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 and offers support for concurrency and arbitrary-width bit-vector arithmetic. The existing static analyzers for *SystemC* consider only small fragments of the language. We present Scoot, a model extractor for *SystemC* based on a C++ frontend. The models generated by Scoot can serve multiple 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 high-level functional description, down to synthesizable gate-level. Due to the complexity of C++, existing static analyzers for *SystemC* consider only small fragments of the language, essentially searching for specific key-words. We present Scoot, a model extractor for *SystemC*. The tool sypports a wide range of language constructs, as it based on our C++ front-end. The models generated by Scoot can serve several purposes, ranging from verification and simulation to synthesis. The tool is tightly integrated with verification back-ends for Bounded Model Checking (CBMC) [4] and SAT-based predicate abstraction (SATABS) [2]. Results on applying model checking to models generated by Scoot have been reported before [5].

As an example of the utility of Scoot beyond formal verification, we report results indicating that our tool can be used to improve the performance of dynamic execution up to five times.

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

* Supported by ETH research grant TH-21/05-1 and Foundation Tasso, Switzerland.

C.R. Ramakrishnan and J. Rehof (Eds.): TACAS 2008, LNCS 4963, pp. 467–470, 2008.

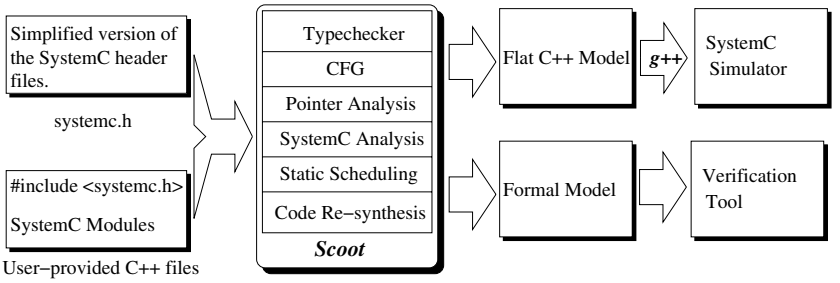

Fig. 1. Overview of SCOOT

implement 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 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.

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. The function 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 [7] that implements context switches with fast goto statements.

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 on the *SystemC* library anymore. 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 on a 3 GHz Intel Pentium 4 processor. 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	# Proc.	# 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	99.73	3.2
192-bits AES	32	99	344.21	105.45	3.26

4 Related Work and Conclusion

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 [3,1]. In [8], Savoiu et al. propose to use Petri-net reductions for *SystemC*, and report a speedup of 1.5 for an AES core. In [6], Pérez et al. present a static-scheduling technique restricted to method processes. Our sequentialization technique extends the benefits of static scheduling to general threads by eliminating the overhead caused by context switches.

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. Castillo, J., Huerta, P., Martinez, J.I.: An open-source tool for SystemC to Verilog automatic translation. In: SPL, vol. 37, pp. 53–58 (2007)
2. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005)
3. Kostaras, N., Vergos, H.T.: SyCE: An integrated environment for system design in SystemC. In: RSP, pp. 258–260. IEEE, Los Alamitos (2005)
4. Kroening, D., Clarke, E., Yorav, K.: Behavioral consistency of C and Verilog programs using bounded model checking. In: DAC, pp. 368–371. ACM, New York (2003)
5. Kroening, D., Sharygina, N.: Formal verification of SystemC by automatic hardware/software partitioning. In: MEMOCODE, pp. 101–110 (2005)
6. Pérez, D.G., Mouchard, G., Temam, O.: A new optimized implemention of the SystemC engine using acyclic scheduling. In: DATE, pp. 552–557. IEEE, Los Alamitos (2004)
7. Qadeer, S., Wu, D.: KISS: keep it simple and sequential. In: PLDI, pp. 14–24. ACM, New York (2004)
8. Savoiu, N., Sandeep, S., Rajesh, G.: Improving SystemC simulation through Petri net reductions. In: MEMOCODE, pp. 131–140 (2005)