided by AspectC++, while extensive, does not provide the ability to describe arbitrary points in the source code. While CHIMP attempts to remedy this by providing additional expressions to describe certain code constructs, using features such as information about the program flow to analyse a given design is a hard, if not downright impossible task. While regular expressions could be used to e.g. match `if`-statements, C++'s notoriety of relying on preprocessor directives that modify the code just before compilation leaves a string-matching based approach with the issue of also matching the according macros. Generally, as regular expressions have no notion about the underlying program structure, pointcut statements such as "in a class that inherits from `sc_module`, match each variable declaration that has a type that inherits from a particular class" become impossible to achieve.

However, as AOP is – in this context – supposed to merely provide the means to apply another analysis tool, such limitations are of no concern as long as the chosen tool performs the given task as required using the means AspectC++ offers. This means that the detailed, static analysis (e.g. which lines were executed in particular, which branches were taken etc.) can be left to the tool that was chosen to be inserted. The *this* reference that is crucial for the division between the static information (i.e. which part of the code is being executed) and the dynamic information (i.e. which instance provides the context for a given execution) can *only change with each function call*. As AOP *does* provide pointcut functions to match function calls before and after their call and return statements, using AOP to apply a given coverage tool to a given design that can then be enriched with the information of its current context is a viable approach that bypasses both,

- AOP's shortcomings concerning the descriptions of pointcuts that are neither namespace/class/structure declarations nor function calls and
- issues of analysis methods concerning the tracing of dynamic features of a given program.

Integrating AOP and *gcov* to extract behaviour information

It is thus a valid approach to integrate AOP and the *gcov* coverage tool to accomplish the task of retrieving coverage items on a per-instance-basis, allowing a more detailed extraction of a system's modules' behaviour.

Following the AOP-based programming scheme allows for a non-intrusive integration, i.e. the proposed analysis functionality can easily be integrated into existing SystemC projects. In fact, since SystemC is a C++ library, AspectC++ itself can be used off-the-shelf, i.e. unchanged. Only the existing project building process needs to be altered as the weaver needs to be built into the existing compilation workflow. However, this is a one-time setup and works on all major platforms [100]. Afterwards, a set of aspects can be used to add the new tracing functionality to any given SystemC design.

This results in a system in which calls of functions f are conducted as shown in Figure 26: first, information recorded thus far (by the respective line coverage analysis) is flushed prior to each function call. Thus, all results of the analysis are saved (in the current implementation, they are written to the hard drive) and associated with the current scope (i.e. the instance which calls the function). Then, function f is executed. Since the line coverage analysis is still running, new information is gathered. Before the function terminates, another flush is executed, i.e. the

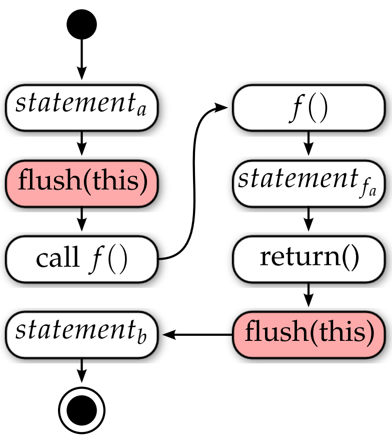


Figure 26: Integrated flow.

newly collected information is saved again (now associated to the instance which hosts the function f, again by storing the `this` reference).

The result is a list of coverage analysis files – all associated to the instances on which the respective code was executed. The chosen implementation stores the information only in relation to modules, i.e. any classes that inherit from `sc_module` in any way for several reasons:

Limiting the AOP insertion to objects inheriting `sc_module`

- The implementation can safely assume that the object is a module and its name field can be accessed.

  As SystemC requires the names to be unique (and alters them accordingly if they are not), this field can be used as the unique identifier to link the coverage data to. This essentially also means that no other unique feature needs to be found – which would not be a trivial problem as well, as e.g. memory addresses may be reassigned and C++ objects do not have any means to uniquely identify them in place. While AspectC++ could also be used to rewrite any object to contain a unique identifier, this may result in issues when the designer expects his objects to have their original memory layout.

- Object creation does not need to be tracked during simulation.

  As SystemC prohibits the creation of new objects that inherit from `sc_object` once the simulation has begun, assuming that the tracked objects and chosen IDs remain unique and valid is reasonable.

  While AOP does provide pointcuts to match the construction and destruction of new instances, these may be frequent if each and every class' and struct's creation was tracked, requiring the tracing method to constantly change the given context, which may be a performance problem depending on the implementation of the analysis method.

- There is a direct connection to the simulated system for each traced feature.

  Objects that are created but not explicitly referenced may have been set up somewhere deep within the program to provide a functionality that is not directly relevant to the system itself. Hiding such instances lets the designer focus on the objects that are parts of the simulated system.

Notice that this does *not* mean that code that is part of non-SystemC-objects is not traced. The suggested application of *gcov* e.g. results in code that is part of other classes being traced appropriately but attributed to the module that called the according function in the first place.

Example 9 *Assume that two modules a and b are created, both instances of the CLS class. Both modules share a static reference to a single third object generic_object that is* not *inheriting any SystemC class and that provides a public do() method. During simulation, a calls do() while b does not. The resulting traces do* not *track the execution as belonging to generic_object. Instead, the call is attributed to a as the SystemC object that was responsible for calling the third instance's method.*

Overall, this leads to several advantages which make the proposed solution very applicable for feature localization in SystemC designs:

Advantages of applying AOP in SystemC designs

- The original source code does not need to be modified in any way. Existing SystemC projects can easily be analyzed by only modifying the compilation workflow to include the aspect weaving step.
- The existing compilation setup can be used for the resulting, woven code, as long as the given compiler offers the coverage mechanisms that the designer wants to apply.
- The SystemC library does not need to be modified in any way. Unlike approaches that analyze a running SystemC design by modifying e.g. the simulation kernel, the given approach works purely on the user-generated code base. This means that the approach can be applied to future versions of SystemC without any further changes and that it can be combined with setups that already rely on a modified SystemC library.
- The proposed approach is flexible, i.e. various tools and/or schemes for line coverage analysis can be chosen. Since tracking the information obtained by these tools/schemes can be realized using aspects, e.g. the time-consuming gcov operations can easily be omitted if performance is crucial.

- The proposed approach is platform-independent. As long as a corresponding line coverage analysis tool is available (which is the case for all major platforms), the proposed solution can be implemented. This is possible, because AOP-implementations such as AspectC++ are available for all major platforms as well.

Using AOP does require the designer to make certain trade-offs though.

Issues of applying AOP in SystemC designs

- The compilation flow must be compatible to AspectC++.

  While AspectC++ is available for all major compilers, it is still a third party tool that does neither guarantee that it works in all cases and will continue to do so. This issue can be seen with the current switch toward C++11 and C++14 [33], updated versions of the C++ standard that provide extensions to the C++ language and its standard library. Both of these are unsupported as of now, with C++11 being on the roadmap and C++14 not yet being considered. The applicability of this approach is of course ultimately depending on the AOP implementation being available for the chosen platform, which includes the supported code constructs. Therefore, even with AspectC++ being open source, it is critical that this framework is kept updated and in line with the C++ language for it to remain applicable.

- Depending on the analysis method and the extraction frequency defined by the aspects, performance may take a significant hit.

  The gcov approach e.g. requires storing the coverage data on disk for each inserted extraction point, consecutively mapping the files to the according ID (which is, in this case, done by renaming the file). Even when running this method on a RAMDisk (i.e. a part of the computer's RAM instead of a hard drive), the I/O becomes a bottleneck that slows down the simulation speed. The simulation of the `pkt_switch` system shown in Figure 21 for the basic system and the applied AOP/gcov combination on both, the hard drive and a RAMDisk can be seen in Table 1. While the execution time grows significantly by several orders of magnitude, storing the data is the major culprit in this case as even simulating these 20 cycles results in the storage of more than $30,000$ files on the hard drive.

  Using alternative approaches that do not require the intermediate files to be written to disk could be an option to improve performance.

Generally speaking, tracing the program output results in a performance penalty depending on the chosen extraction method. However,

Table 1: `pkt_switch` execution time for 20 simulated cycles

	no coverage	AOP/gcov on HD	AOP/gcov in RAM
time	0.103s	135.767s	62.404s
factor	1	≈ 1318	≈ 606

due to the chosen approach, there are hardly any other drawbacks, apart from the required alterations to the system's build workflow in order to add the weaving process before the program compilation.

The resulting workflow offers an applicable, non-intrusive way of injecting analysis methods into a given design, allowing the behaviour of the program to be altered without any further requirements concerning.

The given approaches, the extraction itself and the injection using AOP, usually need to be adapted in order to work together. The aspect code needs to know which code fragments need to be inserted where and the analysis code needs to be adapted to work in the target source code, woven into the project. Still, the AOP approach allows to transfer these methods into a given C++ design easily – for the given insertion of gcov, the coverage tool itself did not need to be altered at all and the aspect code handled the required interactions with gcov in less than 200 lines of aspect code and scripts.

The proposed method therefore represents a sound approach for analysing SystemC designs. It is non-intrusive and yet allows for arbitrary analysis aproaches to be inserted into any given SystemC design.

CONCLUSION

This chapter presented methods to analyse and extract the behaviour of SystemC designs.

It first outlined existing software approaches to analyse a system's behaviour. Afterwards, the simple case of applying the methods introduced in Chapter 3 was suggested, which works but shifts the burden of applying it to a given system to the designer, which may be a tedious and time-consuming process.

The issue was then divided into two core parts – the extraction of the data itself and the invocation of said extraction method. The former could remain an application of the methods provided in Chapter 3 or the usage of other methods such as the usage of coverage tools. The latter was realized using Aspect Oriented Programming which allows to specify the points at which the extraction methods should be invoked without requiring the designer to alter the source code manually.

This separation of concerns allows the final program to extract detailed information from a running SystemC program while remaining non-invasive (i.e. not altering the original sources before the compilation workflow starts) and works on all major compiler platforms.

5 FILLING THE GAPS WITH MACHINE LEARNING

Maybe the only significant difference between a really smart simulation and a human being was the noise they made when you punched them.

Terry Pratchett [87]

The approaches introduced in chapters 3 and 4 retrieve detailed data about a design's structure and behaviour. Still, some information remains that cannot be extracted using these methods. Using the methods from Chapter 3 the *state* of the model is retrieved, with the objects that have been created and their values being the aspect that is retrieved.

Extracting the internal state may not be enough

However, classic hardware models such as a Mealy [68] or Moore [74] automata clearly indicate that a design's state is but one aspect of a circuit's output. The other part is the logic itself, which (depending on the model) together with the input determines the output of a system. This part – the logic of the module – is not extracted by the structural extraction features. Figure 27 illustrates the issue: the state which is extracted does not solely define a system's output, as the internal logic is not handled by the given approaches.

That internal logic is a direct representation of the program's source code. For a detailed model of the logic, designers may therefore simply look at the source code, especially when trying to understand a certain module in detail. However, design understanding, albeit important, is only one of many applications of detailed system models. Other use cases may require complete models of the system, so analysing the logic may be needed for some cases.

The source code is the element in a SystemC program that constitutes the logic that defines both, the system's output and its consecutive

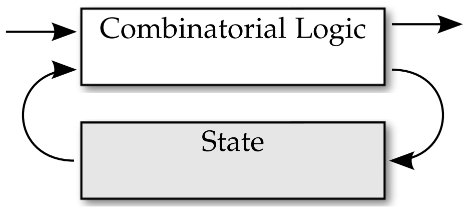


Figure 27: The extraction methods so far only cover the system's state (bottom, grey) but not its modules' internal logic (top, white).

states. Especially for designs that utilize SystemC, the automaton model from Figure 27 becomes obvious. As each module contains one or more processes that are invoked by the simulation kernel and run until they are finished (and the module has reached its next state and produced the desired output), each of these processes (which are described using arbitrary C++ source code) can be regarded as the logic part of the automaton.

This is again raising the issue outlined earlier. While C++ source code is generally machine-readable (as it needs to be interpreted by a compiler to be translated into machine-code), this is neither an easy task nor is it possible to write a single cross-platform code parser that is compatible to all dialects and frameworks that are built for C++.

Issues for retrieving module connections in SystemC designs

Hence, while analysing the internal structure is easy for circuits provided at the RTL or the gate level, new obstacles are introduced if a circuit or system is available at the ESL for several reasons:

- SystemC allows for a significant number of different description means (the full expressive power of C++ including its dialects) which need to be supported by the respective analyzer. This, of course, can be addressed by altering the code so that it is composed of supported constructs only. However, this might become a time-consuming and, therefore, expensive task.
- The availability of source code usually stops at pre-compiled libraries, user inputs, file or network access, and all other possibilities of the program interfacing with any other element other than source code. For all these elements, it is unknown how to obtain the respective internal structures.
- Global variables could be changed by any function or library call at any time. Consequently, if the result of a structural analysis includes a global variable, the number of further components possibly also being part of the respective dependencies increases significantly. This results in the accumulation of dependencies that might not be part of the logic behind the source code.
- In order to reduce these overestimations, an analyzer needs to decide whether it should act conservatively or progressively, i.e. either taking all possible value changes into account (and maybe end up discovering much more dependencies than there are) or leaving out those that it deems reasonable (and maybe end up not having the one connection that is important for a certain case).

As a result, if source code is not or only partially available (which might already be the case when using standard library calls), analysing the internal logic at the ESL is restricted. Even if it is, retrieving the information is not trivial though. Generally, there are two approaches to read the C++ source code. Either, a custom front end can be written to specifically target e.g. SystemC (which still has to encompass the full C++ language in order to be complete) or an existing compiler can be used to leave the full pre-processing to an existing tool and focus on retrieving the information from it. However, *both of these options violate the non-invasiveness principle* as they are not portable and thus require the source code to adhere to the constraints imposed by the tools that are being used. The former does it by not being able to support all compiler-specific additions to the language such as extended standard libraries, the latter by not being applicable to certain compilers such as MSVC++.

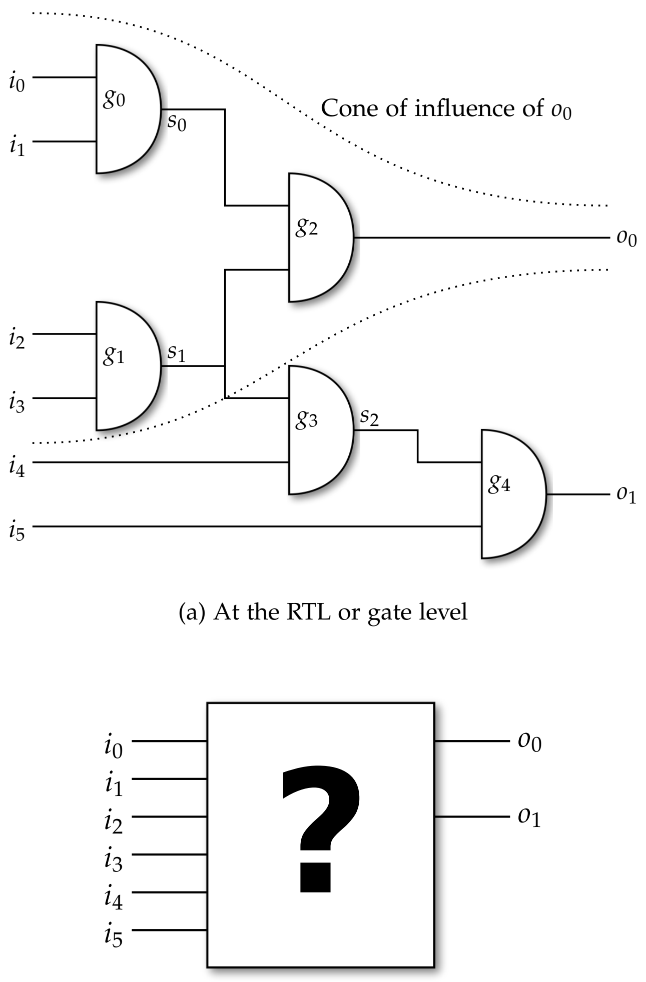
Figure 28b illustrates the extraction issues: while the respective connections between SystemC modules can be extracted via an API (solid lines), connections within a module (which are described using C++ source code) cannot be derived if the implementation of these modules is not available, if the corresponding description means are not supported, or if their respective values might be affected through side effects like global variables. Consequently, e.g. the dependencies of signal o_0 cannot exactly be determined.

Knowing the dependencies within hardware systems has always been a key issue in several design tasks though. The general idea is to take only those parts of the circuit into consideration that are relevant to the respective design task. Applications can e.g. be found in the following domains:

- Design understanding [40]: If a designer wants to understand the purpose of a signal, only the components triggering this signal need to be inspected.
- Debugging [1]: If an erroneous behavior occurs on a signal, the reason for this error must be within the components triggering this signal.
- Verification [3, 17, 6]: If the correctness of a signal shall be verified, the verification engine only has to consider the components triggering this signal.

These applications' general idea is to ease the execution by taking only those parts of the circuit into consideration that are relevant to the respective design task.

At abstractions like RTL or gate level, components that influence a certain signal can easily be obtained by a backward traversal starting from the currently considered signal. At the ESL, however, this backward search cannot be easily achieved as the internal structures are not available as a graph structure. Instead, as the internal module logic may be described using arbitrary C++ constructs, analysing these structures is far from trivial, especially considering the non-invasiveness principle



(a) At the RTL or gate level

(b) At the ESL

Figure 28: Module analysis at the RTL and the ESL

(which e.g. requires to not depend on a particular compiler – basically rendering a reliable source code analysis unattainable).

Example 10 *Consider the gate level circuit shown in Figure 28a and assume the logic for the output signal* o_0 *shall be determined. Since* o_0 *depends on gate* g_2 *and its two inputs* s_0 *and* s_1*, these elements are added to the formula. Since, in turn,* s_0 *and* s_1 *depend on gates* g_0 *and* g_1 *with their inputs* i_0*,* i_1*,* i_2*, and* i_3*, also these components are added. In contrast,* o_0 *does neither depend on the gates* g_3 *and* g_4 *nor on their input signals. Hence, while the overall circuit is composed of five gates in total, only three of them need to be considered in order to observe* o_0*.*

The ESL model shown in Figure 28b, however, cannot be analysed in this way. As the module interiors are not part of the structural information that can be retrieved using the SystemC API, retrieving the information that the signal o_0 *depends on the input* i_0 *but not* i_5 *is not easily possible.*

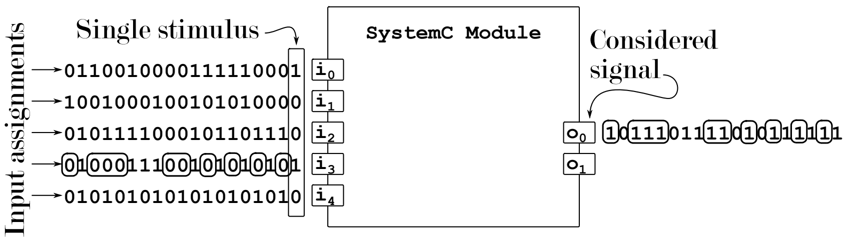


Figure 29: Deduction of links between stimuli and signals: While the streams of stimuli might seem random at first, i_3 correlates with the considered output signal, indicating a dependency between input and output.

Motivated by this consideration, an alternative to the "classical" approach of locating the dependencies is needed which supports the new requirements at the ESL. Hence, instead of a structural analysis, a behavioural scheme is proposed. To this end, a set of stimuli is applied to the system with the intent of triggering various behaviours. Afterwards, the relations between the respective signal assignments are observed in order to deduce the desired information. The following example illustrates the idea.

Example 11 *Consider the abstract SystemC module as illustrated in Figure 29 and assume that the formula for the output signal o_0 shall be determined by means of a behavioral analysis. To this end, 19 input/output assignments are provided. At a first glance, these assignments are merely detached streams of Boolean values. However, at a second glance, certain characteristic relations can be observed. For example, each time input i_3 is set to 0, the considered signal o_0 outputs 1. This may lead to the conclusion that the value of o_0 depends on the value of i_3. In a similar fashion, further conclusions can be drawn.*

In summary, this means that there is a variety of crucial applications that require the structure of the modules' internal logic but no way to retrieve it that works in the desired manner. In order to fill this information gap, this chapter proposes an approach based on established algorithms in Section 5.1 and details ESL-specific issues and additions in Section 5.2.

REVERSE-ENGINEERING MODULE LOGIC

The proposed approach determines the internal logic preceding a given output from a thorough analysis of certain assignments to the signals of the considered system from which characteristics of their relations are automatically deduced. Consequently, the quality of the approach significantly relies on the set of stimuli applied to the system. For the machine learning algorithm to perform well, a number of "good" stimuli to be considered need to be generated. To this end, different strategies may solely or in parallel be applied (see e.g. [126]). In particular, they should incorporate the following two characteristics:

1. *Diversity*

   An important indicator for a well-chosen set of stimuli is the diversity of the assignments to the considered signal that is supposed to be classified. If these assignments rarely change, the machine learning algorithm will usually have trouble drawing helpful conclusions. For example, consider a simple system computing $c = a \wedge b$. If only stimuli $\{a = 0, b = 0\}, \{a = 0, b = 1\}, and \{a = 1, b = 0\}$ are applied, a possible conclusion could be that signal c is *always* set to 0. In order to avoid that, stimuli leading to $c = 1$ should also be considered. Following the same reasoning, also the assignments to the respective input signals should be diverse. Otherwise, it is harder to determine which input signals actually triggered a change in the signal.

   In general, the machine learning algorithm attempts to "tidy up" a given set of cluttered signal assignments and, by doing this, extracts the desired information. Therefore, the provided set of data should be as "untidy" or as diverse as possible. For the considered signal, this is not easy to achieve since its values are determined by the module's inner, unknown structure. However, many approaches for stimuli generation have been proposed in the past, which can be exploited to satisfy these requirements including approaches for simple random simulation or directed simulation [126], or more elaborated methods like e.g. constraint-based random simulation [123, 125].

2. *Quantity*

   Apart from the assignments to be applied to the signals, the amount of different assignments is also an important factor to the quality of the result. In general, the more stimuli are generated, the better results can be achieved. When using machine learning algorithms to classify a given signal, the algorithms tend to suffer from *overfitting* when they are exposed to too much information (see e.g. [44]). The general issue is that instead of learning a system's generic structure, results that may be e.g. noise are learned "by heart" and, in an attempt to use these to anticipate the behaviour, worsen the resulting predictions. When deterministic behaviour is considered (as it is the case for systems specified in ESL), the applied machine learning approaches are usually hardly affected by this as long as the algorithm can access all required variables.

Example 12 *Consider again the abstract SystemC module as illustrated in Figure 29. The depicted assignments already satisfy the characteristics discussed* *above very well. The assignments to both, the considered signal as well as the inputs, are rather diverse. Furthermore, an adequate number of stimuli is available. As shown in the next section, this set leads to very precise conclusions on a module's internal structures.*

Using the assignments generated in the first step, relations between them from which the desired information can be deduced are determined next. To this end, a machine learning approach is utilized. Basically, the idea of machine learning is to locate those patterns in the given set that provide the simplest explanation possible of the given phenomenon. This follows Occam's razor which states that, if there are several theories that explain a given phenomenon, the one making the least assumptions probably is the right one [44]. This idea of mere correlation implying causation usually should be applied carefully. However, especially in discrete and manageable environments like system designs, the assumption that correlating signals are somehow connected certainly is legitimate.

The C4.5 algorithm

While there are many algorithms that can be used to classify data sets (see e.g. [49]), in this case the C4.5 algorithm introduced in [89] is applied for the purpose of module analysis. This approach is widely used and has been proven efficient in many applications as computer vision [34], language processing [61], medical diagnosis [129], financial analysis [65], general game playing [97], robotics [13], and others.

C4.5 takes a given data set and recursively splits this set into more "tidy" sub-sets. To this end, all possible splits are previously evaluated by means of the entropy function:

$$-(p_0 \log_2(p_0) + p_1 \log_2(p_1)) \qquad (1)$$

In this function, p_0 (p_1) denotes the probability of the considered signal being set to 0 (1) according to the data set. If no progress can be obtained from further splitting anymore, the algorithm terminates. The resulting tree gives an assumption of the internal structures that lead to a certain signal setting. More precisely, the following steps are performed:

1. Determine the entropy of the currently considered set.

2. Determine an input signal on which the currently considered data set shall be split. To this end, apply the entropy function, i.e. choose the input signal which would lead to two sub-sets whose entropy (i.e. "tidiness") is better than in the currently considered set.

3. In case no such input signal could be determined, add a leaf node to the decision tree and terminate. This happens if all available stimuli (i.e. the given data set)

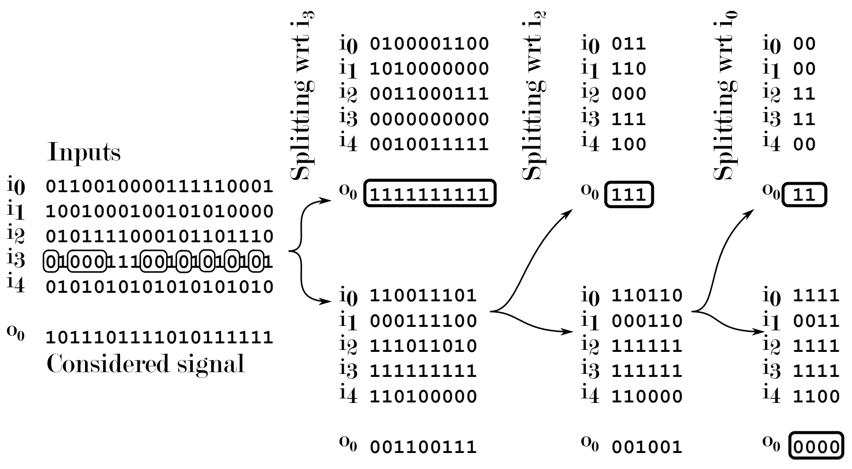


Figure 30: Applying machine learning

- always lead to a constant assignment for the considered signal or
- cannot further be refined through splitting.

4. Split the currently considered data set with respect to the chosen signal. Afterwards create a decision node whose successors pinpoint to the newly created sub-sets.
5. Recursively start over at Step 1 using the newly created sub-sets.

The resulting decision tree represents an approximation of the internal logic of a module that influences the considered signal. If more than one output signal needs to be extracted, the approach can simply be applied iteratively to all signals.

Example 13 *The described scheme is applied to the example from Figure 29. First, the entropy of the given set is calculated. To this end, the entropy function* $-(p_0 \log_2(p_0) + p_1 \log_2(p_1))$ *is applied. That is, using all stimuli from Figure 29, an entropy of* $-\frac{4}{19}log_2(\frac{4}{19}) - \frac{15}{19}log_2(\frac{15}{19}) \approx 0,742$ *results. Based on this, it is determined which splitting on what signal would lead to sub-sets with the best entropy. For example, if the data set from Figure 29 would be split with respect to input* i_0*, a sub-set of 10 stimuli and entropy of 0 (considered signal is always* 0*) and a sub-set of 9 stimuli and entropy of* 0.991 *would result leading to an average entropy of* $\frac{10}{19} \cdot 0 + \frac{9}{19} \cdot 0.991 = 0.469$*. Overall, splitting with respect to the available inputs would lead to the following average entropies:*

- i_0: $\frac{10}{19} \cdot 0 + \frac{9}{19} \cdot 0.991 = 0.469$
- i_1: $\frac{12}{19} \cdot 0.619 + \frac{6}{19} \cdot 0.918 = 0.681$
- i_2: $\frac{9}{19} \cdot 0.619 + \frac{11}{19} \cdot 0.946 = 0.547$
- i_3: $\frac{10}{19} \cdot 0 + \frac{9}{19} \cdot 0.991 = 0.469$

- i_4: $\frac{10}{19} \cdot 0.722 + \frac{9}{19} \cdot 0.764 = 0.742$

Hence, splitting with respect to i_0 and i_3 would lead to the most tidy sub-sets. Assume the algorithm decides for i_3. Then, a decision node and two new sub-sets are added as shown in Figure 30. Here, it can be seen that the first sub-set is only composed of stimuli setting the considered signal to 1, i.e. the entropy is 0 and this sub-set is not further split. In contrast, the second sub-set can further be refined. This is done in the remaining iterations eventually leading to the complete decision tree as shown in Figure 30.

From this decision tree, it can now be deduced that the considered signal very likely depends on the inputs i_0, i_2, and i_3, i.e. the input signals that had the most possible effect to it. In contrast, inputs i_1 and i_4 are not part of this tree and, hence, probably do not influence the considered signal.

All in all, the machine-learning approach is non-invasive, does not rely on the availability of the code, and performs fast, but does not guarantee exactness of the results. The method could be applied for design understanding, speeding up tests and verification tasks and visualizing an approximation of a module's interior structure.

ESL-SPECIFIC ADDITIONS

While the machine learning approach works nicely "off the shelf", i.e. without being altered in any way, it faces problems when certain SystemC features are utilized or the design is done in a way that prohibits further analysis. This chapter addresses the problems present and either suggests viable approaches to solve the issues or outlines why they may not be solvable when retaining the non-invasiveness principle.

Just as with the previously described methods, no a priori information should be required for the approach to be applied. While a lot of the required information could be supplied by the designer in e.g. a formalized language of some kind, this would mean a major effort on top of designing the system itself, which is undesirable at best and inconceivable at worst. This point again reflects the non-intrusiveness, which is a desired trait of any approach handling SystemC designs and a focus of this work in particular: as the analysis should work on all given designs without any further alteration being required, no knowledge can be taken for granted that cannot be extracted from the standard interfaces that are available anywhere.

The application of the *C4.5* algorithm generally works well as illustrated in Section 5.1. Despite this general applicability, depending on the language features being used the algorithm may face certain problems, resulting in less precise results or the omission of certain traits of a system. While one advantage of machine learning algorithms is that they "make do" with whatever information they get and simply end up with less precise (but still usable) results when they analyse incomplete sets of data, increasing the amount of information that is salvageable from a design will still result in the algorithm being able to improve its results.

Hence, in order to exploit certain traits of SystemC and to make this generic approach work better in conjunction with arbitrary designs, issues specific to the application of the algorithm to SystemC and approaches to handle them are described in this Section. The goal is to analyse which of these features can be handled properly by an automatic, non-invasive approach and which need to be communicated to the designer as something that may lead to poor results when a machine learning approach is used to analyse a given system.

Non-invasiveness of the machine learning approach

While especially the latter may be regarded as a breach of the non-invasiveness, it is not – the general approach of machine learning remains applicable even when guidelines are not followed, albeit at the cost of decreased precision for the resulting models. Therefore, the idea of doing both, providing technical solutions wherever they work while still giving designers hints concerning design principles that they may want to follow for the given approach seems a valid trade-off, resulting in an approach that remains applicable and non-invasive for existing systems but may offer better results when certain criteria are matched.

A major distinction between the simple application of the *C4.5* algorithm onto a given set of data and adapting it to SystemC's features is that, in order for it to be integrated into a given design, it needs to be run in conjunction with a simulation and given access to the SystemC kernel. Strictly speaking, simply applying a learning approach can even be done using simulation traces alone, allowing the algorithm to be tested with recorded values of a given simulation. In contrast, in order to retrieve all required data from the simulation and the kernel at run-time, a module was implemented that can be added to a simulation. This analysis module is performing the required calls to the SystemC API and any other methods that are required to retrieve the needed information.

While this may have a certain impact on the simulation performance (depending on the size of the system and the operations that are performed), this also means that the approach can be used to interact with the design, allowing additional applications to be considered.

SystemC's modularity

SystemC's modularity is a concept that differs from the approach of building a single decision tree for a given set of data.

SystemC designs usually contain several modules that interact via ports and signals. While it would be possible to build a single, large decision tree for the signal that is supposed to be set (which would have to take all connections to the inputs into account), building smaller models for the modules has several advantages.

- A better division of labour, allowing the machine learning approach to work on small, independent problems. While the *C4.5* is quite fast, Section 5.1 shows that an increased amount of variables results in decreased precision. Generally, a smaller amount of input stimuli enables the algorithm to provide better results. Using the modularity to retrieve much smaller sets of input stimuli is thus a viable way to increase the precision of the output result.

- Additionally, it allows re-using the data for modules that are cloned throughout the circuit. If the circuit was analysed as a whole, interconnected modules of the same type would increase the complexity of the data. If instead the modules are recognized as the same type, the algorithm can safely assume that their internal logic is equal and thus can build a single, simple model of the given module instead of building a larger, complex one.

SystemC allows for the retrieval of the modules via its own API, giving the proposed implementation access to the list of modules in order to analyse them separately. Additionally, with the SystemC module base class already utilizing virtual methods, Run-Time Type Inspection (RTTI) can be used to retrieve a unique type specifier for each module as previously discussed in Chapter 3. This means that all module instances being used in a design can be mapped to their types automatically, allowing the given implementation to calculate separate decision trees for each type and handling duplicates accordingly.

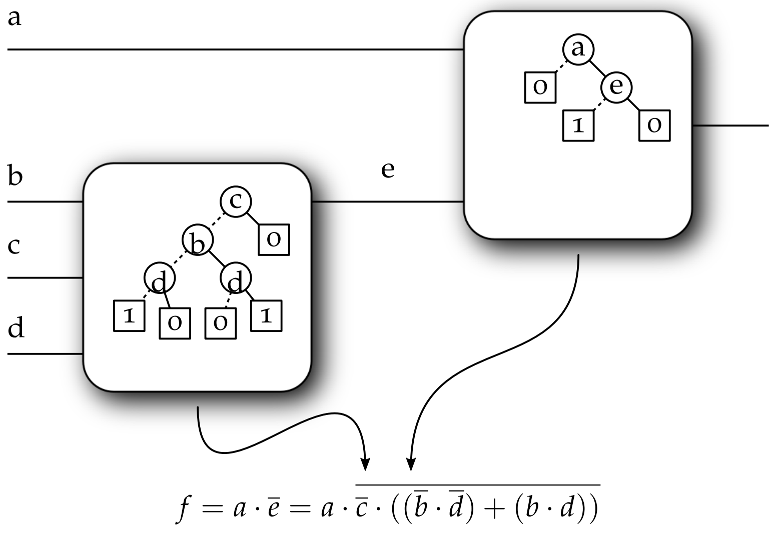
The approach therefore is to use this pre-defined modularity that is described in the design to first divide the learning problem into smaller parts, generate models for each individual class of modules and later merge these parts to be able to generate the inputs that are needed for a particular result. Figure 31 illustrates this method, building decision trees for each module which can then serve as a foundation to calculate a general formula for any given output signal.

In order to still provide a single model of an arbitrary system for a given output signal, the proposed implementation takes the trees for the desired result, translates them into boolean statements and merges these. This gives any application both, simple models of modules or a large one of the whole design while still utilizing the modularity to simplify the calculation of the latter.

Time of extraction

A SystemC simulation does not have to follow the traditional clock scheme.

While SystemC allows for clocks to be used, they are not required for a system to work. Instead, the designer may e.g. simply tell the



$$f = a \cdot \bar{e} = a \cdot \overline{\bar{c} \cdot ((\bar{b} \cdot \bar{d}) + (b \cdot d))}$$

Figure 31: Several trees may be merged over module borders to retrieve the formula for a single output.

simulation kernel to wake up a given thread after an arbitrary amount of simulation time and generate new input for the design.

This makes it hard for any algorithm to determine any point in time when a system's status should be observed and logged for further analysis.

When a clock is attached to the stimuli generation module, it uses the clock's period information to invoke itself a δ-cycle (which denotes the minimum amount of time that may pass in a SystemC simulation) before the next clock edge is set in order to wait for the system to have reached a stable state with all signals and variables being set coherently.

If the system does not rely on a clock to update its modules, the stimuli generator needs to be invoked manually (i.e. via a method call) each time the system has reached its next state.

Example 14 *A SystemC module `asnycAdder` that realizes an adder, accepting two integers as inputs and providing a single integer output and another boolean overflow output (that fires if the addition resulted in an integer overflow) is built around a single `add` method that is invoked as soon as any of the inputs is altered. As the module is built asynchronously, it does not require a clock signal. Instead, its output is computed as soon as the required inputs change.*

However, this computation neither means that the output change immediately (as delays may be introduced in the method to simulate the module taking time to compute the results) or at the same time (as – even if the signals are set immediately after each other and without arbitrary delays being inserted – the *simulation kernel picks an arbitrary one to be set first). Thus, the machine learning module cannot deduce the point in time at which the module is done with its computations and the results indeed reflect the proper output.*

If a synchronized module uses a clock, this clock signal is used to bypass this problem: due to SystemC simulative nature, the outputs can be read immediately before *the next clock tick results in a recalculation of the module's values, assuming that the clock will only be toggled as soon as the system has reached a stable state.*

Time delays

These may result in a given signal to affect the output an arbitrary amount of cycles after it has been set.

A module that is connected to certain signals may have any of its methods set to be *sensitive* to any of these signals, meaning that the given method will be executed if the given signal alters its value. This is usually used to calculate a module's outputs when a certain input changes. As an example, several modules may be connected in a straight line, with all of them being connected to a clock. The method that recalculates a module's output can be set to be sensitive to the signals a_n and b_n or the clock, triggering its behaviour when the respective signal is set to a new value. Assume that all signals, a_n, b_n and the clock, are set at time $t = 0$. If the modules are set to be recalculated upon a change of signals a_n or b_n, a change in the respective signal will trigger a chain of recalculations that travel through the system as each module will recalculate its output as soon as the result of the previous module is available. If, however, the modules only react to changes in the clock signal, the result will take one clock cycle per intermediate module to travel through the system as the method will not be triggered when a previous module has finished its calculations.

As this means that an arbitrarily large delay may be introduced in a given system, the machine learning algorithm may not assume that a result is available once all calculations have been performed.

However, unless a module stores a value internally, it cannot delay the calculation and propagation for longer than a single time step.

The current solution to this issue is to feed the machine learning algorithm for a module with the current and the previous signal assignments as inputs. If the machine learning algorithm detects any dependency to the previous time step, the formulas are generated with the additional variables that represent the previous time step. The delay is propagated backwards when merging the trees into formulas, rising with each required delay.

If the stimuli generator then targets a given signal and attempts to assign a given value, it needs to start the according amount of steps in advance, assigning the according variables at the given steps until the result can be read and compared to the expected one.

Internal states

SystemC programs may store values at arbitrary locations in the program. Modules may contain internal variables, the program may contain global variables that are accessed from anywhere, temporarily unreferenced locations in memory may still be used e.g. using pointer arithmetic etc.. In short, for arbitrary C++ it is next to impossible to retrieve a simple, well-defined state for a given object such as a SystemC module.

The simplest solution is to require the program to adhere to certain standards. If all internal values were stored using signals instead of native variables, the stimuli generator finds them via the SystemC API and retrieves the according values without any further intervention.

If, however, this approach cannot be applied (e.g. because an existing design relies heavily on the usage of field variables and was not meant to be refactored), the information can instead be gathered by reading the object's state from memory. While this approach works in many cases, it is easy to come up with corner cases that are hard to cover appropriately: In order to read e.g. a pointer's value instead of merely the memory address, the stimuli generator needs to know the object's structure. While this information can be gathered from the debug symbols [108], the question still remains which of the information is required. Iterating through memory, gathering all states that are somehow accessible for a given module, regardless or being logically related or not, results in a tremendous amount of data that could theoretically influence a module's output, increasing both, the memory footprint and the computation time for the machine learning algorithm.

Example 15 *A SystemC module `dsp` has two inputs: a boolean `setFilter` and an integer `data` input. Usually, the `data` input provides the module with data to process, applying arbitrary signal processing methods. If, however, the `setFilter` bit is set, it is instead supposed to change the filter being applied. While the plan is to later send one of several yet-to-be-defined opcodes to the module to switch its functionality, in this version, the integer being sent with a positive `setFilter`-bit is interpreted as a pointer to a function that is stored and invoked in the following cycles.*

Such a setup relies on an internal state (the stored function reference) that cannot be easily read or interpreted. The machine learning algorithm will thus not be able to properly determine the module's behaviour, as identical inputs to the data port may, over several cycles, result in different output values being computed without the machine learning module having access to the according state.

User-defined datatypes

As the designer may define any arbitrary type to be used for communication on a signal, the stimuli generator may be required to handle these.

While the stimuli generator could provide the required signal assignment by creating an object that simply has the required bits set to a specific value, the current implementation does not offer this feature. There are several reasons against the automatic generation of bitmasks for user-defined types:

- The creation of new instances of these types from bitmasks is an inherently unsafe operation as the resulting object may not adhere to the constraints required by operations that work on this object.
- Pointers cannot easily be detected, so types that contain references to other parts in memory can neither be analysed nor created automatically.
- The size of types may not be known at runtime as C++ is not type-safe and the type information of objects is usually discarded by the compiler.

We therefore suggest to supply the stimuli generator with methods to handle the according types. This way, while requiring the designer to write additional code, the underlying SystemC code base does not need to be adapted. As the different amount of types that are used on signals should be manageable, this can be considered an acceptable trade-off to avoid the issues that arise from an automatic translation of arbitrary types.

Example 16 *A SystemC module is equipped with a single input `sc_in<cat>`, with `cat` being a user-defined class that contains a `char*` reference (called `name`), a boolean value `valid`, another boolean value `purring` and a `int*` reference `hairs`. While it is possible to retrieve this information from e.g. the debug symbols, making use of it is not possible.*

First of all, the `char` and `int` references can be read as such – it is even possible to read the target address to actually retrieve the data. However, the designer has actually used both these references for more than a single value, using `name` as a 0-terminated string and – if the `valid` bit is set – `hairs` as a large (fixed size) integer array to store the colours of each of the cat's hairs. Generating a `cat` instance automatically – as provided by the class description – is very well possible. However, doing so according to the designer's assumptions is not. Creating single `char` and `int` values on the heap is not enough in this case, as the designer has simply assumed both references to adhere to certain standards (i.e. 0-termination or a certain fixed size). Without these, running the program may result in undefined behaviour, possibly resulting in e.g. memory corruption.

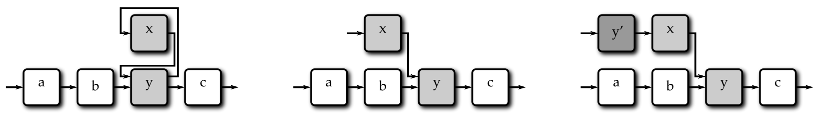


Figure 32: Circles (gray modules, left) are not handled in the machine learning algorithm's first analysis iteration, with the process stopping as soon as any element would appear the second time (center). They are unrolled in a second step as far as required to be on par with the elements going the furthest back in time (dark gray module, right).

It is thus impossible to properly support user-defined datatypes, especially references to the heap with an unknown size.

Circles

SystemC designs may contain circular structures.

This may pose a problem especially with regard to the idea of iterating through the modules and combining their learned models into one large formula over the consecutive time steps.

In order to handle circles appropriately, a method is proposed that builds the model in two steps, illustrated in Figure 32:

First, the machine learning algorithm builds its models per module and iterates backwards through the system to calculate the time delays for its model, *avoiding* any circles. In this step, no circles are handled. While this algorithm calculates a potentially incomplete model, it is guaranteed to terminate. The amount of timesteps for calculating the given output is stored for the next step.

Second, the same algorithm is executed but circles are unrolled over time just like the remaining circuit to a depth that matches the one calculated in the previous step. This means that for a design that requires the stimuli generator to plan n steps ahead, circles are unrolled exactly n steps, enabling the generator to take circular dependencies into account for its dependencies.

Summary

With the given methods combined, a stimuli generator can be written that properly handles a given SystemC design. This generator is implemented as a module that reads all signals of the ESL design it is part of and hooks into all unused input ports before the simulation starts in order to provide stimuli to the system. When a target is met (or missed) it extends the model with the newly added information from the behaviour since the last learning process and generates the next input, which is then fed to the system (over an arbitrary amount of time) until the output can again be observed. It is able to target *specific signals deep within the system* in order to e.g. boost coverage of certain signals *without any a priori knowledge about the system itself*.

Issues for applying C4.5

The proposed solution currently does not handle TLM designs. The TLM framework that has been incorporated into the SystemC standard and its reference implementation not only features generic payloads to travel along the modules' connections but more importantly offers timing modifications in order to be able to more quickly calculate certain values [37]. This leads to designs that "drift apart" concerning the timing, with parts of the model waiting for other parts to catch up. It also is a problem for the suggested time delay computation which requires the model to provide intermediate results at the according points in the simulation time. Extending the algorithm to be able to handle designs using the TLM framework properly therefore remains an open question that may hopefully be handled in a future work.

The usage of a machine learning algorithm that recursively splits a given data set means that the input values need to have some kind of natural order. C4.5 works best when the input data is an enumeration or binary input (e.g. a boolean value or a set of just a few values that are not connected in any way) and is able to handle e.g. floating point values as they can be ordered, values such as strings essentially cannot be handled well. However, due to the domain of hardware design, signals and states usually rely less on such parameters than other programs, allowing the approach to still be applied to a wide variety of designs.

Additionally, the approach relies on being able to actually learn a module's behaviour. A module that implements a function that is supposed to be tough to guess or that contains a lot of inputs, outputs and according interdependencies is inherently hard to learn and thus will not work well with the presented approach. Especially trapdoor functions such as hash value computation or encryption methods are basically incompatible to the given approach: as a hash function serves the purpose of not allowing a specific output to be generated at will, trying to make an Artificial Intelligence (AI) learn to do just that is a rather difficult task at best.

Still, the proposed approach represents an easy-to-use implementation that is close to a "fire and forget" application, allowing the designer to add the given stimuli generator to a design that connects itself as required and generates stimuli to boost a certain fitness criterion as long as it is either connected to a clock or triggered when the system has reached its next state.

CONCLUSION

This chapter illustrated two approaches to SystemC analysis using Machine Learning algorithms.

Section 5.1 illustrated how off-the-shelf algorithms are a viable solution by themselves, giving designers an easily applicable tool to analyse their designs. Section 5.2 then illustrated how the results of these algorithms can be improved by properly preparing the input data, taking SystemC's traits into account to avoid ambiguities in the data set.

While there is certainly room for improvements left (e.g. concerning user-defined data types in the code or performance improvements for the learning algorithm itself), this shows that the method's concept is working well.

6 APPLICATION

Equipped with his five senses, man explores the universe around him and calls the adventure Science.

Edwin Hubble

With various novel methods for SystemC analysis being available from the previous chapters, another core question is what should be done with them.

While information retrieval may be regarded as a valid goal by itself, it also serves as a foundation for various applications. As for most users, the applicability of methods is a core question regarding their usefulness, several of these have been implemented in order to prove that the given algorithms not only work but can be used for real-world purposes.

This Chapter illustrates several applications for the extracted data. Each section in this Chapter focuses on a different one, motivating why the data extraction is useful and serving as an example of the methods' usefulness.

Generally, the applications being used are one of either category:

- Section 6.1 focuses on visualization techniques, allowing designers to inspect their designs visually.
- Section 6.2 illustrates how the data can be used to determine connections within SystemC modules in order to help designers locate the causes of errors and better understand how data flows through their design.
- Section 6.3 details how the approaches can be used to locate certain features within existing projects, down to the line and instance of their implementation.
- Sections 6.4 shows how the data retrieval may be used to validate that a design adheres to the constraints that were set in its specification.
- Finally, Section 6.5 automatically increases the performance of runtime tests by applying the machine learning methods to a stimuli generation tool.

These applications are therefore both, the justification as to why the presented algorithms are needed in the first place and the front end that defines why end-users could and should be interested in the presented techniques. Hence, despite not focusing on "hard" algorithmic research topics, this chapter should be regarded as a crucial requirement that gives meaning to the methods that would otherwise be interesting but ultimately meaningless.

VISUALIZATION

Visualization is an important issue for several tasks during the design process. It can e.g. be used to more rapidly understand a system, to locate errors in running systems, or to illustrate the project's documentation.

Conventional visualization techniques that focus on hardware design do not cover the abstract layers and usually assume that the modeled hardware is the only part of the system that needs to be displayed. ESL designs, however, may have large software parts that cannot be translated to hardware elements and, hence, are not properly visualized thus far. This mixture of hardware and software yields two main questions that need to be addressed for visualizing an ESL design:

1. How can the information needed for this visualization be retrieved and
2. how should the retrieved information become visualized?

The former question is a problem that is especially apparent in SystemC: As C++ does not offer sophisticated reflection or introspection methods, the extraction of program information at runtime is difficult at best. While SystemC acknowledges this problem and provides an API that allows the extraction of `sc_object` instances, user defined types that are instantiated and/or referenced by the simulated system should also be part of a visualization of the system. Also, unlike a hardware system that has a distinct state for each clock cycle, ESL designs may execute certain functions without keeping trace of intermediate states, making the location of certain errors by means of a visualization of clock-accurate states impossible.

The latter question deals with the differences in paradigms between hardware and software design. A hardware system usually consists of a static architecture that changes its states to generate a certain result. The state transitions are often synchronized using a clock, resulting in systems that change their internal state at fixed intervals. ESL systems do not follow such a pattern. Although they model hardware systems that do behave similarly, that behavior is often generated by much less homogeneous patterns. In case of SystemC, the non-restrictive permission of any C++ construct gives the designer the option to use all kinds of behaviors that have no resemblance in classic hardware systems.

Hardware/software co-visualization is a topic that – despite offering vast potential benefits – is highly complex and has only recently been brought up [24]. While there are visualization techniques for both, hardware and software systems, not all methods of both domains may work well together or show a consistent system. Both, the visualization itself and the back-end to retrieve the needed data are non-trivial problems to address, but a more accurate representation of ESL designs might help to grasp the features of a given system more easily.

Having all desired information available, the next step is to properly visualize them. In this section, existing approaches are briefly reviewed before the concepts needed to visualize ESL designs are discussed.

Previous Work

Current visualization approaches focus on either software or hardware designs. For each, there is a variety of methods or standards available.

Software visualizations have to deal with systems that are constantly redesigned at runtime: Object instances, which resemble the concept of a "thing" that does something like a hardware part, are created and deleted at will. However, unlike hardware, the program logic itself mostly follows strictly linear patterns. Although parallel algorithms have started gaining traction with the widespread availability of multi-core systems and have always been a focus of super computing systems, they are still linear patterns that interact at certain points. UML as a standard to design and visualize software systems proposes several vastly different views to grasp all aspects of a software system, all of which statically represent either structure or behaviour. The main notion is that UML is a language that was designed to be printed.

With the advances in computation power that is available on even mediocre systems, more advanced solutions have been proposed: visualizations such as *gource* [15] or CodeCity [120] use 3D engines to display a software system. In order to visualize different properties at once, the CodeCity-metaphor has received attention in the domain of software visualization. Here, different design properties are displayed in different "dimensions" of the visualization, i.e. the number of attributes, methods, and lines of a Java class have been mapped to three-dimensional cubes that represent buildings in a city. Classes from the same package were placed in the same district to emphasize structural interrelation. As an example, Figure 33 shows a picture of a CodeCity taken from [120]. Unproportional looking buildings immediately pinpoint the designer to problematic classes in the software project. The visualization reveals classes that are too complex in terms of code and may better be split into subclasses, or classes that are not well-balanced in terms of their number of attributes to number of methods ratio.

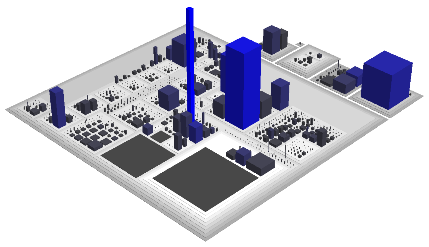
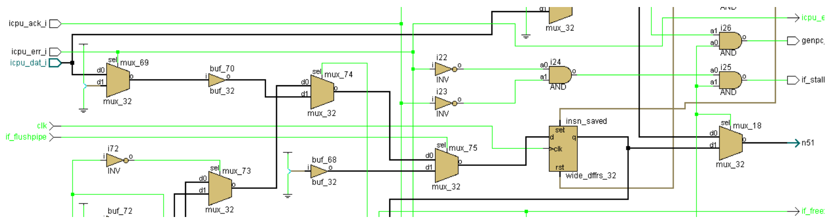


Figure 33: Visualization as CodeCity



(a) RT level visualization

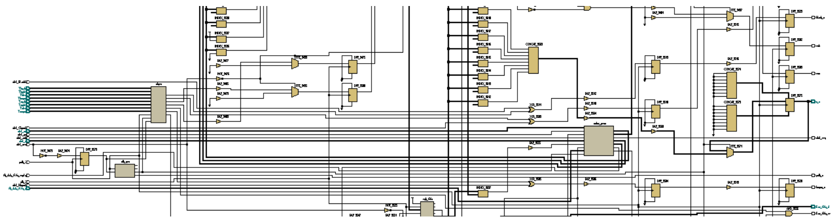

(b) Gate level visualization

Figure 34: Classic hardware visualization focuses on static, printable images of a system

Despite the fact that the CodeCity-metaphor is easily comprehensible for designers, the metrics that have been applied for Java source code cannot be applied directly to system level programs written in SystemC. Beside structural information such as lines of codes, number of signals, number of attributes, number of methods, and connectivity, also quality metrics such as complexity, maintainability, as well as test and verification coverage are of high interest to the designer. In particular, the focus lies in integrating quality metrics into the visualization. In software, e.g. condition complexity [66] measures the number of linear independent paths in a program. For hardware an entropy-based concept has been presented in [69].

Overall, while the CodyCity-metaphor is a proper visualization technique allowing for a multi-dimensional visualization, its concepts need to be redeveloped in order to explicitly support the requirements of ESL designs.

Classic hardware visualization on the other hand usually evolves around established descriptions for the various levels of design. This starts on the transistor level, encompasses the gate level and ends on the register transfer level (Figure 34 provides some examples). All these visualizations evolve around the core concept of hardware: That the system by itself is fixed and only the information being encoded in it is changing. These values are often visualized using waveforms that, while accurate, are not necessarily the best to see connections between and patterns of the signals. While these concepts do represent the hardware appropriately, they are not well suited to illustrate the dynamics of ESL designs.

However, the combined hardware/software co-designed systems at the ESL so far have not been visualized. Only recently, a single work envisioned such systems but did not offer a prototypical implementation [24].

Realization

The goal is to present a working system visualization that displays different visualization schemes in a coherent environment. Even if different concepts require different visualization techniques, they should be an integral part of each other. If viewing both in the same environment is for some reason not feasible, at least going from one to the next should be seamless in both directions. A single visualization for hardware, software, and behaviour is anticipated to avoid repeated swapping of views and to illustrate that the system in question is indeed a single whole and not a collection of separated parts.

The proposed system uses CodeCity [120] as a baseline. The representation of elements using simple geometric shapes (mostly boxes) does not only keep the system requirements low but also all shapes on a common ground level. This simplifies the orientation in the three-dimensional space. Generally, this semi-3D view (three-dimensional objects on a two-dimensional plane) also allows for a simplification of the navigation: the camera requires less degrees of movement freedom to view all objects (in the present case only a two-axes pan motion, a one-axis rotation and a zoom), simplifying the controls to a degree where it is possible to use it on a touch screen without losing any navigation abilities.

Modules themselves are merely instantiated objects, albeit of a particular type. However, other objects that are referenced from the described system should be displayed as well. Apart from the different semantics, there is no real difference between these two, so displaying the objects in a similar manner seems reasonable.

However, presenting all this information at once is usually too much to be displayed on a single screen. Hence, a hierarchical view is applied that uses several levels level of details. By "zooming in", more details of the respective components will blend in. The application of this technique results in an intuitive way to get more information about something in particular: just get closer to it to see more about it.

To visualize important correlations of system metrics, such as lines of code and complexity of each SystemC module, the designer should be able to choose which metrics are important to him and how they have to be visualized, for example as height or ground size of a box. Maintainability could be a suitable metric for this purpose. It reflects the adaptability and modifiability of a SystemC module which is required to correct errors or to improve the performance. The maintainability index as proposed in [119] already provides a proper definition for this. The goal of such an individually customizable system representation is to help the designer to obtain information she needs about certain parts of the system quickly.

Another important part is the behavior of the system. Using time as a dimension to display itself seems a more straightforward solution than the classic idea of timelines or flowcharts, especially when it comes to monitoring running ESL simulations. While displaying all of a simulation's states is out of the question due to the discrepancy between monitor refresh rate and simulation speed, several metrics can be used to analyze and quantize the system changes over a certain timeframe. Such metrics could visualize system changes e.g. by different colors over time. As an example, one can show how often signals are used in a simulation or the individual activity of each module. Furthermore, non-functional properties can be considered if they are available such as power consumption.

Showing the design's behavior in a dynamic visualization for a longer period of time allows the designer to detect correlations among compo-

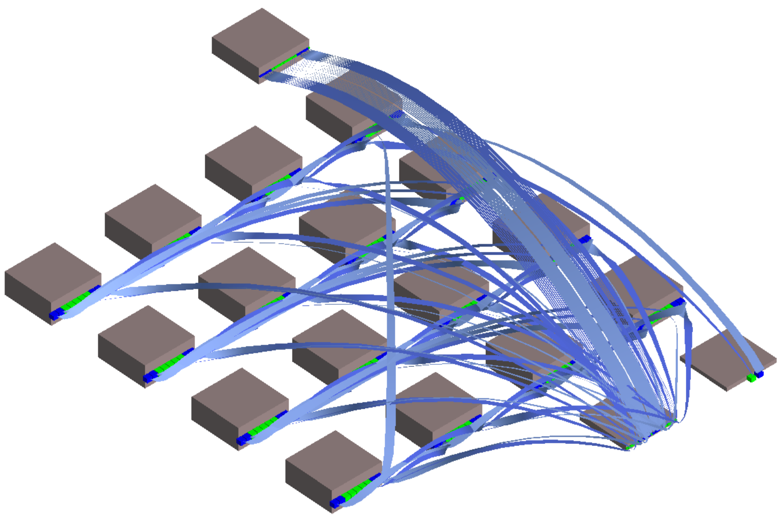


Figure 35: SystemC modules and connections visualized: modules are represented by boxes, signals by connecting lines

nents and signals which help for a better design understanding and to find bugs.

In conclusion, while there are visualization approaches for software and hardware systems, especially the hardware visualizations do not go far beyond classic paper drawings of circuits and therefore do not really make much use of the opportunities a computer-based visualization provides. The visualization of hardware/software co-designed systems which contain a mixture of both has not been done before beyond printable layouts (e.g. in SysML).

A prototype that shows several of the aspects outlined above was implemented using LibGDX [127]. LibGDX is a cross platform framework which allows to run the prototype on various systems.

The data representation, as seen in Figure 35, is fixed concerning its basic structure (e.g. a module is always a gray box), but different attributes can be mapped to the parameters of the given object (e.g. a module's height may represent its memory consumption, the number of lines of code of its corresponding class (as shown in Figure 36) or its class's code complexity). The connectors are smaller boxes attached to their belonging modules. The color of each connector indicates its connector type: Green represents input connectors and blue output connectors.

For the prototype, a simple layout solution that groups sub-modules in squares was used. While this is just a quick and simple solution, it still allows the concept to be illustrated. Also, there is currently no routing solution used: Connections between ports are illustrated using Bézier paths that evade other modules by describing a three-dimensional arc above the ground plane.

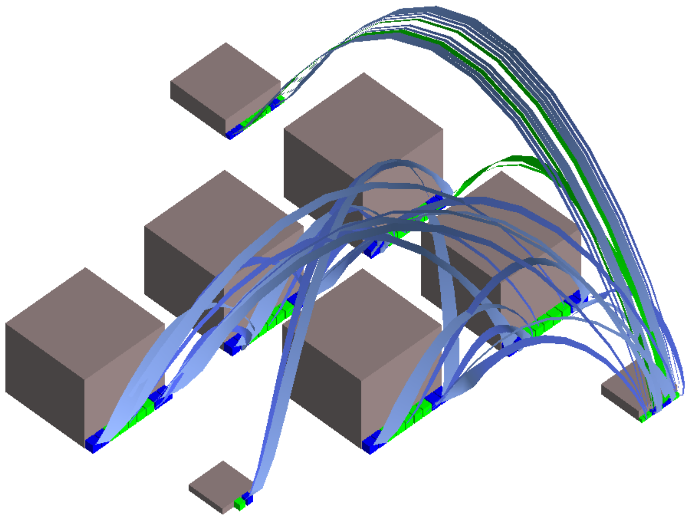


Figure 36: SystemC Design metrics: The height of the modules in this image corresponds to the according class's number of lines of code their base area resembles their respective code complexity

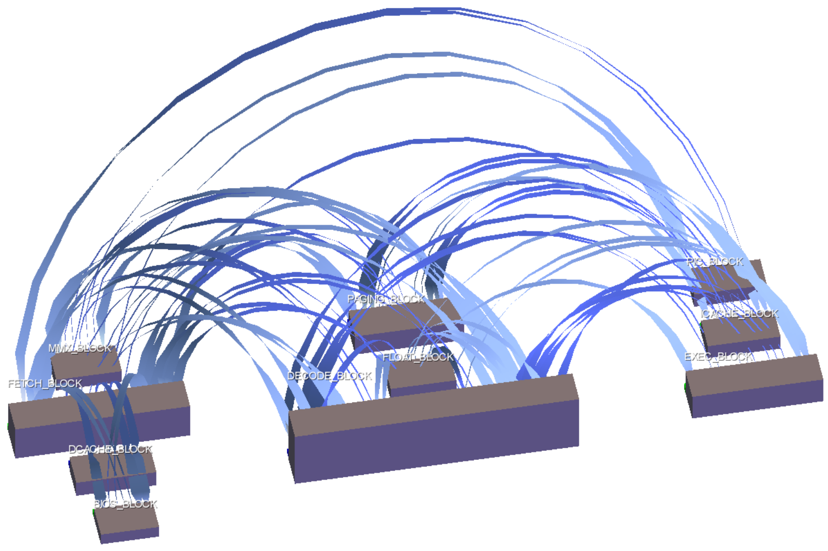


Figure 37: Visualization of the SystemC RISC CPU example. Module height represents the size of a module in memory, module width illustrates its amount of ports.

The behavior of the system can be displayed by using the standard log/dump-files created during a running simulation. The activity of the system can be displayed in real-time while the simulation is running or offline. While this limits the data to be displayed to that which can be extracted using standard dump-file extraction methods (and therefore e.g. excludes custom types), it is universally applicable for any SystemC design and already widely used. Immediate compatibility to existing project setups is in this case an important factor. Using other means to retrieve changes that are usually not part of the manually selected fields and signals would be an obvious next step for the visualization.

While the current state of the implementation does not cover all possible aspects and metrics, it illustrates the first steps towards a comprehensive system visualization for ESL designs. This working proof of concept offers an interactive view that enables the user to navigate through her SystemC design in an innovative and intuitive way. As a result, a new approach is provided which gives the designer a quick overview of a system and aids the design understanding on a different level than source code.

CONE OF INFLUENCE EXTRACTION

Knowing the dependencies within hardware systems has always been a key issue in several design tasks. As a prominent example, Cone of Influence (CoI) analysis is an established method which is heavily applied e.g. in design understanding [40], debugging [1], verification [3, 17, 6], and more. The general idea is thereby to take only those parts of the circuit into consideration that are relevant to the respective design task. As long as circuits and systems are designed and verified at the RTL or the gate level, the extraction of the CoI is simple.

On the ESL, however, the desired system is no longer implemented through a netlist description which is composed of signals connected to components or gates. Instead, the system is implemented in an algorithmic fashion from which, eventually, a binary to be executed is derived. As a consequence, the "classical" structure of a hardware system is no longer available, leading to crucial obstacles for CoI extraction.

Side effects harden the analysis further. For example, if a (partial) CoI contains a global variable, the number of further components possibly also being part of the CoI increases significantly – to *any* other component of the design. Thus, CoI analysis becomes significantly harder at the ESL.

In this section, an approach which addresses this problem is presented. Instead of a structural analysis as commonly applied at RTL and gate level, a behavioral scheme is proposed. That is, stimuli representing various executions of the system under consideration are analyzed. From the results, eventually, the desired information on the CoI for a considered signal are derived.

To this end, methods as outlined in Chapter 5 are being used. As the proposed approach approximates the internal structures of modules, these can be used to in turn retrieve the CoI. To determine how far the approximation differs from the actual CoI, i.e. to evaluate the quality of the approach, several case studies have been conducted.

For stimuli generation, a random scheme has been applied. The case studies have been conducted on a AMD Phenom II X4 machine with 3.4 GHz and 8 GB of memory running Windows 7.

In order to ensure a precise analysis, the proposed approach was evaluated using specifically generated SystemC models that realize arbitrary logic operations. More precisely, SystemC programs are generated that instantiate a module with n inputs and a single output (representing the considered signal for which a CoI shall be determined). The output value is thereby triggered by a randomly generated functional polynom based on an arbitrarily chosen set of inputs. By these means, the exact CoI (which is constituted by the applied monoms of the polynom) is tracked. At the same time, the solution provides a realistic scenario in which the proposed approach can be evaluated.

Table 2 summarizes the results obtained in the case studies. The first two columns provide the number of inputs of the considered SystemC modules as well as the number of their operations. Afterwards, the results of the proposed CoI analysis are presented which have been obtained when either 10, 20, 50, 100, 200, 500, 1,000, 5,000, or 10,000 stimuli were applied. Column 3 respectively denotes thereby the number of input signals which have correctly been identified as being part of the CoI. Column f_p respectively denotes the number of incorrectly classified input signals (i.e. the *false positives*), while column f_n respectively denotes the number of missed input signals (i.e. the *false negatives*). That is, in all cases with $f_p = 0$ and $f_n = 0$, the exact CoI has been determined. In all other cases, too many (if $f_p > 0$) and/or too few (if $f_n > 0$) input signals have been classified to be in the CoI, i.e. an over-approximation and/or and under-approximation resulted. The last two columns provide the number of incorrectly classified input signals ($\frac{f_p}{Inp.}$) and missed input signals ($\frac{f_n}{Inp.}$) as a percentage of the total number of inputs. *All* results reported in Table 2 have been obtained in less than one CPU minute, i.e. in negligible run-time.

The results confirm the discussions from the previous sections. The following conclusions can be drawn:

- Applying machine learning indeed enables an efficient CoI approximation for system descriptions at the ESL. All results have been determined in negligible run-time.

Table 2: Experimental Evaluation of the Machine Learning Approach

		Results for a Number k of Applied Stimuli																									Inaccuracy wrt. Inputs	
		k=10			k=20			k=50			k=100			k=200			k=500			k=1,000			k=5,000			k=10,000				
Inp.	Op.	3	f_p	f_n	3	f_p	f_n	3	f_p	f_n	3	f_p	f_n	3	f_p	f_n	3	f_p	f_n	3	f_p	f_n	3	f_p	f_n	3	f_p	f_n	$\frac{f_p}{Inp.}$	$\frac{f_n}{Inp.}$
5	3	1	1	1	2	0	0	2	0	0	2	0	0	2	0	0	2	0	0	2	0	0	2	0	0	2	0	0	0%	0%
	6	0	0	3	0	0	3	3	0	0	3	0	0	3	0	0	3	0	0	3	0	0	3	0	0	3	0	0	0%	0%
	15	1	0	3	2	0	2	4	0	0	4	0	0	4	0	0	4	0	0	4	0	0	4	0	0	4	0	0	0%	0%
	30	0	0	5	4	0	1	3	0	2	5	0	0	5	0	0	5	0	0	5	0	0	5	0	0	5	0	0	0%	0%
	60	0	0	5	3	0	2	4	0	1	5	0	0	5	0	0	5	0	0	5	0	0	5	0	0	5	0	0	0%	0%
10	3	1	0	1	2	0	0	2	0	0	2	0	0	2	0	0	2	0	0	2	0	0	2	0	0	2	0	0	0%	0%
	6	1	1	3	2	2	2	4	1	0	4	0	0	4	0	0	4	0	0	4	0	0	4	0	0	4	0	0	0%	0%
	15	1	0	7	3	0	5	5	0	3	8	0	0	8	0	0	8	0	0	8	0	0	8	0	0	8	0	0	0%	0%
	30	2	0	7	2	0	7	5	1	4	7	1	2	7	0	2	9	1	0	9	1	0	9	1	0	9	1	0	10%	0%
	60	0	0	10	0	0	10	5	0	5	0	0	10	10	0	0	8	0	2	10	0	0	10	0	0	10	0	0	0%	0%
20	3	2	0	0	2	0	0	2	0	0	2	0	0	2	0	0	2	0	0	2	0	0	2	0	0	2	0	0	0%	0%
	6	0	1	5	1	1	4	5	0	0	5	1	0	5	0	0	5	0	0	5	0	0	5	0	0	5	0	0	0%	0%
	15	1	0	9	2	2	8	4	0	6	9	3	1	9	6	1	10	4	0	10	4	0	10	0	0	10	0	0	0%	0%
	30	1	0	15	3	0	13	6	0	10	11	2	5	12	3	4	16	3	0	16	4	0	16	4	0	16	4	0	20%	0%
	60	2	0	18	3	0	17	3	0	17	5	0	15	11	0	9	13	0	7	17	0	3	17	0	3	17	0	3	0%	15%
50	3	0	0	3	0	0	3	3	0	0	3	0	0	3	0	0	3	0	0	3	0	0	3	0	0	3	0	0	0%	0%
	6	0	0	5	2	0	3	3	2	2	5	4	0	5	0	0	5	0	0	5	0	0	5	0	0	5	0	0	0%	0%
	15	1	0	13	2	0	12	3	3	11	8	11	6	10	8	4	13	23	1	14	25	0	14	23	0	14	18	0	36%	0%
	30	1	0	22	2	0	21	5	2	18	7	3	16	8	8	15	16	9	7	16	13	7	16	17	7	16	17	7	34%	14%
	60	0	0	36	0	0	36	2	1	34	3	2	33	6	5	30	11	6	25	19	5	17	22	9	14	22	9	14	18%	28%
100	3	0	0	3	0	0	3	1	2	2	3	0	0	3	0	0	3	0	0	3	0	0	3	0	0	3	0	0	0%	0%
	6	0	1	6	1	2	5	2	5	4	6	2	0	6	2	0	6	0	0	6	0	0	6	0	0	6	0	0	0%	0%
	15	1	0	13	0	2	14	1	2	13	4	6	10	9	15	5	13	24	1	13	30	1	13	34	1	13	29	1	29%	1%
	30	0	1	26	1	1	25	2	6	24	5	3	21	9	14	17	17	18	9	18	35	8	21	59	5	21	60	5	60%	5%
	60	0	1	48	2	0	46	3	1	45	7	0	41	8	8	40	15	12	33	21	15	27	39	31	9	40	38	8	38%	8%

Inp.: Number of inputs Op.: Number of operations 3: Number of correctly classified input signals
f_p: Number of incorrectly classified input signals (i.e. *false positives*) f_n: Number of missed input signals (i.e. *false negatives*)
Inaccuracy wrt. Inputs: Incorrectly classified input signals ($\frac{f_p}{Inp.}$) and missed input signals ($\frac{f_n}{Inp.}$) as a percentage of the total number of inputs
All results have been determined in less than one CPU minute.

- The quality of the approximations ranges from very good to debatable. However, in more than two thirds of the cases, the *exact* CoI was determined (all entries with $f_n = 0$ and $f_p = 0$).

- In the remaining cases, the results should be distinguished between false positives and false negatives. Any number of false positives ($f_p > 0$) represents an over-approximation, i.e. more signals than necessary are considered. This is unwanted, but not crucial. Much more relevant are false negatives ($f_n > 0$) as they represent missed signals, i.e. signals which are entirely not considered although they influence the considered signal. As can be seen in Table 2, in the few cases where no exact result has been achieved, this number of false negatives is usually small. That is, even if it was not possible to determine an exact result, an approximation that may still be helpful for e.g. giving a rough idea where to look for an error was obtained. Those few cases with a larger f_n are all tied to quite complex functions, indicating that the learning process may need more stimuli to sufficiently retrieve the structures or may be unable to generate an exact representation, maybe due to *C4.5* not performing any backtracking when splitting concerning the wrong variable.

- Finally, the effect of the number of applied stimuli on the quality of the approximations can be observed. The more stimuli are applied, the closer the approximated cone on influence is to the exact one. This particularly holds for larger designs which, obviously, require more stimuli to get better approximations.

Overall, the proposed approach provides good approximations, in many cases even the exact determination, of the desired CoI in SystemC designs.

In order to test the behaviour of the algorithm if some of the variables remain hidden (e.g. because they are internal states of a library and cannot be logged), the same test runs were also executed with $min(\frac{inp}{5} \cdot 2, \frac{op}{3} \cdot 2)$ of the used variables remaining hidden to the ML algorithm. While the results suffer from an increasing number of false positives as the amount of applied stimuli increases, the false negatives remain unaffected (according to the Wilcoxon signed-rank test [121]). This complies with the aim to keep the amount of missed signals low.

FEATURE LOCALIZATION

Current chip designs are becoming more and more complex. As designs tend to shift towards *System-on-Chips* (SoCs) and even *Networks-on-Chips* (NoCs), both, the number of features realized in a single design and the number of atomic elements needed to realize that functionality is growing significantly. As a consequence, such designs are increasingly realized through the re-use of existing parts and external *Intellectual Property* (IP) [64]. These parts may even form a hierarchy, with complex functionality being implemented in blocks that, in turn, are composed of several blocks themselves. This results in a complex functionality being implemented in various layers across the design. Additionally, more designers are usually collaborating to work on a single design. As designs get larger, a separation of concerns is usually carefully organized for both, the design and the people working on it – designers are not working on every part of the design but focus on their specific parts. Thus, designers are enabled to work in large teams on a single system [22].

This development has long passed the point where a designer working on a project is able to know all the details about all the parts present in the design. At the same time, they will not be able to design all components by themselves from scratch anymore. Consequently, designers e.g. frequently work on components which they did not create. This is a severe problem, since designers frequently need to work on parts of an implementation that they are not familiar with. Hence, establishing the needed *design understanding* as quickly as possible is crucial [50].

Usually, a well-written documentation is supposed to be the first measure to transfer the knowledge needed to perform a particular task. But since people tend to need different means for understanding and resources for documentation are usually limited, alternatives are needed. Methods for *feature localization* provide such an alternative. They aid designers by pinpointing them to distinguished characteristics of a design and thus allow them to quickly locate implementations of certain features of a system. Supported in that way, the designer avoids a manual inspection of large parts of the design and can directly focus on those parts that matter for the currently considered design task.

Previous Work

Several methods for feature localization have been proposed (see e.g. [28, 59, 60]). The underlying techniques usually involve running several simulations which are marked as triggering certain features while tracing which parts of the design were used in the respective run. Based on that information, the implementation of a given feature can usually be located in the given implementation. But while these feature localization techniques provide an effective way to direct a designer to the part of the design that is relevant for the current task, most of the existing implementations can only be applied to designs at the RTL.

However, in order to meet the demand of shorter development cycles and working prototypes early in the design process [92], systems are increasingly also designed at the more abstract ESL – which motivates the need for feature localization for this abstraction level. Unfortunately, corresponding support for implementations in SystemC – the current de-facto language at the ESL [96] – is very limited. The only approach for feature localization at the ESL has been proposed in [70]. Here, no support for dynamically generated designs is provided.

Example 17 *Consider a CPU which shall be extended so that it additionally is capable of performing a vector multiplication operation. For this purpose, the ALU needs to be extended. If the designer in charge is not familiar with the given design (which might be composed of thousands of lines of code), pinpointing her to the respective parts in the implementation would significantly support her during the development. In the considered scenario, useful features to be located in the design might be the multiplication of integer or floating point values.*

Identifying those features and hence getting a sufficient understanding of the design in acceptable time is a cumbersome task. Often, the documentation is not as detailed as needed or became obsolete due to changes in the implementation that have not entirely been propagated to all parts of the documentation [83]. As a consequence, *feature localization* is a crucial step within the design process which (up until now) has mainly be conducted manually (e.g. by inspecting the HDL implementation). This added a time-consuming and, hence, cost-intensive step into today's design flows in which neither any new functionality is added nor a single bug is fixed.

In order to aid this process, researchers developed automatic methods for feature localization (see e.g. [28, 59, 60]). These means aid the designer by automatically locating features based on so-called *Coverage Items* (CIs), i.e. parts of the implementation whose execution can be tracked. If the execution of these CIs is tracked, a designer can easily check whether a particular feature does or does not depend on the respective parts of the implementation. More precisely, if an execution (or a *run*) triggering a CI includes the feature the designer is looking for, he/she can conclude that the respective parts of the implementations may relate to the considered feature. Performing several runs, possible CIs and, by this, responsible parts of the implementation can further be refined.

Example 18 *Consider again the scenario from Example 17. The designer wants to add the vector multiplication and, for this purpose, needs to locate the implementations related to floating point multiplication. When running corresponding tests, obvious CIs are triggered which point to the implementation of fetching and decoding of instructions and data, respectively. Performing further runs that include integer multiplication and floating point addition unveils that most CIs are also triggered in runs that do* not *perform the multiplication. From that, the designer can conclude that there are items in the ALU which are triggered if and* only if *a multiplication is executed: there is a distinct set* *of CIs that are executed for the floating point multiplication but not for floating point addition or integer multiplication. Hence, when extending the CPU, he/she should probably investigate these items of the ALU first.*

Overall, methods for feature localization narrow down the items, i.e. the parts of the implementation that need further inspection to a tiny fraction. They thus significantly aid designers to quickly determine the relevant parts of the implementation in order to conduct their improvements, extensions, or bugfixes.

Existing approaches for feature localization of SoCs emerged from the software domain [28] and mainly focused on the RTL and its corresponding programming languages such as VHDL [59, 60]. Methods for the analysis of code coverage form the foundation for almost all approaches for feature localization [122, 94]. Automatic feature localization for SystemC is limited significantly – particularly due to the missing support of dynamic behavior in SystemC.

In fact the only approach for feature localization at the ESL has been proposed in [70]. Here, motivated by the fact that SystemC is purely based on C++ [82], existing C++ coverage tools such as *gcov* and the respective coverage metrics have been applied. This approach is restricted to the static code description of the given SoC though. The dynamic behavior supported by SystemC – particularly differentiating between multiple instantiations of the same type of class – are not supported.

This represents a severe limitation since different modules, even when they have been derived from the same type of class, may implement different features of a system. Moreover, although these instances are dynamically created at run-time they are treated as a static component of the given system. This is because SystemC divides the execution of a system into a so-called elaboration phase and the simulation phase [7]. The former allows the designer to create the design (using any C++ description means he/she deems necessary), while the latter performs the actual simulation but does *not* allow any further modifications of the design. Hence, components might dynamically be instantiated during the elaboration phase, but are treated as static components during simulation. Previous approaches, entirely relying on a static source code analysis, are not capable of distinguishing between a type of class and its instantiations. As a consequence, features might not be trackable.

Example 19 *Consider the simplified representation of the* pkt_switch *system, one of the standard examples which are provided by SystemC, as shown in Figure 21. The switch in the center distributes data and is connected to four senders and receivers which generate and receive arbitrary packages, respectively. The senders (receivers) are instances of a respective class* `sender` *(*`receiver`*) and, therefore, rely on the same source code.*

This leads to severe problems for automatic feature localization which entirely relies on a static view of the source code. In fact, those methods are unable to *differentiate between a feature that is statically defined in the source code and a feature that is dynamically defined through the instantiation within the elaboration phase. As an example, those approaches would not be able to differentiate between the feature "send to 0" and "send to 1".*

Realization

To properly locate features in SystemC designs, a new technique based on the approaches outlined in chapters 3 and 4 is proposed. For this purpose, a smart combination of SystemC/C++ utilities for line coverage analysis such as *gcov* together with the scheme of AOP [99] introduced in Section 4.4 is applied. This way, a hybrid static/dynamic coverage metric emerges that enables the designer to precisely get pinpointed to features in existing SystemC designs. The proposed solution clearly overcomes the limitations of previously proposed approaches, i.e. additionally considers dynamically generated designs, while remaining as non-intrusive as possible and, hence, applicable to a wide variety of SystemC projects.

As feature localization methods require tracing Coverage Items (CIs) in order to retrieve any valid results, the AOP based data retrieval should be working well. The required tracing methods can be injected using AOP – regardless of whether they are the ones introduced in this work, existing ones such as the coverage methods provided by *gcov* or other, custom solutions for tracing the execution of a given system design.

This approach has been implemented and evaluated by a case study. In this section, the results are representatively discussed and compared to the approach presented in [70]. As representative, the `pkt_switch`-system – one of the standard examples in the SystemC-library which has already been considered before in Figure 21/Example 19 – is considered.

The design realizes a system for distributing data packages and is assumed to be instantiated with four `sender` and four `receiver` instances – all of them connected to a central switch. The senders create packages including a random payload which is distributed by the switch (running on a slower clock) to the receivers.

The features a designer may look for in this system may be related to the distribution of data. In particular which parts of the design are triggered when a package is sent to a specific destination might be of interest. For this purpose, features *send to 0*, *send to 1*, *send to 2*, and *send to 3* are defined which are supposed to pin-point the designer to parts of the implementations where the delivery of packages to receiver 0, receiver 1, receiver 2, and receiver 3 are realized, respectively.

For the actual feature localization, five runs are performed. One run simulates the original setup where packages with an arbitrary payload are delivered to an arbitrary receiver. The other runs simulate the delivery to one specific receiver (as there are four receivers, four additional runs are required for this). The feature localization is configured so that only coverage items are reported which are *always* triggered when also the respectively feature was triggered and vice versa. This is sufficient for the considered purpose; however, alternative criteria as discussed e.g. in [29] could easily be considered as well (e.g. being *sometimes* triggered in runs that do not contain said feature).

In the following, results of the feature localization obtained by both, the approach previously proposed in [70] as well as the approach proposed here, are discussed with respect to the runtime performance and quality of the obtained results.

In the current implementation, the coverage data is stored before each termination of a function on disk (as described above by means of Figure 26). This obviously results in a heavy disk usage and, hence, affects the run-time performance. In fact, this increases the run-time for the simulation from less than a second (using feature localization proposed in [70]) to a total of 65.3 CPU seconds (using the proposed feature localization) for the considered `pkt_switch`-system on an Intel i5-3320M CPU at 2.60 GHz with 12 GB memory running Ubuntu 14.04.

While this performance impact is of course a drawback, this does not neccessarily pose a serious issue which questions the applicability of the proposed methodology. This is justified e.g. by the following arguments:

- The total execution time of approximately one minute is still within the boundaries of being applicable in an interactive work process.
- In order to narrow down the coverage of items that are not implementing the desired feature, runs for feature localization are usually rather short. That is, run-times usually do not get significantly larger than the minute reported above.
- If necessary, the amount of information to be tracked and stored can easily be reduced e.g. to a certain namespace that is currently of interest or a set of modules that need to be covered. This would significantly reduce the required runtime and still would provide designers with better results than obtained by the previously applied approach.

Moreover, as shown next, this increase in runtime allows for the determination of results which are of much better quality.

For all considered features, Table 3 provides the results obtained by the approach previously proposed in [70] and obtained by the novel approach. As it can clearly be seen, the previously proposed approach pin-points the designer only to parts of the code related to the switch-module. This can easily be explained by the fact that the dynamically instantiated receivers (where the feature that something is sent to them

Table 3: Comparison between the results obtained by the approach previously proposed in [70] and the one outlined in this section.

Feature	Coverage Items			[70]	proposed
	File	Instance (proposed only)	Lines		
send to 0	receiver.cpp	RECEIVER0	40 – 41; 43; 45 – 51	✗	✓
	switch.cpp	SWITCH	194 – 195	✓	✓
send to 1	receiver.cpp	RECEIVER1	40 – 41; 43; 45 – 51	✗	✓
	switch.cpp	SWITCH	201 – 202	✓	✓
send to 2	receiver.cpp	RECEIVER2	40 – 41; 43; 45 – 51	✗	✓
	switch.cpp	SWITCH	207 – 208	✓	✓
send to 3	receiver.cpp	RECEIVER3	40 – 41; 43; 45 – 51	✗	✓
	switch.cpp	SWITCH	213 – 214	✓	✓

is realized) are not considered by the purely static approach in [70]. In contrast, the proposed methodology does not only pinpoint the designer to the switch but to the respective parts of the receiver as well. Moreover, even the precise instantiation of the receiver is provided (see column denoted by *Instance*). Obviously, this provides a much more comprehensive feature localization.

VALIDATION OF FSL SPECIFICATIONS

The extracted models can e.g. be compared to a given FSL specification. As the static and dynamic information that were extracted from the memory can simply be translated e.g. to block definition diagrams (as already illustrated above), checking for compliance with a given model is simple. Existing modeling frameworks such as the *Eclipse Modeling Framework* can be utilized for this purpose.

Validating the behavior instead requires some manual additions. As the proposed method only extracts program states ("snapshots") for certain points in the execution of the program and is unable to supervise a certain behavior, checking if a certain protocol is adhered cannot be performed fully automatically. In fact, the designer has to explicitly enforce when a snapshot shall be retrieved. If e.g. the behavior is specified by means of a sequence diagram, then a snapshot after each operation call (which changes the state of the system) is appropriate. Thus, the respective states of the implementation are retrieved and can be validated against the specification. This of course neither guarantees that the change in state is a result of the communication nor that the given communication was indeed the only behavior occurring between those states, but valid checks in these places indicate whether an implementation complies with its formal specification.

The suggested method is able to extract a significant amount of information to be used for the validation of the structure and the behavior of an ESL implementation against a corresponding FSL specification. The performance depends thereby on what exactly is extracted. The determination of the static information of the considered arbiter required e.g. approximately one minute on an AMD Phenom II X4 965 with 8GB RAM – most of the time is thereby spent on writing the information onto the disc (with output disabled, the process takes approximately 12 seconds). Retrieving a snapshot, e.g. the one from Figure 17, required approximately 14 CPU seconds. Note that, in the current proof-of-concept implementation, information is stored in an external XML file that needs to be parsed and searched during the dynamic extraction. Hence, further improvements in the performance are very likely but were not in the scope of this work. For a variety of SystemC designs, the respective information was successfully retrieved.

Besides, the proposed solution

- can essentially be transferred to any setup, as long as readable debug symbols and RTTI data are generated by the compiler,
- is able to extract not only the system's modules but also any other C++ objects to which they are connected to,
- relies on the SystemC API to retrieve the objects to be inspected and analyzed, i.e. the user does not have to add any additional code apart from the statement that denotes at which points during the execution a snapshot is required,
- does not add any overhead to the execution until the extraction statements are executed.

The proposed approach has been applied in order to validate the SystemC implementation from [23] against its formal specification. For this purpose, a simple model to state matcher was implemented which utilizes the retrieved information and compares them to the originally given FSL specification. This enabled the detection of the following inconsistencies:

- Additional wrapper modules have been added to the SystemC implementation of the respective cells. Furthermore, the original specification does not contain the module `Scalable_Arbiter` that hosts the cells within.
- The identifiers of the blocks and its corresponding modules did not match. That is, identifiers `Cell 0`, `Cell 1`, ..., `Cell n-1` are used in the specification, while the identifiers `cells`, `cells_0`, `cells_1`, ..., `cells_n-2` are used in the SystemC counterpart. This is a crucial issue as designers might likely map e.g. `Cell 2` to `cells_2` (though it should be mapped to `cells_1`).
- Similarly, the identifiers of the inputs and outputs did not match. The implementation abbreviated the names, i.e. the identifier `override_out` became an `ove_o`.

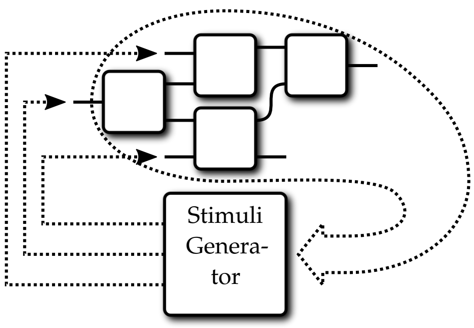


Figure 38: The machine learning feedback loop

- The implementation contains inputs for the clock, several other inputs for the wrapper module, and a `StimGen`-module that was obviously added for testing the setup by generating signal stimuli. All that is not part of the formal specification. In particular, the addition of clock inputs is crucial as this changes the interface of the module.
- A consecutive "snapshot" of the state after each clock cycle showed that the implementation does not make the token of the arbiter travel around the cells. This obviously is a serious design error which needs to be inspected.

After explicitely pin-pointed to it, most of these issues can also be seen by comparing the specification to the extracted data. Of course, these issues are not necessarily errors. However, they clearly emphasize parts of the design which are different from the original specification. In particular, with increasing complexity of the designs, the proposed approach and the according implementation provide a helpful tool which warns the designer about potential discrepancies and possible sources for confusion later on in the design process.

STIMULI GENERATION VIA MACHINE LEARNING

The basic idea of the ESL Coverage Driven Stimuli Generation (CDSG) scheme is to analyse a system's structures in order to be able to generate stimuli that trigger certain signals within the system.

Figure 38 illustrates this principle. While the original system remains untouched, a new module is introduced into the design that observes all other signals that are present. The module attaches itself to all unassigned input ports of the system, allowing it to generate the stimuli that drive the design throughout the simulation. If a certain signal e.g. falls behind for certain coverage criteria, the stimuli generator can then attempt to generate the stimuli that are required to increase the given signal, either

- succeeding, thus improving the results or
- failing, thus improving its own model of the system by taking the new data into account.

For SystemC in particular, due to the difficulties in analysing the design in the first place, but also due to the difficulties in attaching a stimuli generator to an arbitrary design, this scheme is hard to implement. However, the methods suggested in Chapter 5 may be used to realize this scheme despite the difficulties of analysing SystemC, especially at run-time.

Previous Work

As dedicated ESL approaches for AI-supported CDSG are sparse, this section also gives a brief overview over existing RTL approaches.

Several algorithms have been developed to generate test patterns for a system simulation. However, only few of them focus on SystemC in particular: most CDSG approaches focus on the RTL or Gate-Level, generating stimuli for systems that are guaranteed to be synthesizeable.

These approaches all follow the same pattern of establishing a feedback loop that connects the observations of a system's behaviour to new stimuli that feed the system itself. This loop is then used to verify assumptions about the system and refine the model that is generated from the observations [47].

Currently, there are three major classes of approaches to generate test patterns for the devices using machine learning:

- Evolutionary Algorithms (e.g. [124, 16]). These are based on methods that simulate evolutionary processes to optimize a set of individuals against a fitness criterion. The advantage of this approach is that it corresponds nicely to a given coverage value: individuals (corresponding to a certain set of stimuli) that have already gained a better coverage are the ones that are used to generate the next generation of individuals, resulting in an improving performance.
- Probabilistic models such as Bayesian Networks or Markov Models (e.g. [32, 116]). These rely on probabilistic connections between nodes in directed graphs, modelling the probabilities how a system changes its state as labels for transitions between a graph's nodes.
- Data Mining approaches that attempt to build comprehensive models from a vast set of recorded data (e.g. [57]).

The resulting models are then used to generate new stimuli to test the simulated device.

SystemC approaches in contrast focus on the generation of the stimuli themselves, specifically not targeting signals within the model. Frameworks such as the Constrained RAndom Verification Environment (CRAVE) [54] allow for a detailed specification of how the stimuli are supposed to be generated but are not able to generate stimuli that target a specific signal within the model.

While the general approach of watching the execution and generating stimuli based on this performance is portable, algorithms for other levels of abstractions of course use the information available from the according descriptions.

One defining feature of development on the ESL is that the designer does not have to specify a synthesizeable system but may as well leave decisions concerning implementation details for later. Hence, any algorithm focusing on CDSG for the ESL can not rely on information such as the final hardware structure and its connections as this information does not exist yet.

To summarize, existing approaches either target synthesizeable languages in order to be able to set signals within the design or focus on SystemC but do not support setting signals within the system.

Applying Machine Learning for CDSG

To test the given approach, scalable circuits were set up that could be increased in size in order to see how the approach would perform.

treeDelayed$n$ is a tree of SystemC modules, each realizing an arbitrary boolean gate with inverters inserted at random locations. The amount of inputs available to the stimuli generator is 2^n, with the tree consisting of n levels of modules. Each module (including the NOT gates) is connected to a clock and only refreshes its outputs upon a clock tick, resulting in varying amounts of ticks until the signal passed the tree and the result is set. The test terminated as soon as the output signal on the top of the tree was toggled 1000 times.

treeInternal$n$ enriches this test with modules that have an internal state that is toggled with each clock cycle and switches its behaviour between being a NOR and an OR gate. For the machine learning algorithm, this results in a circular dependency, with the signal depending on its own previous state (or the clock, which also depends on its own previous state) that needs to be handled.

The machine learning algorithm usually allows for an unlimited collection of information about the system, which results in a certain overhead when handling the data. Another setup was therefore limited to collecting data from merely 32 cycles, allowing it to handle less data. The former shows that the algorithm itself works, the latter serves as an example to illustrate that a hybrid approach may yield faster results.

Table 4 illustrates the results. While the machine learning algorithm always needs less simulated cycles than the random stimuli generator

Table 4: Experimental Results

benchmark	rnd cycles / s	ml cycles / s	ml_{32} cycles / s
treeDelayed2	3,283 / **0.08**	**2,445** / 1.33	**2,445** / 0.20
treeDelayed4	10,211 / **1.06**	**5,045** / 17.46	6,018 / 1.98
treeDelayed6	40,683 / **17.24**	**9,485** / 242.41	26,340 / 19.15
treeDelayed8	167,799 / 301.90	**15,197** / 2975.78	151,341 / **275.90**
treeInternal2	2,279 / **0.06**	2,101 / 3.45	**2,085** / 0.74
treeInternal4	4,331 / **0.39**	**3,197** / 90.25	3,347 / 8.35
treeInternal6	9,375 / **4.24**	5,733 / 1165.34	**5,449** / 50.97
treeInternal8	14,471 / 25.02	—	—

Amount of simulated cycles / real time until the model's outputs were toggled 1000 times using either random stimuli generation or the proposed method, with unlimited data collection or limited to 32 stimuli sets. Best results are bold.

to reach the toggle goal, it is slower in real time when being allowed to collect all the data it can retrieve from the simulation.

The test that contained internal states for all modules in large structures could not be completed – the interdependencies between these values, including unrolling the timed dependencies over the required amount of steps resulted in quite large structures that resulted in stack overflows when handling the tree structures.

Generally though, the approach works. The toggle coverage, which the algorithm is supposed to increase, grows faster when the stimuli generator is applied to the system – on signals that are only indirectly connected to the generator.

The CDSG approach is able to target certain signals but does so at a cost. Signals that are e.g. toggled regularly by a random stimuli generator may be toggled more quickly by the machine learning module concerning the system's clock, but the computation of the stimuli results in a slower simulation speed, which may result in a real time performance decrease when relying solely on the given approach. The sensible approach thus is to use the proposed method only for signals that are not easily toggled, starting with a broad random generation first, as it is common practice on the RTL [56]. The proposed limitation of input data serves the same purpose, requiring the algorithm to resort to random generation more often.

The bottom line is that the approach is able to target given stimuli at will, bypassing the issue of black boxes inbetween and setting signals within a given SystemC model by learning the modules' behaviour alone. It thus provides an easily applicable solution to analyse a system and trigger a certain behaviour without requiring any manually made system descriptions.

CONCLUSION

In this chapter, several applications for the previously explained analysis methods were shown. Two core use cases were illustrated that SystemC analysis methods can be used for:

- Design understanding applications that use the data to give designers new insights into the given design. These can serve as tools for designers joining a new team in order to “get into” a design and more quickly start developing or for developers that are searching for specific parts of the design, e.g. during the debugging process.
- Automated testing tools that check whether or not a given design complies with certain constraints that have been formulated before. This allows designers to catch errors earlier in the design process, thus lowering the development cost and speeding up the design process.

These applications show that the proposed methods work for SystemC designs and have real-world use cases that are beneficial to the development of ESL designs using SystemC. They thus form a proof-of-concept framework for the beneficial usage of SystemC analysis methods.

7 CONCLUSION

This is the end
My only friend, the end
Of our elaborate plans, the end
Of everything that stands, the end

— Jim Morrison

In this thesis, a novel take on SystemC analysis was presented. Existing methods focus on the analysis of the source code or the adaptation of existing source code processing units such as compilers. While these methods provide an insight into SystemC designs, their applicability is limited by their ability to analyse existing SystemC projects.

The focus of this thesis is to provide designers with methods to analyse arbitrary designs instead. Instead of focusing on the ability to extract more information from a design, the core issue is *non-invasiveness* of the approaches, allowing them to be applied to a broader range of designs. The presented methods can be classified into one of three core approaches.

Three core approaches

STATIC ANALYSIS encompasses all methods that can be applied *before* the given SystemC program is executed, i.e. at compile-time. This refers to the information that can be retrieved from the source code such as the fields of a certain type of module.

The major issue for the retrieval of this information is the diversity of the underlying language. With C++ offering complex description means, various compilers interpreting code differently and vendors each including their own, extended versions of given libraries, even this static information is not trivial to retrieve.

The presented extraction means rely on the compiler pre-processing the source code and storing intermediate information on disk for later debugging purposes, thus gaining additional reliability by using established interfaces and being able to re-use the processing steps done by the compiler that is being used in the existing compilation pipeline. These points set this approach apart from parsers that rely on reading the source code directly and hence suffer from discrepancies between the custom tool that is used to interpret the design and the established one that is used to build an executable simulation framework from it.

With proof-of-concept implementations that work with gcc, clang and MSVC++, the major compilation platforms are already covered, illustrating that the approach works well for a diverse set of tools.

DYNAMIC ANALYSIS encompasses all methods that need to be applied *at run-time*, i.e. after the given SystemC program was compiled and during its execution. The information that is retrieved using dynamic analysis methods thus encompasses data such as e.g. the instances of modules and their variable assignments over time.

There are several major issues for the retrieval of this information. The core problem though is that C++ programs are usually translated into native binary executables and being *stripped of all information that is not required for the execution* in order to make them as small and fast as possible. Most other issues arise from this fundamental design decision of C++, as e.g. memory remains unmanaged and type information is, if present, unreliable and sparse.

The presented methods to still extract the desired information at run-time are divided in two parts.

- The extraction method itself that is used to extract the information from a running program. For this purpose, existing analysis tools may be used. If the information that is retrieved via these is too sparse, a novel approach was presented that relies on the data that is retrieved using the static analysis tools to build models of the information present in memory for given types and is able to extract detailed snapshots of a program at a given time.
- The method that is used to *invoke* the chosen method. An approach using AOP was presented that can be embedded into existing workflows and allows the invocation of the given methods at various points in a given design without altering the underlying source code.

This way, a detailed model of a design's behaviour can be built. Unlike existing methods that focus on the extraction of static structures from a given model, the information thus contains not only dynamically created (but static) SystemC structures but also their behaviour, e.g. allowing developers to more deeply investigate their designs' simulation properties.

MACHINE LEARNING ANALYSIS finally is used for the remaining features that are hard to cover even with the proposed methods. This is mainly used for structural information that cannot be retrieved from the debug symbols (which may contain more or less information depending on the compiler).

While machine learning should always be used with caution as the results may be incomplete or wrong, the proposed methods were shown to generate enough information to improve existing algorithms or give helpful indications concerning design features to a designer. The approach of having an AI assist other algorithms or designers can thus be applied safely in contexts that do not require precise information.

The common denominator for all proposed methods is *non-invasiveness*. None of them rely on specific compilers being used or the source code being altered (or adhering to certain constraints). This eases the application of the approaches as they can be used in conjunction with virtually any SystemC setup.

Apart from this collection of novel methods for the analysis of SystemC designs themselves, various applications were implemented as well. These serve to illustrate use cases and examples as to how the retrieved information could be used in practice.

While the scientific impact of the proposed methods will only show over time, talks about a possible commercial usage of some of the methods are still underway with the German-based Electronic Design Automation (EDA) schematic generation and visualization company Concept Engineering. While this is currently still in its infancy, it illustrates how the issue of SystemC analysis is a very relevant and ongoing topic for both, scientists and commercial players alike.

Open and future questions

Despite the given methods tackling basically all aspects of SystemC analysis, there are of course still several questions left for future research.

First and foremost, the AI, despite its successful application, remains a makeshift solution. Any method that retrieves exact data about a given part of the design should always be preferred to the imperfect approach of resorting to educated guesses. While there are several approaches that may be able to retrieve more precise information (e.g. with clang offering a much more sophisticated interface to its compilation pipeline than e.g. gcc), utilizing these would result in sacrificing the non-invasiveness for more precise results. Hence, while these methods would be worth researching them in their own right, investigating how more information could be retrieved while retaining the non-invasiveness of given results would be a core question for any work that follows the idea of universal applicability.

Another question is whether these methods could be used not only for an extraction of the data but for an interface. As the data that is being extracted currently is either used for offline applications that only utilize the data after the simulation has ended or (in case of the stimuli generation) does not leave the system, inserting an interface to manage a simulation while it is running would be the next step.

As most of the problems that are handled in this work arise from the underlying structure of C++ that is being used as a foundation for SystemC, a rather fundamental question is whether or not using C++ is the right approach or if modern system design should not rather be done on more modern languages. These offer approaches for designing systems using languages that provide detailed reflection and introspection methods, type safety, managed memory etc. SystemC was chosen as the core framework for this work because it is a "de-facto-standard" [7]. While this is a valid argument, especially with regard to the relevance and impact of the proposed methods, it is of course a weak one when considering the fundamental question of whether or not it was a good decision in the first place to pick C++. Despite several arguments that favour C++ (such as hardware designers being used to working on lower abstraction levels, performance, a vast amount of libraries etc.), the core issue remains that several of the analysis issues that are being handled in this work would not even have occurred in more modern languages. Concerning their analysis capabilities, utilizing other frameworks may thus be a more favourable approach. Generally though, in order to even be able to come to a qualified decision, an in-depth comparison of these ESL HDLs would be required.

While these open questions are of course interesting research topics, they neither quite fit the scope nor the topic of this work. Instead, this thesis presented thorough and novel ways to analyse SystemC designs, enabling developers to better understand the design they are working on and providing them with more ways to locate errors or communicate their design decisions.

ACRONYMS

AI	Artificial Intelligence.
ALU	Arithmetic and Logical Unit.
ANTLR	ANother Tool for Language Recognition.
AOP	Aspect Oriented Programming.
API	Application Programming Interface.
AST	Abstract Syntax Tree.
BDD	Binary Decision Diagram.
CDSG	Coverage Driven Stimuli Generation.
CI	Coverage Item.
CoI	Cone of Influence.
CRAVE	Constrained RAndom Verification Environment.
EDA	Electronic Design Automation.
ESL	Electronic System Level.
FSL	Formal Specification Level.
gcc	GNU Compiler Collection.
gdb	GNU Debugger.
HDL	Hardware Description Language.
HJJ	Hardware Join Java.
IC	Integrated Circuit.
IDE	Integrated Development Environment.
ISL	Informal Specification Level.
JavaCC	Java Compiler Compiler.
KaSCPar	Karlsruhe SystemC Parser Suite.
MSVC++	Microsoft Visual C++ Compiler.
PCCTS	Purdue Compiler Construction Tool Set.
RTL	Register Transfer Level.
RTTI	Run-Time Type Inspection.

sc2v	SystemC to Verilog Synthesizable Subset Translator.
SHaBE	SystemC Hierarchy and Behavior Extractor.
TLM	Transaction Level Modeling.
VHDL	VHSIC Hardware Description Language.
VHSIC	Very High Speed Integrated Circuit.

BIBLIOGRAPHY

[1] Moayad Fahim Ali, Sean Safarpour, Andreas G. Veneris, Magdy S. Abadir, and Rolf Drechsler. Post-verification debugging of hierarchical designs. In *International Workshop on Microprocessor Test and Verification (MTV)*, pages 871–876, 2005.

[2] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *Design Automation Conference (DAC)*, pages 1216–1225. ACM, 2012.

[3] Sergey Berezin, Sérgio Campos, and Edmund M. Clarke. Compositional reasoning in model checking. In Willem-Paul Roever, Hans Langmaack, and Amir Pnueli, editors, *Compositionality: The Significant Difference*, volume 1536 of *Lecture Notes in Computer Science*, pages 81–102. Springer Berlin Heidelberg, 1998.

[4] David Berner, Jean-Pierre Talpin, Hiren Patel, Deepak Abraham Mathaikutty, and Sandeep Shukla. SystemCXML: An Extensible SystemC Front End using XML. In *Forum on Specification and Design Languages (FDL)*, pages 405–409, 2005.

[5] Peter Bertelsen. Semantics of java byte code. 1997.

[6] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. *Symbolic model checking without BDDs*. Springer, 1999.

[7] David C Black, Jack Donovan, Bill Bunton, and Anna Keist. *SystemC: From the Ground Up*. Springer, 2004.

[8] Nicolas Blanc, Daniel Kroening, and Natasha Sharygina. Scoot: A tool for the analysis of systemc models. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 467–470. Springer, 2008.

[9] Grady Booch, James Rumbaugh, and Ivar Jacobson. Unified modeling language (uml). http://www. rational. com/uml/(UML Resource Center), 1998.

[10] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Media, Inc., 2008.

[11] Carlo Brandolese, Paolini Di Felice, Luigi Pomante, Daniele Scarpazza, and Ambrosio Goikoetxea. Parsing systemc: an open-source, easy-to-extend parser. In *International Conference on Applied Computing (IADIS)*, pages 706–709, 2006.

[12] Harry Broeders. Extracting behavior and dynamically generated hierarchy from SystemC models. In *Design Automation Conference (*, pages 357–362, 2011.

[13] James Brusey and Lin Padgham. Techniques for obtaining robust, real-time, colour-based vision for robotics. In Manuela Veloso, Enrico Pagello, and Hiroaki Kitano, editors, *RoboCup-99: Robot Soccer World Cup III*, volume 1856 of *Lecture Notes in Computer Science*, pages 63–73. Springer Berlin / Heidelberg, 2000.

[14] Javier Castillo Villar and Pablo Huerta. SystemC to Verilog Synthesizable Subset Translator. http://opencores.org/project,sc2v, 2010. Accessed 2014-12-23.

[15] Andrew H. Caudwell. Gource: Visualizing software version control history. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, pages 73–74, New York, NY, USA, 2010. ACM.

[16] Adriel Cheng and Cheng-Chew Lim. Optimizing System-on-Chip Verifications with Multi-Objective Genetic Evolutionary Algorithms. *Journal of Industrial and Management Optimization*, 10(2):383–396, 2014.

[17] Alessandro Cimatti, Enrico Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Integrating BDD-based and SAT-based symbolic model checking. In Alessandro Armando, editor, *Frontiers of Combining Systems*, volume 2309 of *Lecture Notes in Computer Science*, pages 265–276. Springer Berlin / Heidelberg, 2002.

[18] G De Michell and Rajesh K Gupta. Hardware/software co-design. *Proceedings of the IEEE*, 85(3):349–365, 1997.

[19] David Déharbe and Sergio Medeiros. Aspect-Oriented Design in SystemC: Implementation and Applications. In *Annual Symposium on Integrated Circuits and Systems Design (SBCCI)*, pages 119–124. ACM, 2006.

[20] Rob Dekker. Whats the difference between VHDL, Verilog, and SystemVerilog? http://electronicdesign.com/what-s-difference-between/what-s-difference-between-vhdl-verilog-and-systemverilog. Accessed 2016-04-28.

[21] Encyclopædia Britannica Online. “hardware”. http://www. britannica. com/technology/hardware-computing, 2016. Accessed 2016-04-28.

[22] Ludovic Dibiaggio. Design Complexity, Vertical Disintegration and Knowledge Organization in the Semiconductor Industry. *Industrial and Corporate Change*, 16(2):239–267, 2007.

[23] Rolf Drechsler and Daniel Grosse. Reachability Analysis for Formal Verification of SystemC. In *Euromicro Symposium on Digital System Design*, pages 337–340, 2002.

[24] Rolf Drechsler and Mathias Soeken. Hardware-software co-visualization: Developing systems in the holodeck. In *Proceedings of the 16th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems DDECS*, pages 1–4, 2013.

[25] Rolf Drechsler and Jannis Stoppe. Hardware/software co-visualization on the electronic system level using SystemC. In *International Conference on VLSI Design (VLSID)*, pages 44–49, 2016.

[26] Rolf Drechsler and Jannis Stoppe. Hardware/software co-visualization: The lost world. In *International Workshop on Boolean Problems (IWSBP)*, 2016.

[27] Michael J Eager. Introduction to the DWARF debugging format. http://www. dwarfstd. org/doc/Debugging using DWARF-2012. pdf, 2012.

[28] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Aiding Program Comprehension by Static and Dynamic Feature Analysis. In *International Conference on Software Maintenance (ICSM)*, pages 602–611. IEEE, 2001.

[29] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating Features in Source Code. *Transactions on Software Engineering (TSE)*, 29(3):210–224, 2003.

[30] Yusuke Endoh. Asystemc: an aop extension for hardware description language. In *International Conference on Aspect-Oriented Software Development (AOSD)*, pages 19–28. ACM, 2011.

[31] Görschwin Fey, Daniel Große, Tim Cassens, Christian Genz, Tim Warode, and Rolf Drechsler. ParSyC: An Efficient SystemC Parser. In *12th Workshop on Synthesis And System Integration of Mixed Information technologies*, pages 148–154, 2004.

[32] Shai Fine and Avi Ziv. Coverage directed test generation for functional verification using bayesian networks. In *Design Automation Conference (DAC)*, pages 286–291. IEEE, 2003.

[33] International Organization for Standardization (ISO). ISO/IEC 14882: Standard for programming languages – c++. Technical report, 2011.

[34] Mark A. Friedl, Douglas K. McIver, John C.F. Hodges, X.Y. Zhang, D. Muchoney, Alan H. Strahler, Curtis E. Woodcock, Sucharita Gopal, Annemarie Schneider, Amanda Cooper, Alessandro Baccini, Feng. Gao, and Crystal Barker Schaaf. Global land cover mapping from modis: algorithms and early results. *Remote Sensing of Environment*, 83(1-2):287–302, 2002.

[35] Christian Genz and Rolf Drechsler. System exploration of systemc designs. In *Annual Symposium on Emerging VLSI Technologies and Architectures*, pages 6–pp. IEEE, 2006.

[36] Christian Genz and Rolf Drechsler. Overcoming limitations of the SystemC data introspection. In *Conference on Design, Automation and Test in Europe*, pages 590–593, 2009.

[37] Frank Ghenassia et al. *Transaction-level modeling with SystemC*. Springer, 2005.

[38] Mehran Goli, Jannis Stoppe, and Rolf Drechsler. AIBA: an automated intra-cycle behavioral analysis for SystemC-based design exploration. In *International Conference on Computer Design (ICCD)*, 2016.

[39] Mehran Goli, Jannis Stoppe, and Rolf Drechsler. Automatic equivalence checking for SystemC-TLM 2.0 models against their formal specifications. In *Design, Automation and Test in Europe (DATE)*, 2017.

[40] Daniel Große, Rolf Drechsler, Lothar Linhard, and Gerhard Angst. Efficient Automatic Visualization of SystemC Designs. In *Forum on Specification & Design Languages*, volume 1, pages 646–657, 2003.

[41] IEEE 1076.1 Working Group et al. IEEE standard VHDL 1076.1 language reference manual-analog and mixed-signal extensions to VHDL 1076, 1997.

[42] IEEE 1364-2005 Working Group et al. IEEE standard for Verilog hardware description language 1364, 2005.

[43] Ian G Harris. Extracting design information from natural language specifications. In *Design Automation Conference (DAC)*, pages 1256–1257. ACM, 2012.

[44] Douglas M. Hawkins. The Problem of Overfitting. *Journal of Chemical Information and Computer Sciences*, 35(19):1–12, 2004.

[45] Ivan Herman, Guy Melançon, and M Scott Marshall. Graph visualization and navigation in information visualization: A survey. *Visualization and Computer Graphics, IEEE Transactions on*, 6(1):24–43, 2000.

[46] John Hopf, G Stewart Itzstein, and David Kearney. Hardware Join Java: a high level language for reconfigurable hardware development. In *Field-Programmable Technology (FPT)*, pages 344–347. IEEE, 2002.

[47] Charalambos Ioannides and Kerstin I Eder. Coverage-directed test generation automated by machine learning–a review. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 17(1):7, 2012.

[48] FZI Karlsruhe. KaSCPar - Karlsruhe SystemC Parser Suite, 2012.

[49] Sotiris B Kotsiantis, ID Zaharakis, and PE Pintelas. Supervised machine learning: A review of classification techniques. *Informatica*, 31:249–268, 2007.

[50] Ulrich Kühne, Daniel Große, and Rolf Drechsler. Property Analysis and Design Understanding. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1246–1249. European Design and Automation Association, 2009.

[51] James Lapalme, El Mostapha Aboulhamid, Gabriela Nicolescu, Luc Charest, François R Boyer, JP David, and Guy Bois. ESys.Net: a new solution for embedded systems modeling and simulation. In *ACM SIGPLAN Notices*, volume 39, pages 107–114. ACM, 2004.

[52] James Lapalme, El Mostapha Aboulhamid, Gabriela Nicolescu, Luc Charest, François R Boyer, JP David, and Guy Bois. .NET framework – a solution for the next generation tools for system-level modeling and simulation. In *Design, Automation and Test in Europe (DATE)*, volume 1, pages 732–733. IEEE, 2004.

[53] Chris Lattner. LLVM and clang: Next generation compiler technology. In *The BSD Conference*, pages 1–2, 2008.

[54] Hoang M Le and Rolf Drechsler. CRAVE 2.0: The next generation constrained random stimuli generator for SystemC. *DVCon Europe*, 2014.

[55] Feng Liu, Qingping Tan, Xiaoyu Song, and Naeem Abbasi. Aop-based high-level power estimation in systemc. In *Great Lakes Symposium on VLSI (GLSVLSI)*, GLSVLSI '10, pages 353–356, New York, NY, USA, 2010. ACM.

[56] Lingyi Liu, David Sheridan, William Tuohy, and Shobha Vasudevan. Towards coverage closure: Using goldmine assertions for generating design validation stimulus. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2011.

[57] Lingyi Liu, David Sheridan, William Tuohy, and Shobha Vasudevan. A technique for test coverage closure using goldmine. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 31(5):790–803, 2012.

[58] Zhicheng Liu, Biye Jiang, and Jeffrey Heer. imMens: Real-time visual querying of big data. In *Computer Graphics Forum*, volume 32, pages 421–430. Wiley Online Library, 2013.

[59] Jan Malburg, Alexander Finder, and Görschwin Fey. Automated Feature Localization for Hardware Designs Using Coverage Metrics. In *Annual Design Automation Conference (DAC)*, pages 941–946. ACM, 2012.

[60] Jan Malburg, Alexander Finder, and Görschwin Fey. Tuning Dynamic Data Flow Analysis to Support Design Understanding. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1179–1184. IEEE, 2013.

[61] Inderjeet Mani and George Wilson. Robust temporal processing of news. In *Meeting on Association for Computational Linguistics*, ACL '00, pages 69–76, Stroudsburg, PA, USA, 2000. Association for Computational Linguistics.

[62] Kevin Marquet and Matthieu Moy. PinaVM: a SystemC Front-End Based on an Executable Intermediate Representation. In *Proceedings of the tenth ACM international conference on Embedded software*, pages 79–88. ACM, 2010.

[63] Kevin Marquet, Matthieu Moy, and Bageshri Karkare. A Theoretical and Experimental Review of SystemC Front-ends. In *Forum on Specification & Design Languages*, pages 124–129. IET, 2010.

[64] Grant Martin. Design Methodologies for System Level IP. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 286–289. IEEE, 1998.

[65] Edmar Martinelli, André de Carvalho, Solange Rezende, and Alberto Matias. Rules Extractions from Banks' Bankrupt Data Using Connectionist and Symbolic Learning Algorithms. In *International Conference on Computational Finance*, pages 515–533, New York, NY, USA, 1999. MIT Press.

[66] T. J. McCabe. A complexity measure. *IEEE Trans. Software Eng.*, 2(4):308–320, 1976.

[67] Edward J McCluskey. Minimization of boolean functions*. *Bell system technical Journal*, 35(6):1417–1444, 1956.

[68] George H Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[69] Benjamin Menhorn and Frank Slomka. Design entropy concept: a measurement for complexity. In *Int'l Conf. on Hardware/Software Codesign and System Synthesis*, pages 285–294, 2011.

[70] Marc Michael, Daniel Große, and Rolf Drechsler. Localizing Features of ESL Models for Design Understanding. In *Forum on Specification and Design Languages (FDL)*, pages 120–125. IEEE, 2012.

[71] Microsoft. Debug Interface Access SDK. `http://msdn.microsoft.com/de-de/library/x93ctkx8%28v=vs.100%29.aspx`, 2010.

[72] Microsoft. Program Database Files (C++). `http://msdn.microsoft.com/en-us/library/yd4f8bd1%28v=vs.100%29.aspx`, 2010.

[73] Microsoft. Microsoft extensions to C and C++. `http://msdn.microsoft.com/en-us/library/34h23df8.aspx`, 2013. Accessed 2014-12-29.

[74] Edward F Moore. Gedanken-experiments on sequential machines. *Automata studies*, 34:129–153, 1956.

[75] Gordon E Moore. Cramming more components onto integrated circuits. *Electronics*, pages 114–117, 1965.

[76] Gordon E Moore et al. Progress in digital integrated electronics. In *Electron Devices Meeting*, volume 21, pages 11–13, 1975.

[77] Matthieu Moy, Florence Maraninchi, and Laurent Maillet-Contoz. Pinapa: An extraction tool for systemc descriptions of systems-on-a-chip. In *5th ACM International Conference on Embedded software*, pages 317–324. ACM, 2005.

[78] Matthieu Moy, Florence Maraninchi, and Laurent Maillet-Contoz. LusSy: An open tool for the analysis of systems-on-a-chip at the transaction level. *Design Automation for Embedded Systems*, 10(2-3):73–104, September 2006.

[79] Hassan Mujtaba. Intel xeon e5-2600 v3 "haswell-ep" workstation and server processors unleashed for high-performance computing. http://wccftech.com/intel-xeon-e52600-v3-haswellep-workstation-server-processors-unleashed-highperformance-computing/, 2014. Accessed 2016-04-27.

[80] Zainalabedin Navabi. *VHDL: Analysis and modeling of digital systems*. McGraw-Hill, Inc., 1997.

[81] O.S.C. Initiative. IEEE Standard SystemC Language Reference Manual. *IEEE Computer Society*, 2006.

[82] Preeti Ranjan Panda. SystemC – A Modeling Platform Supporting Multiple Design Abstractions. In *International Symposium on System Synthesis (ISSS)*, pages 75–80. IEEE, 2001.

[83] David Lorge Parnas. Software Aging. In *International Conference on Software Engineering (ICSE)*, pages 279–287. IEEE, 1994.

[84] Terence J. Parr and Russell W. Quong. Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

[85] Terence John Parr, Henry G Dietz, and Will Cohen. *Purdue Compiler-Construction Tool Set*. Purdue University, School of Electrical Engineering, 1990.

[86] Alan J. Perlis. Epigrams on programming. *SIgPLAN Notices*, 17(9):7–13, 1982.

[87] Terry Pratchett and Stephen Baxter. *The Long Earth*. Transworld/-Doubleday, London, 2012.

[88] Nils Przigoda, Jannis Stoppe, Julia Seiter, Robert Wille, and Rolf Drechsler. Verification-driven design across abstraction levels: A case study. In *Euromicro Conference on Digital System Design (DSD)*, pages 375 – 382. IEEE, 2015.

[89] J. Ross Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[90] Martin Ring, Jannis Stoppe, Christoph Lüth, and Rolf Drechsler. Change impact analysis for hardware designs. In *Forum on specification & Design Languages (FDL)*, 2016.

[91] Martin Ring, Jannis Stoppe, Christoph Lüth, and Rolf Drechsler. Change management for hardware designers. In *Workshop on Design Automation for Understanding Hardware Designs (DUHDe)*. IEEE, 2016.

[92] Tero Rissa, Adam Donlin, and Wayne Luk. Evaluation of SystemC Modelling of Reconfigurable Embedded Systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 253–258. IEEE Computer Society, 2005.

[93] Frank Rogin, Christian Genz, Rolf Drechsler, and Steffen Rülke. An integrated systemc debugging environment. In *Embedded Systems Specification and Design Languages*, pages 59–71. Springer, 2008.

[94] Raul Santelices, James A Jones, Yanbing Yu, and Mary Jean Harrold. Lightweight Fault-Localization Using Multiple Coverage Types. In *International Conference on Software Engineering (ICSE)*, pages 56–66. IEEE, 2009.

[95] Kevin Leonard Schneider, Oliver Keszöcze, Jannis Stoppe, and Rolf Drechsler. Effects of cell shapes on the routability of digital microfluidic biochips. In *Design, Automation and Test in Europe (DATE)*, 2017.

[96] Carsten Schulz-Key, Markus Winterholer, Thomas Schweizer, Tommy Kuhn, and Wolfgang Rosentiel. Object-oriented modeling and synthesis of SystemC specifications. In *ASP-DAC 2004: Asia and South Pacific Design Automation Conference 2004 (IEEE Cat. No.04EX753)*, pages 238–243. Ieee, 2004.

[97] Xinxin Sheng and David Thuente. Using decision trees for state evaluation in general game playing. *KI - Künstliche Intelligenz*, 25:53–56, 2011.

[98] Wilson Snyder. SystemPerl. http://www.veripool.org/wiki/systemperl, 2014. Accessed 2014-12-24.

[99] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. AspectC++: An Aspect-Oriented Extension to the C++ Programming Language. In *International Conference on Tools Pacific: Objects for internet, mobile and embedded applications*, pages 53–60. Australian Computer Society, Inc., 2002.

[100] Olaf Spinczyk, Daniel Lohmann, and Matthias Urban. Advances in AOP with AspectC++. In *International Conference on Intelligent Software Methodologies, Tools, and Techniques*, pages 33–53. IOS Press, 2005.

[101] Richard Stallman. Using and porting the GNU compiler collection. In *MIT Artificial Intelligence Laboratory*. Citeseer, 2001.

[102] Jannis Stoppe and Rolf Drechsler. Analyzing SystemC designs: SystemC analysis approaches for varying applications. *Sensors*, 15(5):10399 – 10421, 2015.

[103] Jannis Stoppe and Rolf Drechsler. Ecore model generation from SystemC/C++ implementations. In *Workshop on Design Automation for Understanding Hardware Designs (DUHDe)*. IEEE, 2015.

[104] Jannis Stoppe and Rolf Drechsler. KI-Unterstützung im Systementwurf. *Industrie Management – Zeitschrift für industrielle Geschäftsprozesse (IM)*, 1:21 – 24, 2015.

[105] Jannis Stoppe, Arved Friedemann, and Rolf Drechsler. SystemCDG – AI based coverage driven stimuli generation for SystemC. In *International Workshop on Logic & Synthesis (IWLS)*, 2016.

[106] Jannis Stoppe, Marc Michael, Mathias Soeken, Robert Wille, and Rolf Drechsler. Towards a multi-dimensional and dynamic visualization for ESL designs. In *Workshop on Design Automation for Understanding Hardware Designs (DUHDe)*. IEEE, 2014.

[107] Jannis Stoppe, Robert Wille, and Rolf Drechsler. Cone of influence analysis at the Electronic System Level using machine learning. In *Euromicro Conference on Digital System Design (DSD)*, pages 582 – 587. IEEE, 2013.

[108] Jannis Stoppe, Robert Wille, and Rolf Drechsler. Data extraction from SystemC designs using debug symbols and the SystemC API. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 26 – 31. IEEE, 2013.

[109] Jannis Stoppe, Robert Wille, and Rolf Drechsler. Validating SystemC implementations against their formal specifications. In *Symposium on Integrated Circuits and Systems Design (SBCCI)*. ACM, 2014.

[110] Jannis Stoppe, Robert Wille, and Rolf Drechsler. Automated feature localization for dynamically generated SystemC designs. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 277 – 280. EDA Consortium, 2015.

[111] Deian Tabakov and Moshe Y Vardi. Automatic Aspectization of SystemC. In *Workshop on Modularity in Systems Software (MISS)*, pages 9–14. ACM, 2012.

[112] Donald Thomas and Philip Moorby. *The Verilog® Hardware Description Language*. Springer Science & Business Media, 2008.

[113] Alan M Turing. Computing machinery and intelligence. *Mind*, pages 433–460, 1950.

[114] Dimitri van Heesch. Doxygen: Source code documentation generator tool. URL: http://www.doxygen.org, 2008. Accessed 2014-12-24.

[115] J Craig Venter, Mark D Adams, Eugene W Myers, Peter W Li, Richard J Mural, Granger G Sutton, Hamilton O Smith, Mark Yandell, Cheryl A Evans, Robert A Holt, et al. The sequence of the human genome. *science*, 291(5507):1304–1351, 2001.

[116] Ilya Wagner, Valeria Bertacco, and Todd Austin. Microprocessor verification via feedback-adjusted markov models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(6):1126–1138, 2007.

[117] Jos B Warmer and Anneke G Kleppe. The object constraint language: Precise modeling with UML. 1998.

[118] Alan W. Watts. *The Wisdom of Insecurity*. Pantheon, New York, 1951.

[119] Kurt D. Welker, Paul W. Oman, and Gerald G. Atkinson. Development and application of an automated source code maintainability index. *Journal of Software Maintenance: Research and Practice*, 9(3):127–159, 1997.

[120] R. Wettel, M. Lanza, and R. Robbes. Software systems as cities: a controlled experiment. In *International Conference on Software Engineering*, pages 551–560, 2011.

[121] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945.

[122] Norman Wilde and Michael C Scully. Software Reconnaissance: Mapping Program Features to Code. *Journal of Software Maintenance: Research and Practice*, 7(1):49–62, 1995.

[123] Robert Wille, Daniel Große, Finn Haedicke, and Rolf Drechsler. Smt-based stimuli generation in the systemc verification library. In *Advances in Design Methods from Modeling Languages for Embedded Systems and SoCs*, pages 227–244. Springer, 2010.

[124] Xiaoming Yu, A. Fin, F. Fummi, and E.M. Rudnick. A fenetic testing framework for digital integrated circuits. In *International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 521–526, 2002.

[125] Jun Yuan, Adnan Aziz, Carl Pixley, and Ken Albin. Simplifying Boolean constraint solving for random simulation-vector generation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 23(3):412–420, 2004.

[126] Jun Yuan, Carl Pixley, and Adnan Aziz. *Constraint-Based Verification*. Springer, Secaucus, NJ, USA, 2006.

[127] Mario Zechner and Robert Green. What's next? In *Beginning Android 4 Games Development*, pages 647–651. Springer, 2011.

[128] Jianlong Zhong and Bingsheng He. Medusa: Simplified graph processing on GPUs. *Transactions on Parallel and Distributed Systems*, 25(6):1543–1552, 2014.

[129] Zhi-Hua Zhou and Yuan Jiang. Medical diagnosis with C4.5 rule preceded by artificial neural network ensemble. *IEEE Transactions on Information Technology in Biomedicine*, 7(1):37 –42, march 2003.

[130] Conrad Zuse. museum.informatik.uni-kl.de: Äußerer und innerer Kreislauf – Rechenwerk. `http://museum.informatik.uni-kl.de/Rechner/Zuse/Z11/Schaltbilder/`, 1958. Accessed: 2015-11-29.