Proceedings of the International MultiConference of Engineers and Computer Scientists 2011 Vol II,
IMECS 2011, March 16 - 18, 2011, Hong Kong

Efficient Encoding of SystemC/TLM in Promela

Kevin Marquet, Bertrand Jeannet, and Matthieu Moy

***Abstract*—To deal with the ever growing complexity of Systems-on-Chip, designers use models early in the design flow. SystemC is a commonly used tool to write such models. In order to verify these models, one thriving approach is to encode its semantics into a formal language, and then to verify it with verification tools. Various encodings of SystemC into formal languages have already been proposed, with different performance implications. In this paper, we investigate a new, automatic, asynchronous means to formalize models. Our encoding supports the subset of the concurrency and communication constructs offered by SystemC used for high-level modeling. We increase the confidence in the fact that encoded programs have the same semantics as the original one by model-checking a set of properties. We give experimental results on our formalization and compare with previous works.**

I. Introduction

As the complexity of embedded systems grows, the need for new methods has appeared for the co-design of hardware and software. Indeed, low-level hardware description languages such as VHDL and Verilog simulate slowly, can hardly be used to design complex systems and therefore make early software development difficult. Consequently, higher-level modeling tools have appeared, allowing hardware and software descriptions.

Transaction-Level Modeling [4] (TLM) is an approach in which the architecture and the behavior of a System-on-Chip (SoC) are described in an executable model, but the micro-architecture details and precise timing behavior are abstracted away. SystemC [20] has become the *de facto* standard for TLM modeling. It contains a simulation kernel that can execute concurrent processes communicating through channels and shared variables, using C++ libraries. In this paper, we are interested in TLM programs, written in SystemC. We focus on the subset of SystemC needed for TLM modeling, leaving apart the constructs originally introduced in SystemC to write lower-level programs (like RTL).

SystemC descriptions are C++ concurrent programs that can be tested and/or verified in order to detect design flaws. Verifying a concurrent program can be done with various approaches. One thriving approach is to describe its semantics formally, and then to verify this semantics using verification tools. The first step is called *model extraction* and leads to the translation of the program into a formal representation, and the second step is the verification performed on the formal representation. Different representations can be chosen, that model differently time and concurrency, and that are connected to different verification tools.

Manuscript received December 1, 2010; revised January 15, 2011.

Kevin Marquet is with VERIMAG, Université Joseph Fourier, Grenoble, France. Kevin.Marquet@imag.fr

Bertrand Jeannet is with the INRIA Rhônes-Alpes, Grenoble, France. Bertrand.Jeannet@inrialpes.fr

Matthieu Moy is with VERIMAG, Grenoble INP, Grenoble, France. Matthieu.Moy@imag.fr

```
SC_MODULE(mytop) {
    sc_event e;
    SC_CTOR(mytop) {
       SC_THREAD(myFctP); SC_THREAD(myFctQ);
    }
    void myFctP() {...; wait(e); ... }
    void myFctQ() {...; e.notify(); ... }
}
```

Fig. 1. A basic SystemC module

This paper focuses on the issue of *model extraction*, in the context of the verification of SoC modeled as SystemC concurrent programs. Our contributions are as follows:

1) We present **new encoding principles** in section IV for the extraction of formal representations from SystemC programs, and in particular for modeling the semantics of SystemC scheduler. We argue that this encoding is simple and elegant. Its main goal is however to favor the efficiency of verification tools. This extraction is performed in a fully **automatic** way by our verification chain.[1]
2) In order to **validate their correctness**, we define properties that must hold for an encoding to be valid. These properties and how they are tested are detailed in section V.
3) At last, section VI presents **experimental results** on SystemC examples translated to *Promela*, the asynchronous formalism used as input to the SPIN model-checker. Our results show major improvements over past similar works, thanks to the fact that our encoding does not introduce complex behaviors limiting the applicability of formal verification tools. We show in particular a tremendout reduction of the number of states that SPIN needs to explore.

Before presenting these, we present SystemC in section II and compare our approach to related works in section III.

II. SystemC

We give a very partial overview of SystemC, focusing on the points that are relevant for this paper.

A SystemC program defines an *architecture*, i.e. a set of components and connections between them, and a *behavior*, i.e. components have a behavior defined by one or several processes and communicate with each other through ports. Once the architecture is defined (by the *elaboration phase* performed at the beginning of execution), the *simulation phase* starts: processes execute according to the SystemC scheduling policy. As an example, figure 1 shows a SystemC module containing two processes, one waiting for an event, the other notifying it.

We do not consider here the notion of δ-cycles [20], inspired from traditional HDL languages, since it is not useful

[1]The implementation is open-source and available from http://gitorious.org/pinavm.

ISBN: 978-988-19251-2-1
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

for TLM models (this implies that we do not support SystemC constructs like wait(SC_ZERO_TIME), which makes a process wait until the next evaluation phase, or components sc_signal and sc_fifo). We focus on the following constructs of SystemC, which are the basis for TLM modeling:

wait(d: int) Stops executing the current process, yields back the control to the scheduler and makes the current process to wait for the given duration.

wait(e: event) Stops executing the current process, yields back the control to the scheduler and makes the current process to wait for the event to occur. SystemC also allows the constructs **wait(e1 & e2)** and **wait(e1 | e2)** to wait for conjunctions and disjunctions of events.

event.notify() Makes processes waiting for the specified event eligible (without stopping the current process).

event.notify(delay: int) Triggers a notification after the given delay. In SystemC, only the earliest timed notification is kept, which simplifies the semantics of this primitive.

SystemC scheduling follows a *non-preemptive* scheduling policy. When several processes are eligible at the same time, the scheduler runs them in an unspecified order.

Concerning communications between process, we use shared variables to model several threads belonging to the same module communicating by accesses to the fields of the module. Concerning TLM ports, our implementation does not (yet) manage them explicitly; it requires the function calls to be done directly from modules to modules instead of relying on port/socket bindings [21], which is a (useful) syntaxical sugar. We therefore focus on the notion of method calls.

Restricting ourselves to a strict subset of SystemC is not a limitation as far as we are focused on TLM models. Of course it implies that we cannot handle more general SystemC programs, but it also makes our approach more general in the sense that it could easily be adapted to other discrete-event cooperative simulator (like the cooperative version of jTLM [2]).

III. Overview of the Problem and Related Works

General overview: The challenge raised by formal verification of SystemC models is that SystemC has not been designed for this purpose. An option could be to consider them as regular C++ programs, but few verification tools are available for them, especially when the goal is to check *functional properties*. Moreover, a general verifier would have to analyze the SystemC class library and to rediscover by itself its high-level semantics. For these reasons, most related work proceeds differently: the user's code is *translated and abstracted* into the formal model accepted by the targeted verification tool, whereas the high-level semantics of SystemC/TLM class libraries is *hand-coded* in the formal model. The verification tool is then applied to the resulting model.

Representation of the SystemC scheduler: Modeling the semantics of the SystemC library reduces mainly to modeling the SystemC scheduler. Three options can be imagined to represent the scheduler in a formal representation: (1) model *the deterministic behavior* of the reference implementation described in the SystemC standard [20]; or (2) model a *more general non-deterministic scheduler*, either (2a) as an explicit additional process, or (2b) by incorporating it in the semantics of the synchronization instructions (typically the ones described above). Choosing arbitrarily a specific, deterministic scheduler allows only to explore a subset of the behaviors. We do not want such restriction and therefore do not consider solution 1.

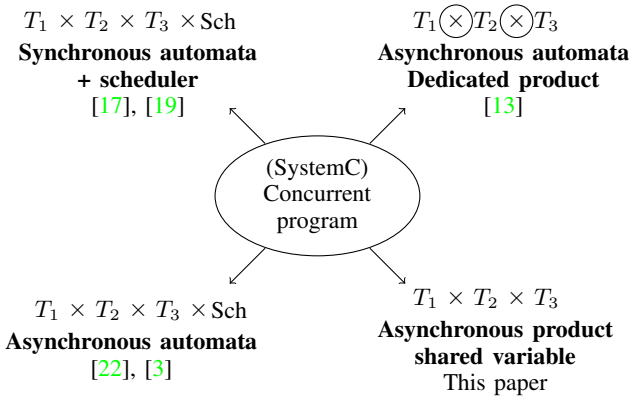

Fig. 2. Different approaches for translating SystemC programs into other formalisms

Solution 2a is interesting as it does not restrict the set of possible behaviors. This is the solution considered in [17]. However, encoding the scheduler as a special process interacting with the SystemC processes complexifies the behavior of the global system. Typically, such an encoding induces additional communications between processes, compared to the original SystemC semantics. For instance, the encoding of the **event.notify()** primitive is likely to induce a context-switch (as it changes the state of the scheduler), which does not occur in the original SystemC semantics. The bad consequence is that such additional communications may prevent verification tools to perform powerful optimizations. Typically, partial-order reduction relies on a notion of "independent transitions", and cannot be applied if the notion of "transition" of the model does not correspond to the notion of atomic sections in SystemC.

Consequently, we have chosen the approach of point 2b: we do not encode the scheduler as an explicit process composed in *parallel* with the SystemC processes. Instead, we integrate the scheduler in the semantics of the synchronization primitives that are used *sequentially* inside each SystemC process, without introducing any "artificial" context-switches.

Related work: The related work based on encoding of SystemC programs in other formalisms we are aware of (see Fig. 2) are all based on solution 2a, but they can be further classified according to the considered formal model, which may be synchronous or asynchronous.

LusSy [17] is a prototype of a complete verification chain. It encodes the processes *and* the scheduler in synchronous automata. The intermediate formalism is called *HPIOM*. The main drawback of this formalism is that it breaks down relevant information into lower-level ones, making the task harder for verification tools, that are unable to handle real case studies. A similar work [7] describes how to generate UPPAAL models from SystemC programs. Several other translation-based approaches have been proposed [19], [10], also introducing a lot of complexity in the encoding.

ISBN: 978-988-19251-2-1
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

Other works considers asynchronous formalisms. We actually show in section IV-C that SystemC's time semantics is encoded naturally and efficiently with deadline variables (similar to "clocks") evolving asynchronously, unlike the semantics of timed automata used in UPPAAL, in which clocks evolves synchronously.

In [13], a SystemC process is encoded with a *MicMac* automaton which distinguishes *micro-states* and *macro-states*. *Micro-states* represent points where the process can not yield, contrarily to *macro-states* that are yielding points (typically following a `wait()`). MicMac automata can be composed in parallel using dedicated product exploiting the notion of micro-states. This approach cannot be used directly in existing verification tools that are not aware of micro-states. [22] proposes first to encode a SystemC programs into MicMac automata *and then* to encode MicMac automata into Promela. However, the last translation loses the specific benefits of MicMac formalism. Moreover, we show that some SystemC notions are encoded naturally in Promela (in particular, atomic sections of SystemC correspond directly to the `atomic` statement in Promela), while using MicMac as an intermediate formalism prevents such direct translation and introduces unnecessary complexity in the encoding. To sum up, the approach implies the re-encoding in an explicit and asynchronous way of some mechanisms that verification tools, including SPIN, can tackle very efficiently *when the corresponding native mechanisms are used*.

Our approach: asynchronous formalism + shared variables: This paper proposes a solution based on an asynchronous model (namely Promela) to encode TLM concurrent programs, that consists in modeling the asynchronous communications and the semantics of the scheduler by inserting synchronization primitives manipulating shared variables into the code of the processes. The expected gain of this approach is to minimize the interactions between processes, so as to let verification tools freely apply reduction techniques such as symmetry or partial order reductions.

Other Validation Approaches: Alternatives to formal verification are based on code execution, for instance standard testing, run-time verification [6] or explicit model-checking [5]. In [5] the original C++ code is instrumented so as to enable an on-the-fly state-space exploration of the model, based on the techniques of the CADP [1] toolbox to execute native code. These methods showed to be very efficient to explore the possible schedulings of a system, but are fundamentally limited to explicit-state exploration, and cannot be extended to perform symbolic model-checking or abstract interpretation. A hybrid approach is presented in [3], which executes C++ code natively for `SC_METHOD`s, but relies on translation for `SC_THREAD`s. This work is probably the closest to the one presented in this paper, as the encoding does not rely on a separate process for the scheduler.

IV. Translation from C++ and Encoding of SystemC Scheduler

We first remind the general principles of our tool chain for SystemC, then we describe precisely the encoding of SystemC synchronization primitives, and last we discuss some alternatives. Among the primitives mentioned in section II, we will not consider delayed notifications, or waiting for conjunctions or disjunctions of events, but discuss in section IV-C how to extend our encoding to handle such constructs.

A. Translating User Processes from C++ with PinaVM

Translating SystemC automatically requires the use of a complete SystemC front-end. Borrowing some ideas from Pinapa [16], we set up a SystemC front-end called PinaVM [15] able to take as input a SystemC program and to produce an intermediate representation. This front-end is based on the compiler infrastructure LLVM [12] and the intermediate representation is mainly composed of basic blocks containing SSA (*Static Single Assignment*) instructions. PinaVM executes the elaboration phase like Pinapa, and uses a *Just-In-Time* compiler to retrieve SystemC information on events or ports to enrich intermediate representation obtained from LLVM.

From the intermediate representation produced by our front-end, a back-end produces automatically a Promela program. Each SSA instruction is translated into an equivalent in Promela instruction. Although Promela provides some of the structuring mechanisms of a call definition, these mechanisms provide no benefit for the verification engine compared to a static inlining, therefore, we chose to inline directly all function calls.

In this translation, each SystemC thread generates a Promela process. We do not consider in this paper dynamic creation of processes, that are seldom encountered in SoC models.

B. Encoding synchronization primitives

In the encoding of SystemC synchronization primitives, we rely on three features related to concurrency that are provided by Promela:

1) The ability to use shared variables.
2) The blocked($cond$) primitive, which stops the execution of the current process until condition $cond$ on shared variables becomes true, and gives the control to another process (the actual syntax in Promela is simply `[`$cond$`]`).
3) The notion of atomic section, that can be interrupted with the blocked primitive.

In the sequel we denote by E^k the event k, with $1 \leq k \leq N_e$ and the set of N_p processes is denoted P.

Events: SystemC events are *non persistent*: the instruction wait(E^k) is blocking, and takes into account only notifications taking place after its execution: if the event E^k is notified before the execution of a **wait**(E^k) instruction, it will be ignored by this instruction. An important consequence is that a process can be waiting for at most one event (we currently do not consider the construct `wait(e1 & e2)` of SystemC).

For encoding events, we thus associate to each process p a bounded integer $0 \leq W_p \leq N_e$ such that:

- $W_p = k$ when process p waits for E^k;
- $W_p = 0$ when process p is not waiting for an event and is eligible;

and we define the wait and notify instructions in Tab. I. We need for this encoding $N_p \log_2(1 + N_e)$ bits.

ISBN: 978-988-19251-2-1
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

TABLE I
ENCODING EVENTS ALONE

p::**wait**(E^k):	p::E^k.**notify**():
1 $W_p := k$	3 $\forall i \in P \mid W_i == K$
2 blocked($W_p == 0$)	4 $W_i := 0$

TABLE II
ENCODING TIME ALONE

p::**wait**(d):
1 $T_p := T_p + d$
2 blocked($T_p == \min_{i \in P}(T_i)$)

Time: SystemC time management internally assumes a discrete time semantics, although in the API timed functions use floating-point durations. We thus assume that we have a specific construct wait(d:int) to wait for the *discrete* duration d to elapse.

For encoding time, we attach an internal *deadline variable* $T_p : int$ to each process p. It represents the next deadline for p when p is waiting, and the current date when p is running. It is not necessary to examine the state of the process p for each value of T_p, we only need to respect the schedulings allowed by the durations waited for by the processes. Consequently, we define the encoding `wait(d)` in Tab. II:

- T_p is incremented with d;
- p becomes eligible if its deadline variable is the minimum of all deadline variables.

Alternatively, we could maintain a global clock T_g to $\min_{i \in P}(T_i)$ and replace the blocking condition by **blocked**($T_p == T_g$). The advantages and drawbacks of this option w.r.t. the efficiency of the verification process is hard to assess *a priori*.

Interaction between time and events: Events and time interact together, and things become subtle when some processes are waiting for events and others for a time duration. We propose the encoding given on table III, based on the following principles:

(1) The value of a deadline variable T_p is meaningful *only if* $W = 0$ (process p is not waiting for an event). When a process is waiting for an event, T_p is not updated. The main invariant becomes thus: *"the deadline variable of a running or eligible process is the minimum of the deadline variables of processes not waiting for an event."*
(2) Concerning the **wait(d)** instruction, the blocked process becomes eligible as soon as its deadline variable is the minimum of deadline variables *of processes not waiting for an event*, according to principle 1).
(3) When process p notifies an event E^k, not only should the variables W_i be reset (for processes i waiting for E^k), but also should their deadline variable be updated to the current date (which is equal to the deadline variable T_p of the running process p). This is because of principle (1): these deadline variables becomes meaningful again, and the invariant above should be maintained. This is important to make a sequence **wait**(E^k)**; wait(d)** behave correctly in a process p.

Fig. 3 depicts the Promela code corresponding to the pseudo-code of Tab. III.

TABLE III
ENCODING EVENTS AND TIME

p::**wait(d)**:
1 $T_p := T_p + d$
2 blocked($T_p == \min_{\substack{i \in P \\ W_i == 0}}(T_i)$)

p::**wait**(E^k):	p::E^k.**notify**():
3 W_i := K	5 $\forall i \in P \mid W_i == k$
4 blocked($W_i == 0$)	6 $W_i := 0$
	7 $T_i := T_p$

```
int e[NBTHREADS];
int T[NBTHREADS];
bool end[NBTHREADS];

inline init_coding(i) {
  i = 0;
  do :: i == NBTHREADS -> break;
     :: else ->
        e[i] = 0; T[i] = 0; end[i] = false;
        i++; od;
}

inline notify(pid, nevent, i) {
  i = 0;
  do :: i < NBTHREADS && e[i] == nevent ->
        e[i]=0;  T[i]=T[pid];  i++;
     :: i < NBTHREADS && e[i] != nevent ->
        i++;
     :: i == NBTHREADS -> break; od;
  i = 0;
}

inline wait(pid, time) {
  T[pid] = T[pid] + time;
  ((end[0]) || (e[0] != 0) || (T[pid] <= T[0]) &&
   (end[1]) || (e[1] != 0) || (T[pid] <= T[1]) &&
   (end[2]) || (e[2] != 0) || (T[pid] <= T[2]));
}

inline wait_e(pid, nevent) {
  e[pid] = nevent;
  e[pid] == 0;
}
```

Fig. 3. Encoding in Promela. Compared to Tab. III, we add the `end` array to handle the particular case where a task is completed in the `wait(d:int)` instruction.

C. Discussion and Improvements

Our encoding implements in some way an asynchronous time semantics, as opposed as the synchronous time semantics of timed automata used in tools like UPPAAL [11], in which clocks evolves synchronously. Our approach thus does not enable the use of these tools. Notice however that we hardcode in our approach the fact that we only need to know the next deadlines, and not all the possible intermediate values that a discrete synchronous clock would take between the current time and the next deadline. As a result, multiplying all the durations by a constant factor does not impact the size of the reachable state-space with our encoding.

Finite-state model-checkers like SPIN [8] do not support unbounded deadline variables. However, it is easy to modify our encoding by exploiting the fact that two global states agreeing on the differences $T_i - T_j$ between deadline variables are equivalent w.r.t. the synchronization primitives of Tab. III. In the resulting *relative time* encoding, the invariant: "the minimum of the deadline variables of processes not waiting for an event is zero" is ensured by shifting accordingly those deadline variables in **wait(d)** instructions.

Implementing delayed notification on a single event could be done with the principles we followed in this section. This would require to add another deadline variable in each

ISBN: 978-988-19251-2-1
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

process. Implementing waiting for conjunction or disjunction of events would require the following modifications:

- The bounded integer variables $0 \leq W_p \leq N_e$ should be replaced by N_e Boolean variables $W_{p,k}$ with $1 \leq k \leq N_e$ denoting the event E^k, because a process p can know wait for a set of events.
- We should also add a Boolean variable per process to distinguish whether the process is waiting for a conjunction or a disjunction of events.

To sum up, our approach can easily model such constructs, at the cost of additional finite-state variables.

V. Validating the encoding principles

The encoding of SystemC primitives defined above may seem intuitively correct, but experience shows that concurrent systems are often faulty !

The ideal solution would be to prove that our encoding is correct for any program using it. Such a quantification on programs requires the use a proof-assistant, which is a very demanding task. This would require to give a formal semantics to SystemC (which implies C++) and to Promela, and to prove that the two programs are equivalent.

The approach we have chosen is to construct a set of properties and to verify them on instances of the translation, in order to get confidence in the correctness of the encoding, just like certifying compilers [18] verify the result of each compilation. Those verifications were actually very useful, allowing us to detect bugs in several preliminary versions of our encoding.

We considered three invariants (see [14]). (i) the invariant stated in section IV-B; (ii) "If process i notifies event E^k for which process j is waiting, then $T_i \geq T_j$"; (iii) "When a process p waiting for an event is made eligible by a notifying process (line (7) of Fig. III), the deadline T_p does not change until its election as the running process." These can be easily translated to a relative time setting discussed in section IV-C.

Two techniques were used to verify them with SPIN: direct assertions in the code, or a "monitoring" process for properties not related to a specific line number. This process only contains assertions, which can be detected as violated in the automata product performed by SPIN. As the examples we considered are deadlock-free, we also verified that the encoding does not introduce deadlocks (for instance, by scheduling processes in the wrong order).

The examples on which we checked these properties are the following. First, we experimented on an adaptation of the reader/writer problem in which two writers and one reader access a FIFO. Second, we considered a model of a communication between a Memory, a DMA, a bus and a CPU. Third, we considered the example used in a previous translation from SystemC to SPIN [22], described in the appendices of [14].

VI. Experiments and efficiency of our encoding

The aim of the previous section was to check that our encoding actually reflects SystemC semantics. However, our motivation for the encoding we propose is to enable better performances of model-checkers, compared to other encoding approaches described in section III. We now compare experimentally the efficiency of our encoding w.r.t. model-checking with the encoding proposed in [22] applied to the same example.

A. A SystemC example

Our test model is the one used in [22] and detailed in [14]. It consists of a chain of modules. The first module triggers an interrupt in the next one. This interrupt notifies an event, allowing the module to trigger an interrupt in the next module, and so on. The last module contains an assertion which is either always false (bug) or always true (no-bug). The latter forces SPIN to compute the whole state space when checking for invalid assertions. While this program may seem artificial, it exhibits the characteristics found in more complex real-world models and leading to state explosion: many processes, synchronized by SystemC events, which can thus be lost depending on the execution order of the various statements. Such study allows to experiment on how the state space that needs to be explored grows depending on parameters. As this test model is untimed, we test here only the efficiency of the encoding of events.

B. Results

The results presented in Fig. 4 focuses on the main parameter which is the number of modules. It shows the number of states computed by SPIN during the model-checking of the example presented above.

Those results show a reduction by a factor of about 10 compared to previous results presented in [22]. The comparison between the two approaches, in the case where there is no bug is shown in figure 4. We can see that, with our encoding, SPIN is able to model check up to 21 processes, compared to 15 in the other approach.

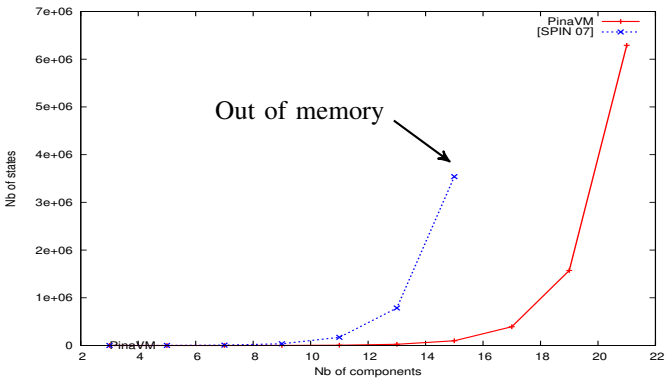

Fig. 4. Experimental results of the two approaches

VII. Conclusion

We investigated the formalization of models of SoC in the form of asynchronous automata. We proposed an encoding of synchronization primitives related to events and time using shared variables and sequential instrumentation of processes. This choice contrasts with other approaches in which parallel instrumentation is used, under the form of an additional process modeling the SystemC scheduler added to the system. We ensured that the encoding principles are correct by verifying a number of invariants. The given principles are general and are applicable to different back-end languages.

We experimented on the SPIN model-checker, showing on a typical example that our encoding leads SPIN to explore

ISBN: 978-988-19251-2-1
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

ten times less states during model-checking of the encoded model, compared to an encoding based on parallel instrumentation. This confirms the conjecture we express in section III. In addition, the translation has been fully automated: our tool reads SystemC code directly, and generates Promela code without human intervention. Our results are thus due to our encoding and not to some specific optimizations. The tool can be downloaded freely from http://gitorious.org/pinavm.

Besides experimenting with a wider set of cases studies, we see at least two point to investigate in the future. First we have yet to compare our time management to other approaches. We intend to compare this solution to approaches based on timed automata and relying on the UPPAAL [7] tool for model-checking to validate our discussion of section IV-C on the asynchronous encoding of time in SystemC. A second perspective to evaluate the relevance and the efficiency of static analysis tools such as CONCURINTERPROC [9] for checking safety properties of timed SystemC models.

REFERENCES

[1] J. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP a protocol validation and verification toolbox. In *Computer Aided Verification*, pages 437–440. Springer, 1996.

[2] Giovanni Funchal and Matthieu Moy. jTLM: an experimentation framework for the simulation of transaction-level models of systems-on-chip. In *DATE*, 2011. (to appear).

[3] Hubert Garavel, Claude Helmstetter, Olivier Ponsini, and Wendelin Serwe. Verification of an Industrial SystemC/TLM Model using LOTOS and CADP. In *7th ACM-IEEE International Conference on Formal Methods and Models for Codesign MEMOCODE'2009*, Cambridge, MA United States, 2009.

[4] Frank Ghenassia. *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[5] Claude Helmstetter. TLM.open: a SystemC/TLM Front-end for the CADP Verification Toolbox. Extended abstract for SBDCES workshop (http://unit.aist.go.jp/cvs/workshop/SBDCES.html) Work financed by the Multival project.

[6] Claude Helmstetter, Florence Maraninchi, and Laurent Maillet Contoz. Full simulation coverage for SystemC transaction-level models of systems-on-a-chip. *Formal Methods in System Design*, 35(Number 2 / October, 2009):pages 152–189, 06 2009.

[7] Paula Herber, Joachim Fellmuth, and Sabine Glesner. Model checking SystemC designs using timed automata. In *CODES/ISSS '08: Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, pages 131–136, New York, NY, USA, 2008.

[8] Gerard J. Holzmann. *Design and validation of computer protocols*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[9] B. Jeannet. Relational interprocedural verification of concurrent programs. In *Software Engineering and Formal Methods, SEFM'09*. IEEE, November 2009. to appear.

[10] D. Karlsson, P. Eles, and Z. Peng. Formal verification of systemc designs using a petri-net based representation. In *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, page 1233. European Design and Automation Association, 2006.

[11] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[12] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 75, Washington, DC, USA, 2004. IEEE Computer Society.

[13] F. Maraninchi, M. Moy, J. Cornet, L. Maillet-Contoz, C. Helmstetter, and C. Traulsen. SystemC/TLM semantics for heterogeneous system-on-chip validation. In *NEWCAS-TAISA 2008: Proceedings of the Joint 6th International IEEE Northeast Workshop on Circuits and Systems and TAISA Conference*, pages 281–284, 2008.

[14] Kevin Marquet, Bertrand Jeannet, and Matthieu Moy. Efficient encoding of SystemC/TLM in Promela—full version. Technical Report TR-2010-7, Verimag Research Report, 2010.

[15] Kevin Marquet and Matthieu Moy. PinaVM: a SystemC front-end based on an executable intermediate representation. In *International Conference on Embedded Software International Conference on Embedded Software*, page 79, Scottsdale, USA, 10 2010. SD B.4.4, I.6.4, D.2.4 OpenTLM (projet Minalogic).

[16] Matthieu Moy, Florence Maraninchi, and Laurent Maillet-Contoz. Pinapa: An extraction tool for SystemC descriptions of systems-on-a-chip. In *EMSOFT*, September 2005.

[17] 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*, 2006. special issue on SystemC-based systems.

[18] G.C. Necula and P. Lee. The design and implementation of a certifying compiler. *ACM SIGPLAN Notices*, 33(5):333–344, 1998.

[19] B. Niemann, C. Haubelt, M. Oyanguren, and J. Teich. Formalizing TLM with communicating state machines. *Advances in Design and Specification Languages for Embedded Systems*, pages 225–242, 2007.

[20] Open SystemC Initiative. *IEEE 1666 Standard: SystemC Language Reference Manual*, 2005. http://www.systemc.org/.

[21] Open SystemC Initiative (OSCI). *OSCI TLM-2.0 Language Reference Manual*, July 2009. Version JA32, available from http://www.systemc.org/downloads/standards.

[22] Claus Traulsen, Jérôme Cornet, Matthieu Moy, and Florence Maraninchi. A SystemC/TLM semantics in Promela and its possible applications. In *14th Workshop on Model Checking Software SPIN*, July 2007.

ISBN: 978-988-19251-2-1
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)