Scoot: A Tool for the Analysis of SystemC Models

Nicolas Blanc[1], Daniel Kroening[2], and Natasha Sharygina[3]

[1] ETH Zurich, Switzerland
[2] Oxford University, Computing Laboratory, UK
[3] University of Lugano, Switzerland

Abstract. *SystemC* is a system-level modeling language implemented as a C++ library, and offers support for concurrency and arbitrary-width bit-vector arithmetic. Due to the complexity of C++, current static analyzers for *SystemC* consider only small fragments of the language. We present Scoot, a model extractor for *SystemC* based on a C++ front-end. The models generated by Scoot can serve several purposes, ranging from verification and simulation to synthesis. Exemplarily, we report results indicating that our tool can be used to improve the performance of dynamic execution up to a factor of five.

1 Introduction

SystemC is a system-level modeling language implemented as a C++ library. It offers support for concurrency and arbitrary-width bit-vector arithmetic. Along with an event-driven simulation environment, the library provides a notion of timing, which is well-suited for modeling circuits. *SystemC* permits describing a system at several levels of abstraction, starting at a very high-level functional description, down to synthesizable gate-level. Due to the complexity of C++, current static analyzers for *SystemC* consider only very small fragments of the language, essentially searching for specific key-words. We present Scoot, a model extractor for *SystemC* based on a C++ front-end. This enables us to support a much wider range of language constructs. The models generated by Scoot can serve several purposes, ranging from verification and simulation to synthesis. On the formal side, the tool is tightly integrated with verification back-ends for Bounded Model Checking (CBMC) [6] and SAT-based predicate abstraction (SATABS) [4]. Results on applying model checking to models generated by Scoot have been reported before [7].

As an example of the utility of Scoot beyond formal verification, we report results indicating that our tool can be used to improve dynamic-execution performances up to five times, a speedup usually not obtainable without a formal model.

2 Overview of Scoot

A *SystemC* program consists of a set of *modules*. Modules may declare processes, ports, internal data, channels and instances of other modules. Processes imple-

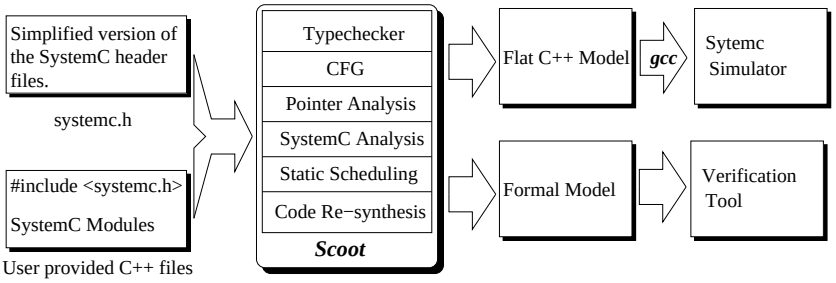

Fig. 1. Overview of Scoot

ment the functionality of the module, and are sensitive to events. As in Verilog or VHDL, ports are objects through which the module communicates with other modules. Although variables are shared between processes, classic interprocess communication is achieved through predefined channels such as signals and FIFOs.

Scoot uses a C++ front-end to translate the *SystemC* source files into a control flow graph. The nodes of the graph are annotated with assignments and guards (implemented in the typechecking and CFG-conversion phases in Figure 1). Subsequently, static analysis techniques are used to determine to the following information, which is specific to *SystemC*:

- The module hierarchy,
- the sensitivity list of the processes, and
- the port bindings.

The *SystemC* library makes heavy use of virtual functions and dynamic data structures, which are not easily analyzed by static analysis techniques. Scoot abstracts implementation details of the library by using simplified header files that declare only relevant aspects of the API and omit the actual implementation. Systems described using *SystemC* shall provide a *sc_main* procedure for building the module hierarchy.

3 Static Scheduling for Dynamic Verification

Technically, *SystemC* modules are plain C++ classes that can be compiled and linked to a runtime scheduler, providing thus a way to simulate the behavior of the system. The model hierarchy is discovered at run-time only and therefore, the compiler is missing opportunities to take advantage of this knowledge. To illustrate the utility of the model generated by Scoot, we re-synthesize more efficient C++ code from the model.

SystemC has a *co-operative multitasking* semantics, meaning that the execution of processes is serialized by explicit calls to a `wait()` method and that threads are not preempted. The scheduler tracks simulation time and *delta cycles*. The simulation time corresponds to a positive integer value (the clock),

while delta cycles are used to stabilize the state of the system. A delta cycle consists of three phases: *evaluate*, *update*, and *notify*.

1. The evaluation phase selects, from the set of runnable processes, a process and triggers or resumes its execution. The process runs immediately up to the point where it returns or invokes the wait function. The evaluation phase iterates until the set of runnable processes is empty. The order in which processes are selected from the set of runnable processes is implementation-defined.
2. In order to simulate synchronous executions, processes can delay change-of-state effects by scheduling update requests. After the evaluation phase terminates, the kernel executes any pending update request. This is called the update phase. Typically, signal-assignments are implemented using the update mechanism. Therefore, signals keep their value for a whole evaluation phase.
3. Finally, during the delta-notification phase, the scheduler determines which processes are sensitive to events that have occurred, and adds all such process instances to the set of runnable processes.

Delta cycles are executed until the set of runnable processes is empty. Subsequently, the simulation time is increased, and processes waiting for the current time-event are notified.

Formally, let S represent the set of states of a *SystemC* model. We write Up to denote the function from 2^S to 2^S that updates a set of states as described by the update phase. Similarly, let $Ev : 2^S \to 2^S$ denote the evaluation phase. The delta phase performs a fix-point computation defined by $\delta(S) = \delta \circ Up \circ Ev(S)$. Finally, we concisely express the semantics of the scheduler with the function $Sim(t) = \delta \circ Up_{time} \circ Sim(t-1)$ that computes the set of final states at a time t. Note that Up_{time} updates the clock.

The standard *SystemC* scheduler contains several sources of inefficiency: first, the scheduler stores data in containers that allocate memory at run-time, and second, it triggers processes using function pointers. SCOOT generates a completely static scheduler by fixing the evaluation order of the processes and resolving dynamic calls. Finally, processes are sequentialized using a similar technique used by KISS [8].

Code Re-synthesis The intermediate representation used by SCOOT was originally designed for model checking, and uses bit-vector arithmetic expressions. After static scheduling, SCOOT translates the intermediate representation back to a flat C++ program that does not rely anymore on the *SystemC* library. The generated model is subsequently passed to `g++`, which results in a faster simulator.

The following table quantifies the advantages of static scheduling compared to dynamic scheduling. We use an AES encryption/decryption core as benchmark. For each module, we report the number of processes, the number of signals, the execution time with dynamic scheduling, the execution time using SCOOT, and the speedup obtained.

Module	Nbr. Proc.	Nbr. Sig.	Dyn. Sched. [s]	Stat. Sched [s]	Speedup
Byte_Mixcolum	2	7	22.94	4.33	5.3
Word_Mixcolum	7	16	65.82	18.01	3.65
Mixcolum	11	30	75.7	28.6	2.65
Subbytes	15	30	49.73	9.84	5.05
128-bits AES	32	97	319.2	91.17	3.48
192-bits AES	32	99	344.21	105.45	3.26

Related Work Due to the complexity of the C++ language, the development of any tool for *SystemC* is a difficult task. Hardware synthesis tools for *SystemC* only consider a small subset of the C++ syntax [5]. In [3], Castillo and Huerta describe a tool that translates *SystemC*-RTL descriptions to synthesizable Verilog. Inversely, VERILATOR [1] is a tool for converting Verilog specifications to *SystemC* or C++ programs. The goal is to achieve higher simulation performance. Other work, such as [2, 9], formalizes the semantics of *SystemC* without providing an actual tool. In [7], we describe a method to automatically distinguish the hardware and software parts of a design.

4 Conclusion

We provide a tool that extracts formal models from *SystemC* code. The tool supports a broad subset of the language, as it is built on top of our C++-front-end. The main applications are formal analysis, e.g., by model checking and synthesis. Exemplarily, we show that formal models have value even in dynamic verification: we show a significant improvement in simulation performance by using a statically scheduled model.

We are continuing to improve the *SystemC* support of our tool. It currently handles the most commonly used features of the *SystemC* API. We are also investigating additional formal techniques to further enhance static scheduling.

References

1. Verilator: http://www.veripool.com/verilator.html.
2. G. Agosta, F. Bruschi, and D. Sciuto. Static analysis of transaction-level models. In *DAC '03: Proceedings of the 40th conference on Design automation*, pages 448–453, New York, NY, USA, 2003. ACM Press.
3. J. Castillo, P. Huerta, and J. I. Martnez. An open-source tool for systemc to verilog automatic translation. In *II Southern Conference on Programmable Logic (SPL2006)*, volume 37, pages 53–58. Latin American Applied Researched, 2007.
4. E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, volume 3440 of *Lecture Notes in Computer Science*, pages 570–574. Springer Verlag, 2005.
5. N. Kostaras and H. T. Vergos. Syce: An integrated environment for system design in systemc. In *RSP '05: Proceedings of the 16th IEEE International Workshop on Rapid System Prototyping (RSP'05)*, pages 258–260, Washington, DC, USA, 2005. IEEE Computer Society.

6. D. Kroening, E. Clarke, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of DAC 2003*, pages 368–371. ACM Press, 2003.
7. D. Kroening and N. Sharygina. Formal verification of systemc by automatic hardware/software partitioning. In *MEMOCODE*, pages 101–110, 2005.
8. S. Qadeer and D. Wu. Kiss: keep it simple and sequential. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 14–24, New York, NY, USA, 2004. ACM Press.
9. J. Ruf, D. Hoffmann, J. Gerlach, T. Kropf, W. Rosenstiehl, and W. Mueller. The simulation semantics of systemc. In *DATE '01: Proceedings of the conference on Design, automation and test in Europe*, pages 64–70, Piscataway, NJ, USA, 2001. IEEE Press.

A Availability of The Tool

The benchmarks and the tool are available from "todo: address".

B Talk Proposal

B.1 Overview of *SystemC*

The first part of the presentation provides background information about *SystemC*. We also explain our motivation to build a tool for the static analysis of *SystemC* models.

The architecture of the *SystemC* language is shown in Figure 2. *SystemC* is essentially a *C++* library with concurrency support. The library offers a set of data types useful for hardware modeling and certain types of software programming. These include 2-valued and 4-valued bit-vectors of arbitrary width, and fixed-point representations. Finally, the library also includes a set of built-in primitive channels such as signals and FIFOs.

Existing validation techniques for *SystemC* rely on simulation and testing, and thus, lack formal guarantees. Current state-of-the-art model checking approaches either target software programs or hardware systems, but fail to bridge both development worlds. Traditionally, formal verification tools are specialized for hardware or software, and therefore, are inadequate to validate the behavior of a hybrid system. Furthermore due to the complexity of C++, current static analyzers for *SystemC* consider small fragments of the C++ language. SCOOT is a step toward a unified verification approach as it extracts formal models from *SystemC* code that can subsequently be passed to verification tools.

The left part of Figure 3 depicts a standard design methodology, which starts from a model of the system written in C or C++. The behavior of the system is refined up to the point where the partitioning between the hardware and software components is decided. Subsequently, a branch occurs in the development process, and the verification for software and hardware parts is performed independently. As a result, the branches may unexpectedly diverge, corrupting the correctness of the global system.

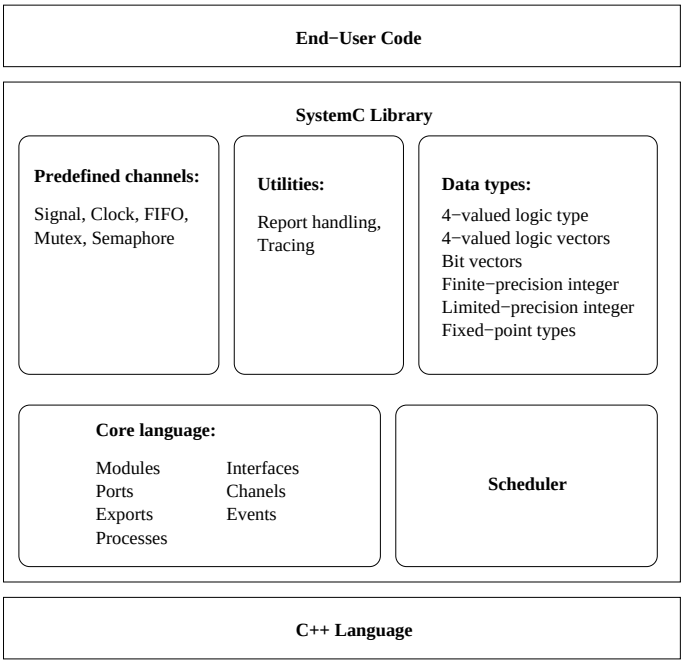

Fig. 2. The *SystemC* language architecture

The right part of the figure depicts a design methodology that uses *SystemC* at every stage of the development. The process starts with a system-level model, which is refined until the partitioning is performed. Since *SystemC* can be used to model circuits, both hardware and software components are described using the same language, and thus, can be refined and verified conjointly. The consistency of the global system can be tested at any time.

B.2 Illustrating example

Scoot permits to re-synthesize C++ from the extracted *SystemC* model, which is an important feature to test the correctness of the extraction. Thanks to the static scheduling of processes, we observe that Scoot generates binaries that run significantly faster than the original ones. In the presentation, we report results indicating that our tool can be used to improve dynamic-execution performances up to five times.

We motivate our approach using the benchmarks from the paper. First, we provide a brief presentation of the 128bits-AES example (Figure 4).

We show how to compile and run the example using the standard approach:

```
> g++ -O2 aes.cpp ...  -I $SYSC/include -L $SYSC/lib \
                              -lsystemc -o systemc_sim
> ./systemc_sim
```

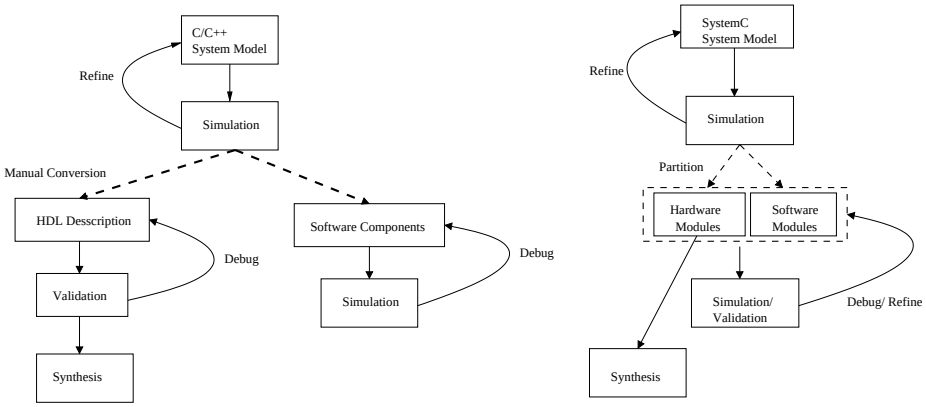

Fig. 3. Different design methodologies

Subsequently, we demonstrate how to run SCOOT on the same example:

```
> scoot aes.cpp ... libsystemc.cpp -I $SCOOT/include --dump-cpp-code
> g++ -O2 scoot_out.cpp -I $SCOOT/bv -o scoot_sim
> ./scoot_sim
```

Finally, the execution times of the two binaries are compared:

```
> time ./systemc_sim
> time ./scoot_sim
```

B.3 Overview of Scoot

In this part of the talk, we present SCOOT using Figure 1. Additionally, we provide an overview of the simplified *SystemC* header files and the bit-vector-based intermediate representation used by scoot. Finally, we explain how SCOOT re-synthesizes C++ files.

The last part of the talk is dedicated to static scheduling. We explain how SCOOT synthesizes the scheduler and how it sequentializes the processes.

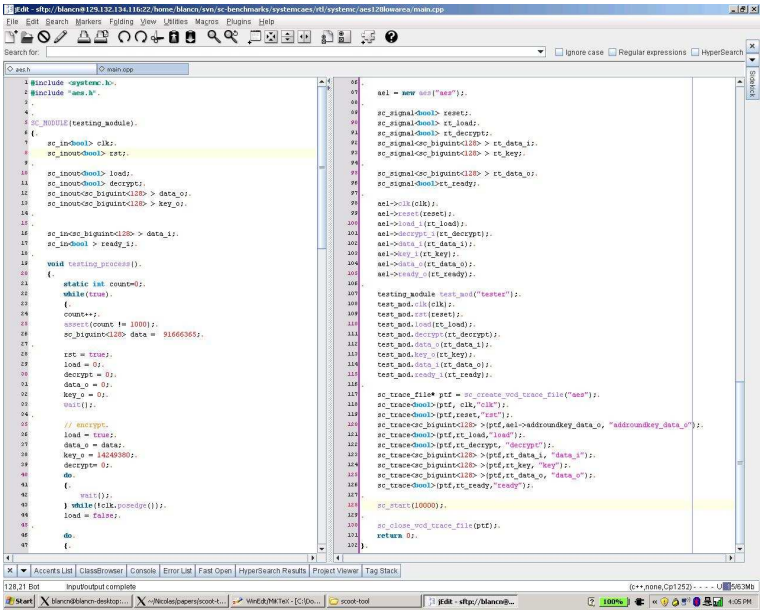

Fig. 4. Main module of a 128-bits AES Core