ParCoSS: Efficient Parallelized Compiled Symbolic Simulation[*]

Vladimir Herdt[1], Hoang M. Le[1], Daniel Große[1,2], and Rolf Drechsler[1,2]

[1] Group of Computer Architecture, University of Bremen, 28359 Bremen, Germany
{vherdt,hle,grosse,drechsle}@cs.uni-bremen.de
[2] Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany

Abstract. We present the tool ParCoSS for verification of cooperative multithreading programs. Our tool is based on the recently proposed Compiled Symbolic Simulation (CSS) technique. Additionally, we employ parallelization to further speed-up the verification. The potential of our tool is shown by evaluation.

1 Introduction

In this paper we propose our tool *ParCoSS* (Parallelized Compiled Symbolic Simulation) for verification of cooperative multithreading programs available in the *Extended Intermediate Verification Language* (XIVL) format. The XIVL extends the SystemC IVL [15, 11], which has been designed to capture the simulation semantics of SystemC programs [2, 13, 10], with a small core of OOP features to facilitate the translation of C++ code [16]. For verification purpose the XIVL supports computations with symbolic expressions and the *assume* and *assert* functions with their usual semantic. Our tool and set of XIVL examples is available at [1].

Verification of (cooperative) multithreading programs is difficult due to the large state space caused by all possible inputs and thread interleavings. Symbolic Simulation, a combination of symbolic execution [14, 4] and *Partial Order Reduction* (POR) [9, 8] has been shown to be particularly effective to tackle state explosion [6, 5, 15]. Recently *Compiled Symbolic Simulation* (CSS) has been proposed as further improvement [12]. CSS works by integrating the symbolic execution engine and POR based scheduler together with the multithreading program, e.g. available in the XIVL format, into a C++ program. Then, a standard C++ compiler is used to generate a native binary, whose execution performs exhaustive verification of the multithreading program. In contrast to traditional verification methods based on interpretation, CSS can provide significant simulation speed-ups especially by native execution of concrete operations.

The implementation of our tool ParCoSS is based on CSS and additionally supports parallelization to further improve simulation performance. Compared to

[*] This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project EffektiV under contract no. 01IS13022E and by the German Research Foundation (DFG) within the Reinhart Koselleck project DR 287/23-1 and by the University of Bremen's graduate school SyDe, funded by the German Excellence Initiative.

the original CSS approach our tool uses a *fork/join* based state space exploration instead of manually cloning the execution states to handle non-deterministic choices due to symbolic branches and scheduling decisions. A *fork/join* based architecture most notably has the following advantages: 1) It allows to generate more efficient code. 2) It drastically simplifies the implementation.

In particular, we avoid the layer of indirection necessary for variable access when manually tracking execution states and use native execution for all function calls by employing coroutines. Besides very efficient context switch implementation, coroutines allow natural implementation of algorithms without unwinding the native stack and without using state machines to resume execution on context switches. Additionally, manual state cloning of complex internal data structures is error prone and difficult to implement efficiently, whereas the *fork* system call is already very mature and highly optimized. Finally, our architecture allows for straightforward and efficient parallelization by leveraging the process scheduling and memory sharing capabilities of the underlying operating system.

2 Extended Intermediate Verification Language (XIVL)

An example cooperative multithreading program illustrating the core features of the XIVL is shown in Fig. 1. The program is using two threads to compute the sum of odd numbers up to the bound specified by the variable *x*, which is initialized using a symbolic expression of type *int* in Line 2 and constrained in Line 28. The threads synchronize using the *wait* and *notify* functions on the global event *e*. The XIVL syntax resembles C++, supports integer and boolean data types with all arithmetic and logic operators, arrays and pointers, is using high-level control flow structures and has a small set of OOP features including classes, inheritance and virtual methods with overrides and dynamic dispatch.

```
 1 event e;
 2 int x = ?(int);
 3 int sum = 0;
 4
 5 bool is_odd(int i) {
 6   return (i % 2) != 0;
 7 }
 8
 9 thread A {
10   int i = 0;
11   while (true) {
12     wait_event(e);
13     i += 1;
14     if (is_odd(i))
15       sum = sum + i;
16   }
17 }
18
19 thread B {
20   while (x > 0) {
21     x -= 1;
22     notify(e, 0);
23     wait_time(1);
24   }
25 }
26
27 main {
28   assume(x >= 8 && x <= 10);
29   start;
30   assert(sum <= 25);
31 }
```

Fig. 1: XIVL example program

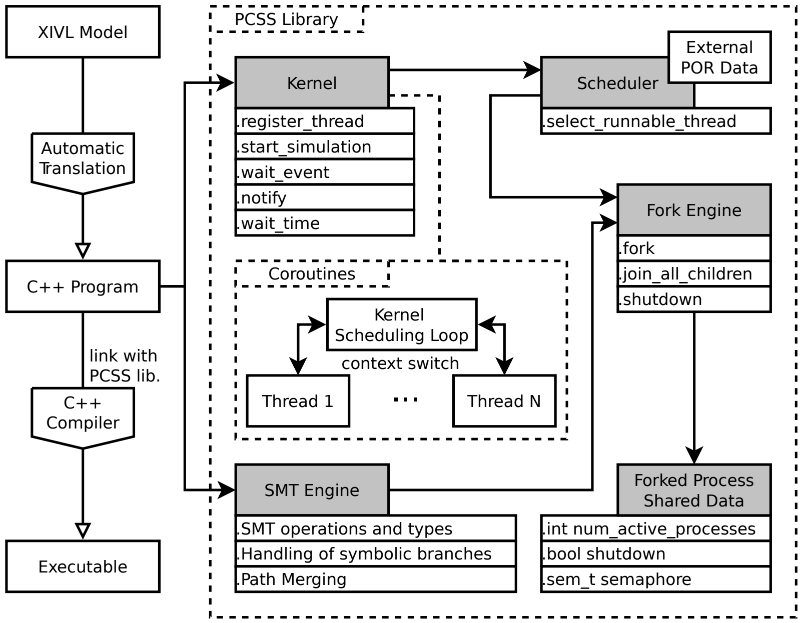

Fig. 2: Tool overview

3 Implementation Details

To simplify development, facilitate code re-use and the translation process from XIVL to C++ we have implemented the PCSS (Parallel CSS) library, which provides common building blocks for parallel symbolic simulation. The PCSS library is linked with the C++ program during compilation. An overview of our tool is shown in Fig. 2. In the following we will describe our PCSS library, provide more details on the *fork/join* based exploration and briefly sketch the translation process from XIVL to C++.

3.1 PCSS Library

The right hand side of Fig. 2 shows the main components, and their interaction, of the PCSS library. Essentially it consists of the following components: kernel, scheduler, SMT engine, fork engine and some process shared data.

The kernel provides a small set of functions which directly correspond to the XIVL kernel related primitives (e.g. *wait* and *notify*) and allows to simulate the SystemC event-driven simulation semantics. Furthermore, all thread functions of the XIVL model are registered in the kernel. The kernel will allocate a coroutine with every thread function as entry point. Coroutines naturally implement context switches as they allow to jump execution between arbitrarily nested functions while preserving the local data. Our implementation is using the lightweight *boost context* library and in particular the functions *make_fcontext*

```
bool on_branch(const SmtExpr &cond) {
  auto stat = check_branch_status(cond);
  if (stat == BranchStatus::BothFeasible) {
    bool is_child = fork_engine->fork();
    if (is_child)
      pc = smt->bool_and(pc, cond);
    else
      pc = smt->bool_and(pc, smt->bool_not(cond));
    return is_child;
  }
  return stat == BranchStatus::FalseOnly ? false : true;
}
```

Fig. 3: Symbolic branch execution

and *jump_fcontext* to create and switch execution between coroutines, respectively. The scheduler is responsible for selecting the next runnable thread inside the scheduling loop of the kernel. Our coroutine implementation allows to easily switch execution between the scheduling loop and the chosen thread. POR is employed to reduce the number of explored interleavings. The POR dependency relation is statically generated from the XIVL model and encoded into the C++ program during translation. At runtime it is passed to the scheduler during initialization.

The SMT engine provides common functionality required for symbolic execution. It keeps track of the current path condition, handles the *assume* and *assert* functions, and checks the feasibility of symbolic branch conditions. Furthermore, the SMT engine provides SMT types and operations. Essentially this is a lightweight layer around the underlying SMT solver and allows to transparently swap the employed SMT solver.

The fork engine is responsible to split the execution process into two independent processes in case of a non-deterministic choice. This happens when both branch directions are feasible in the SMT engine or multiple thread choices are still available in the scheduler after applying POR. One of the forked processes will continue exploration while the other is suspended until the first terminates. This approach simulates a depth first search (DFS) of the state space. As an optimization, the fork engine allows to run up to N processes in parallel, where N is a command line parameter to the compiled C++ program. Parallelization is very efficient as the processes explore disjoint state spaces independently.

3.2 Fork/Join based State Space Exploration

Executing Symbolic Branches The *on_branch* function in the SMT engine, shown in Fig. 3, accepts a symbolic branch condition and returns a concrete decision, which is then used to control native C++ control flow structures. The *check_branch_status* checks the feasibility of both branch directions by checking the satisfiability of the branch condition and its negation. In case both branch

```
 1 bool ForkEngine::fork() {
 2   int pid = ::fork();
 3   if (pid != 0) {
 4     num_children++;
 5     while (!try_fork(shared_data, N)) {
 6       if (num_children > 0) {
 7         join_any_child();
 8       } else {
 9         usleep(1); // wait for someone else to join child
10       }
11     }
12   } else {
13     num_children = 0;
14   }
15   return pid == 0;
16 }
```

Fig. 4: Implementation of parallelized forking

directions are feasible, the execution will fork (Line 4) into two independent processes and update the path condition (pc) with the normal (Line 6) or negated condition (Line 8), respectively. Please note that the execution is not forked and the path condition is not extended when only a single branch direction is feasible.

Parallelization The forked processes communicate using anonymous shared memory, which is created during initialization in the first process using the *mmap* system call and thus accessible from all forked child processes. The shared memory essentially contains three information: 1) counter variable to ensure that no more than N processes will run in parallel, 2) shutdown flag to gracefully stop the simulation, e.g. when an assertion violation is detected. 3) unnamed semaphore to synchronize access. The semaphore is initialized and modified using the *sem_init*, *sem_post* and *sem_wait* functions. Furthermore, each process locally keeps track of the number of forked child processes (*num_children*). Fig. 4 shows an implementation of the *fork* function. First the *fork* system call is executed. The child process (*pid* is zero) will never block, since executing only one process will not increase the number of active processes. The parent process however will first try to atomically check and increment the shared counter in Line 5. When this fails, i.e. the maximum number N of processes is already working, the parent process will wait until a working processes finishes, by either awaiting one of its own children (Line 7) or until some other process joins its children (Line 9).

3.3 XIVL to C++ Translation

We use the XIVL example from Fig. 1 to illustrate the XIVL to C++ translation process, which basically peforms five steps: 1) Replace native data types (integer and boolean) and operations with SMT types and operations where necessary (here variable x). Variables which are never assigned a symbolic value (here

Table 1: Experiment results, T.O. denotes timeout (limit 750s)

Benchmark	Kratos	ISS	ParCoSS		
			P-1	P-4	P-8
buffer-ws-p5	1.400	65.951	9.086	2.882	1.987
mem-slave-tlm-bug-50	T.O.	3.731	<0.1	<0.1	<0.1
mem-slave-tlm-sym-50	T.O.	3.940	<0.1	<0.1	<0.1
pressure-15	1.281	219.300	17.182	5.312	3.855
pressure-bug-50	444.781	0.897	<0.1	<0.1	<0.1
irqmp-8	-	108.670	32.719	10.815	8.237
irqmp-12	-	T.O.	530.705	178.108	128.257

variables *i* and *sum*) can keep their native type and perform native operations. 2) Instrument control flow code to query the SMT engine for a concrete decision. The branch condition $x > 0$ will be transformed into an SMT expression, e.g. as *smt→bv_gt(x, smt→bv_val(0))*, and wrapped by the *on_branch* function of the *smt_engine*. 3) Generate static POR information for the scheduler. Essentially, a static analysis is employed to detect *read/write* and *notify/wait* dependencies. A flow- and context-insensitive pointer analysis is used to increase the precision. 4) Redirect builtin XIVL functions to the SMT engine (*assume* and *assert*) and kernel instance (e.g. *wait_event* and *notify*). 5) Add a new main function that will initialize the PCSS library and call the main function of the XIVL model. It will initialize the global data of the XIVL model and then enter the scheduling loop of the kernel by calling the *start_simulation* function.

4 Evaluation and Conclusion

We have evaluated our tool on a set of SystemC benchmarks from the literature [3, 7, 15] and the TLM model of the *Interrupt Controller for Multiple Processors* (IRQMP) of the LEON3-based virtual prototype SoCRocket [17]. All experiments have been performed on a Linux system using a 3.5 GHz Intel E3 Quadcore with Hyper-Threading. We used Clang 3.5 for compilation of the C++ programs and Z3 v4.4.1 in the SMT Layer. The time (memory) limit have been set to 750s (6GB), respectively. T.O. denotes that time limit has been exceeded. The results are shown in Table 1. It shows the simulation time in seconds for Kratos [7], *Interpreted Symbolic Simulation* (ISS) [6, 15], and our tool ParCoSS with a single process (P-1) and parallelized with four (P-4) and eight processes (P-8). Comparing P-4 with P-8 allows to observe the effect of Hyper-Threading. The results demonstrate the potential of our tool and show that our parallelization approach can further improve results. As expected, CSS can be considerably faster than ISS. On some benchmarks Kratos is faster due to its abstraction technique, but the difference is not significant. Furthermore, Kratos is not applicable to the *irqmp* benchmark due to missing C++ language features. For future work we plan to integrate dynamic information, for POR and selection of code blocks for native execution, into our CSS framework.

References

1. www.systemc-verification.org/ParCoSS.
2. Accellera Systems Initiative. SystemC, 2012. Available at http://www.systemc.org.
3. N. Blanc and D. Kroening. Race analysis for SystemC using model checking. *ACM Trans. Des. Autom. Electron. Syst.*, 15(3):21:1–21:32, June 2010.
4. C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.
5. C.-N. Chou, C.-K. Chu, and C.-Y. R. Huang. Conquering the scheduling alternative explosion problem of SystemC symbolic simulation. In *ICCAD*, pages 685–690, 2013.
6. C.-N. Chou, Y.-S. Ho, C. Hsieh, and C.-Y. Huang. Symbolic model checking on SystemC designs. In *DAC*, pages 327–333, 2012.
7. A. Cimatti, I. Narasamdya, and M. Roveri. Software model checking SystemC. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 32(5):774–787, 2013.
8. C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *POPL*, pages 110–121, 2005.
9. P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem.* Springer, 1996.
10. D. Große and R. Drechsler. *Quality-Driven SystemC Design.* Springer, 2010.
11. V. Herdt, H. M. Le, and R. Drechsler. Verifying SystemC using stateful symbolic simulation. In *DAC*, pages 49:1–49:6, 2015.
12. V. Herdt, H. M. Le, D. Große, and R. Drechsler. Compiled symbolic simulation for SystemC. In *ICCAD*, 2016.
13. IEEE. *IEEE Standard SystemC Language Reference Manual.* IEEE Std. 1666, 2011.
14. J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.
15. H. M. Le, D. Große, V. Herdt, and R. Drechsler. Verifying SystemC using an intermediate verification language and symbolic simulation. In *DAC*, pages 116:1–116:6, 2013.
16. H. M. Le, V. Herdt, D. Große, and R. Drechsler. Towards formal verification of real-world SystemC TLM peripheral models - A case study. In *DATE*, 2016.
17. T. Schuster, R. Meyer, R. Buchty, L. Fossati, and M. Berekovic. SoCRocket - A virtual platform for the European Space Agency's SoC development. In *ReCoSoC*, pages 1–7, 2014. Available at http://github.com/socrocket.