DLINE JOURNALS

Journal of Information Organization

Print ISSN: 2278 - 6503
Online ISSN: 2278 - 6511

JIO 2024; 14 (1)
https://doi.org/10.6025/jio/2024/14/1/1-18

Formal Verification of Systemc Using Error-Free Translation

Claude Helmstetter
Verimag – CNRS
2 Avenue de Vignate, 38610 Gières, France
claude.helmstetter@gmail.com

Received: 18 September 2023
Revised: 29 November 2023
Accepted: 10 December 2023

ABSTRACT

SystemC/TLMs are C++ programs that simulate embedded software before the hardware low-level description is available and are used as gold-plated models for hardware verification. Verification of SystemC/TLMs is important because a mistake in the model can deceive the system designers or reveal a defect in the specifications. There is an open-source simulator, but there are no formal verification tools. To apply model checking to System-C/TML models, you must provide semantics for standard C++ code and specific System-C/TML features. The usual approach is translating the SystemC/TML code into a formal language with a model checker. However, we suggest a different approach that eliminates the risk of error-prone translation. In the case of a system-C/TML program, transitions are obtained by running the original code using G++ and an extended systemC library. We ask the user to provide additional functions to store the current model state.These extra functions are typically less than 20% larger than the original model and allow applying all CADP verification capabilities to the SystemC/TLM model.

Keywords: *Systemc/TML Code, Formal Verification Procedures, Model Checkers*

1. Introduction

The design of abstract models written in SystemC/TLM has become more common in the development of embedded systems. These models allow the simulation of the embedded software before the hardware RTL description is available, and are used as golden models for hardware verification. The verification of the SystemC/TLM models is an important issue since an error in the model can mislead the system designers or reveals an error in the specifications.

ASI (Accellera Systems Initiative, previously OSCI: Open SystemC Initiative) provides an open-source simulator for SystemC/TLM and a library SCV (SystemC Verification) to ease test generation. However, ASI does not provide tools for formal verification. Moreover, while the SystemC specification allows many schedules for a given test case, the ASI simulator always exhibits the same schedule. Thus, even if an execution leads to the expected result, another execution with

a different schedule may be erroneous. To find these kind of bugs, many publications have experimented with the use of model checking. The problem of verifying C++ and SystemC codes could be avoided by writing transactional models in a formal language directly. However, in order to model embedded systems at the transaction level, engineers of industrial companies prefer to use SystemC/TLM. One reason is that SystemC/TLM provides all the useful features directly, like shared memory and transactional communication channels. Another reason is that a SystemC/TLM program is mainly C++ code, so engineers can learn SystemC/TLM quickly, and existing C code can be reused easily.

In order to apply model checking to a SystemC/TLM program, the usual approach relies on the translation of the SystemC/TLM code into a formal language for which a model checker is available. A lot of languages and tools have been tested so far (see Subsection 3.1.2). Nonetheless, there have been few successes with industrial case studies.

We propose another approach that suppresses the translation effort. Basically, an explicit model checker must be able to execute transitions and store states. Given a SystemC/TLM program, we assume that the states are the places where processes yield back to the scheduler. Consequently, transitions correspond to pieces of C++ code delimited by yield points: either wait statements or return statements from the process main function. We obtain the transitions by executing the original code using g++ and a SystemC library, as in any simulator. Storing the state could be done by copying the whole memory used by the simulator, but would be inefficient. Therefore, we ask the user to provide additional functions to store the current state and restore a previous state. Part of the state, including the SystemC kernel, is stored automatically; so in general the user can only store the SystemC module data members.

Following this approach, we have developed a new front-end for the CADP tool suite. The CADP tool suite includes many tools useful for formal verification and bug finding; the main tool is an explicit model checker. This article does not introduce a new verification technique (we did not change anything in CADP) except a pragmatic and efficient way to use existing tools to verify programs written in a language that has not been designed to ease formal verification. The new front-end we have developed is not fully automatic since the user must provide some additional functions; these additional functions generally represent less than 20% of the size of the original model.

The model checking technique is known to be limited by the state space explosion. Because we rely on this technique and there are no changes in the core algorithm, we are limited in this area. Nevertheless, model checking has been applied to many real-life case studies (over 150 using CADP in many application fields[1] most of the times, using model checking allowed to verify properties or discover bugs. We have written our new front-end in a way that avoids to make the state space explosion even worse by adding intermediate states and transitions, which was the case using a previous approach [16]. Experiments which were first made with benchmarks, then with a single SystemC module, and finally with a basic system, show that we can indeed find bugs and prove some properties on real-life TLM models.

The remainder of this article is organized as follows. We present briefly SystemC and TLM in Section 2. Section 3 gives an overview of the related work and presents the existing CADP toolbox. Section 4 describes our technique to connect SystemC/TLM with CADP. The performances of TLM.open are evaluated in Section 5 and Section 6 concludes this article.

2. SystemC and TLM

SystemC [1] is a C++ library published by the Accellera Systems Initiative (ASI) and defined by an IEEE standard which provides classes to describe heterogeneous systems composed of hardware and software. The architecture of a system is defined by a set of modules connected by synchronous or asynchronous ports and channels (sc_module, sc_port, ...). Each module contains zero, one, or various processes (SC_THREAD or SC_METHOD) describing the system's behavior. SystemC processes interact using shared memory or communication channels and are synchronized using SystemC events (sc_event e, e.notify(), wait (e)) with timing annotations (sc_time t, wait (t)).

[1] *Case studies achieved using the CADP toolset: http://www.inrialpes.fr/vasy/cadp/case-studies.*

Each SystemC process is a C++ method that is executed by the SystemC scheduler communicates with other processes using shared memory and may explicitly suspend itself by executing a wait statement. When the process is resumed by the scheduler, its execution continues from the wait statement. Each SystemC process is eligible or running or waiting for a SystemC event. There is, at most, one running process simultaneously. If the running process notifies an event, then all processes waiting for this event move from waiting to eligible.

The Transaction Level Modeling (TLM) library [10] built upon SystemC, provides a transaction mechanism that encapsulates communication protocols (data transfer and synchronization) between modules and accelerates both model design and simulation. Using a transaction, a process in an initiator module can directly call the methods exported by a target module. Thus, a process can read many values from a memory, or set many registers of a peripheral without any costly inter-process synchronization (no context switch is required). At the TLM level of abstraction, processes inside the same module communicate using SystemC events and shared variables. A TLM model can be timed or untimed: a timed model contains timing annotations (sc_time t, wait(t)) whereas an untimed model does not. An untimed model includes more possible behaviors than a timed model, increasing the coverage, but also the cost, of the verification.

Because SystemC and TLM are C++ libraries, simulating a SystemC/TLM model does not require a dedicated SystemC/TLM parser. A SystemC/TLM model is parsed and compiled as with any C++ program, using a regular C++ compiler, such as g++.

3. Related Works

3.1. Verification of SystemC/TLM Models

In order to provide formal verification for SystemC/TLM programs, two approaches were investigated: stateless model checking of a SystemC/TLM program and a translation of a SystemC/TLM program into a language for which a state ful model checker is available.

3.1.1. Stateless Model-Checking

A stateless model-checker explores the set of all the possible executions of a given program without storing the states. Because the states are not stored, a stateless model-checker can execute the same transition many times. Moreover, if the program under verification has at least one possible execution that does not terminate then the stateless model-checker will not terminate either. However, stateless model-checkers have benefits: 1. naive stateless model-checkers are easy to implement because one just needs to modify the functions used for non-deterministic choices; 2. their memory consumption is limited (linear in terms of execution lengths).

Many stateless model checking tools have been implemented for SystemC/TLM programs [15, 21, 2]. In order to reduce the number of executions explored, these model-checkers select a subset for the possible executions; this subset is guaranteed to detect all the errors of a particular family; such as all the assertion failures or all the deadlocks. All these stateless model-checkers implement dynamic partial order reduction [5]; the selection of the executions explored is based on the analysis of detailed execution traces. The dynamic partial order algorithm was specifically adapted for the particularities of the SystemC scheduling policy.

In particular, [14, 15] show how to validate programs with loose timing annotations encoded by bounded intervals. This technique extracts a finite subset from the infinite set of the timings allowed by the specification. Given a program that always terminates and without non-deterministic data choices, this technique detects all the assertion failures and the deadlocks.

These tools give interesting results for small and medium sized industrial examples. Using SCRV [13], a synchronization error was found in a model of a video decoder provided by STMicroelectronics. However, stateless model-checkers can only be applied to terminating programs without non-deterministic unbounded data inputs.

3.1.2. Translate then Verify

For programs that do not terminate, a second approach was investigated. The idea was to translate the SystemC/TLM program to be verified into another language, and then verify the translated program using an existing stateful model checker. This approach has first been applied to the RTL

level SystemC descriptions [4, 11]*. Many translations and languages have been proposed for the validation of transactional models, as in* [25]*, which translates SystemC/TLM programs into finite state machines* (*FSM*)*, similarly* [20]*, which describes abstraction techniques and a translation from SystemC/TLM to labeled Kripke structures. Most of these translations are manual, the first exception being the LusSy tool chain* [24]*, which automatically translates TLM models into synchronous automata with variables; it provides some simple abstraction techniques (e.g., abstract address representation). The LusSy tool chain has been connected to many model checkers, including symbolic model checkers based on BDD or SAT. Some minor examples have been successfully verified, but industrial examples face the state space explosion problem. There are now other automatic translation tools starting from SystemC, including* [17] *that can translate SystemC/TLM models into Uppall models, and allow verification of liveness properties and timing constraints.* [12] *translates TLM models into sequential C programs, in order to use verification tools dedicated to software.*

The state space explosion problem appears mainly because TLM models are mostly asynchronous. Thus, after each transition, there are many valid scheduling choices that should be explored. It is therefore suitable to use the model checkers for asynchronous programs as these model checkers have been specifically optimized to fight state space explosions arising from asynchrony. Translation of TLM programs into Promela [27] *has allowed TLM models to be verified using the SPIN checker which uses partial orders to reduce state spaces. Since then other translations into Promela have been presented* [22, 3]*, allowing the verification of larger models. Note that the translation defined in* [3] *was implemented in an automatic tool where other back-ends were available.*

Furthermore, we firstly proposed a translation of TLM to LOTOS [16, 26] *that enables verification of the benchmark of* [27] *for a slightly greater number of processes than using SPIN. The translation was fully manual, preventing the approach to scale up.*

Next, [7] *presented both an extension of our TLM to LOTOS translation and an application to an industrial case study. As shown in Figure 1, part of the SystemC/TLM code was translated into LOTOS while data-types and related operations were kept as C++ code, thus strongly reducing the translation effort. This paper showed that some properties can be checked on industrial code, but the amount of manual work still limited the efficiency of the approach.*

3.2. The CADP Verification Toolbox

Our goal is to detect synchronization errors between asynchronous processes of SystemC/TLM programs or to guarantee that the communication protocol they use is correct. For this kind of task one of the best techniques is model checking and in particular explicit model checking.

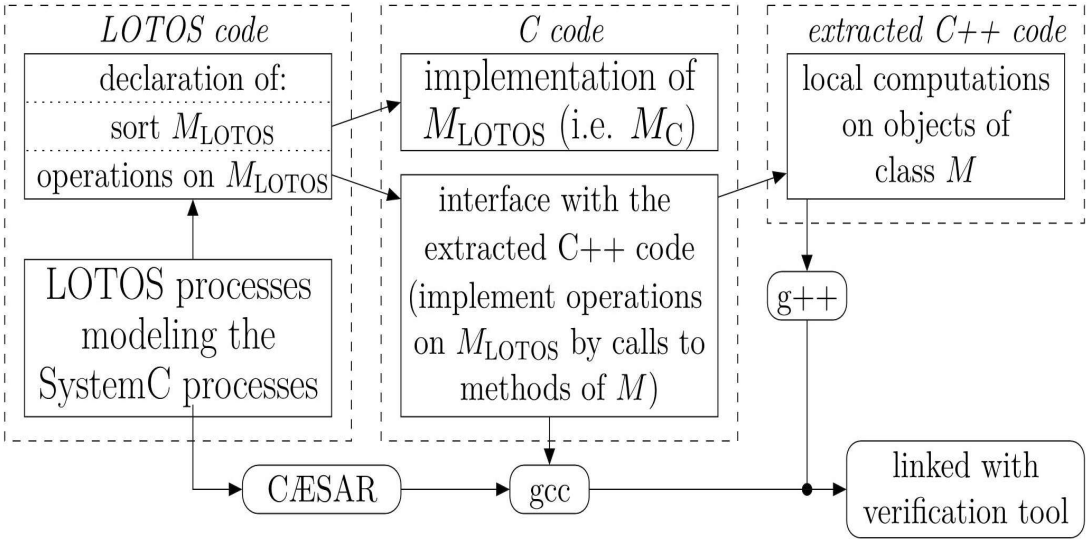

Figure 1. Verification of hybrid LOTOS/C++ models

```
R ← { initial state }  // set of remaining states
E ← ϕ  // set of explored states
while ( ∃ x ∈ R ) do
    for each transition x → y do
      add the transition x → y to the LTS
      if ( y ∈ E ∪ R ) then R ← R ∪ { y }
  R ← R \ { x }
  E ← E ∪ { x }
end while
```

Figure 2. Basic algorithm for LTS generation

A well-known explicit model-checker is SPIN. In this work, we investigated the use of another model-checker; namely CADP.

CADP ("Construction and Analysis of Distributed Processes") [9] *is a toolbox for the validation of communication protocols and distributed systems.*

The usual entry point for CADP is the language LOTOS. The ISO standard LOTOS [18] (*Language Of Temporal Ordering Specification*) *is a process algebra used to describe asynchronous concurrent processes communicating and synchronizing by rendezvous on gates. This language is well suited for designing communication protocols.*

The semantics of a LOTOS specification is formally defined by a state graph, also called an LTS (labeled transition system) – i.e. a set of states and transitions labeled by gates and offers between states.

CADP [8] *includes a compiler from LOTOS to LTS with many tools exploiting the LTS for simulation as well as model checking of modal μ-calculus formulae, equivalence checking, test generation and performance evaluation. The LOTOS to LTS compiler generates the LTS by executing all transitions of the system under verification; each visited state is recorded, so that each transition is executed once only. The algorithm is shown in Figure 2.*

4. Connecting SystemC/TLM with Formal Methods

4.1. The Architecture of TLM.Open

The CADP toolbox architecture is similar to the GNU compiler tool suite, with many front-ends and back-ends. There is one front-end per input language; the front-end reads a program and implements some basic analysis (e.g., type checking). Then there is one back-end per CADP feature, such as simulation, LTS generation, or on-the-fly property checking. The most used frontends are caesar.open which manages LOTOS programs and bcg_open which reads compressed and explicit LTS (bcg stands for "binary coded graphs"). All CADP front-ends connect with CADP back-ends using the OPEN/CÆSAR interface [6].

In this section, we present TLM.open which is a new CADP front-end allowing the use of the same back-ends as caesar.open and bcg_open. TLM.open is a C and C++ library that implements two interfaces: the SystemC interface and the OPEN/CÆSAR interface. The architecture of TLM.open is shown in Figure 3.

A SystemC/TLM program communicates with TLM.open through the SystemC interface as a SystemC/TLM program communicates with a SystemC simulator. The library TLM.open provides the same classes as the ASI SystemC simulator, including the sc_module, sc_port, sc_event, sc_signal, etc.

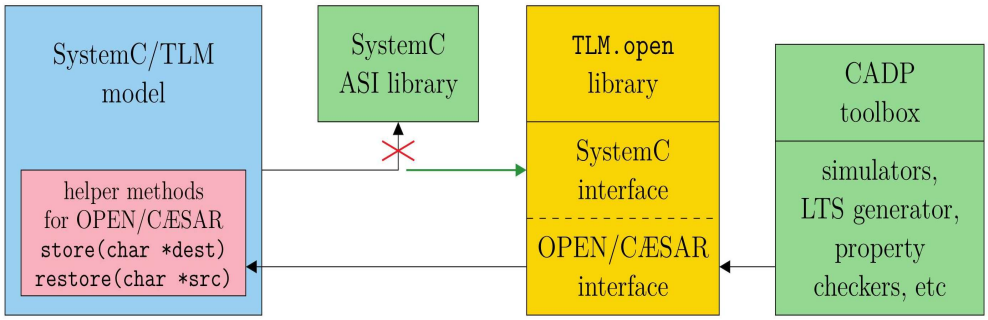

Figure 3. Overview of the verification framework. The model is linked with the SystemC TLM.open library instead of the ASI library

The OPEN/CÆSAR interface provides the operators required by the CADP model-checker itself. In order to implement the algorithm described by Figure 2, the following operators are required and must be provided by the TLM.open front-end:

- *Generation of the initial state.*
- *Enumeration and simulation of the transitions starting from a given state.*
- *Efficient storage of a state (requires comparison and hash functions).*

To simulate a transition, TLM.open executes the corresponding C++ code of the SystemC/TLM program. This C++ code is compiled with an unmodified C++ compiler such as g++. TLM.open does not parse the C++ code itself and does not produce LOTOS code.

The most difficult task is to store and restore the states of the SystemC/TLM program. The person who writes and verifies the SystemC/TLM program, called user in this paper, has to provide some additional functions that allow TLM.open to store the states of each SystemC module. To date, these additional functions have had to be written by hand. Thus, our approach is not fully automatic.

When TLM.open is used with the LTS generator of CADP, the result is an LTS with two kinds of transitions. Here, offers are only used to add information to the transitions, and have no impact on synchronizations or communications.

TE transitions indicate that time has elapsed; the offer gives the duration and the list of events triggered and processes awakened. For example, "TE !o(+41ms, VGAC.compute)" means that the SystemC clock has advanced 41 ms and the process "VGAC.compute" is now awake.

EXEC transitions represent the execution of a SystemC process; the offers name the executed SystemC process, the inputs of this process if the special rand () function of TLM.open was called (cf. Section 4.3.2) and the outputs generated using the overloaded puts () function. For example, "EXEC !VGAC.compute !o(image updated, IRQ sent)" means that the SystemC scheduler has executed the process "VGAC.compute" and this process has printed two messages "image updated" and "IRQ sent".

4.2. Storing and Restoring Program States

The TLM.open library includes a SystemC simulator. The state of this simulator consists of the state (i.e., the current value) of each object that has been instantiated. Some objects are described by SystemC classes (such as: sc_event, sc_signal, ...) and others are described by user classes. SystemC modules are hybrid: some class members are inherited from the base class sc_module but other members are defined by the user.

A stored state contains a copy of each value of the simulator state that may change during the simulation. A stored state must be as small as possible and does not use the same types as the simulator states: constant values are not stored, Boolean values can be grouped in one byte using bit fields.

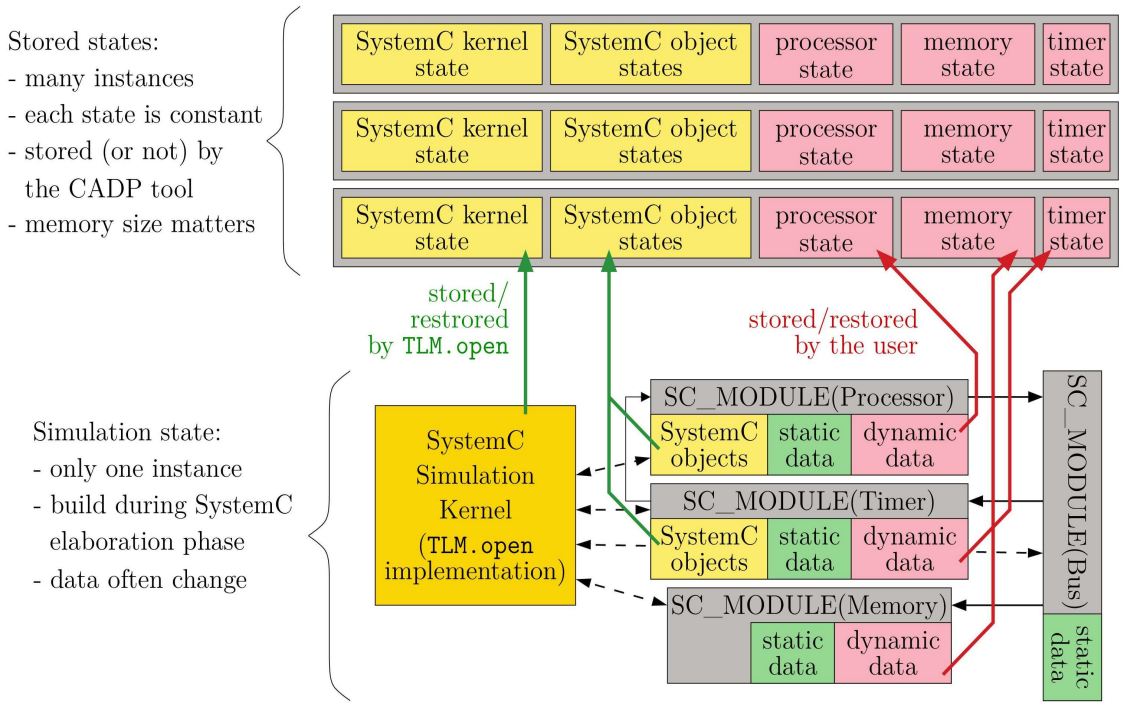

Figure 4. Memory layout: simulation state and stored states

All objects which are defined by a SystemC class are stored automatically by the TLM.open library. The other objects are stored using callback functions implemented by the user. Each SystemC module must provide the following functions:

- *size_t size () const: number of bytes needed to store a copy of the SystemC module.*
- *size_t alignment () const: specify whether padding bytes are needed.*
- *void store (char *dest) const: store the current state of the module in dest.*
- *void restore (const char *src): restore the state of the module according to the copy stored in src.*

The store () function must generate a canonical representation, so that state comparison can be done using memcmp () and hash functions can be generated automatically.

Implementing the functions size () and alignment () generally requires only one line of code for each. The store () function implementation contains two lines of code per module member on average; similarly the restore () function. Implementing these functions requires some manual work, but less than translating the whole model into another language.

Theoretically, generating automatically the store () and restore () functions should be simpler than translating the whole code, because it is not necessary to manage the code describing the behavior. However, such a generator would have to parse and manage a large part of C++, and the generated functions would likely be less efficient than those hand-written.

4.2.1. Storing Modules using Flat State

The user has many possibilities to implement the store and restore functions. The basic solution is to define a new struct type with one field per member of the SystemC module that is not constant and not managed directly by TLM.open. To store the state, the user filled this new type by copying values from the C++ class, and inversely, the user filled the C++ class by copying values from the struct type when the state must be restored. This is shown in the example below.

```
SC_MODULE(Foo) {
sc_event e;        // state stored by the tlm.open library
bool flag;          // dynamic data uint32_t data;  // dynamic data
const sc_time period; // static data, not stored
... // module implementation

// code below is used only by TLM.open
struct State { // container type
   bool flag; uint32_t data;
   void set(Foo *f ) const { f->flag=flag; f->data=data;}
   void set(const Foo *f ) { flag=f->flag; data=f->data;}
};
size_t size() const {return sizeof (State);}
size_t alignment() const {return 4 /*alignmentof (State)*/;}
void store(char *dst) const {
  reinterpret_cast<State*>(dst)->set(this);}
void restore(const char *src) {
  reinterpret_cast<const State*>(src)->set(this);}
}; // Foo
```

In this example, storing the state of an instance of Foo requires 8 bytes (i.e., $sizeof(Foo::State)$). If a program contains n modules $M_1, \ldots, M_n$, each module being stored using a type $M_i::State$, then each state stored consumes at least $\Sigma_{i=1}^{n} sizeof(M_i::State)$ bytes.

4.2.2. Storing Module using Hierarchical State

Most of the time, a transition modifies the state of only one or two modules. If storing a module consumes a lot of memory, it is then mostly better to use a hierarchical state. Using hierarchical states, the main state contains a pointer to the module state instead of the module state itself. When a transition is executed and the module has not been modified, then the new stored state contains only a pointer to the previously stored value.

Moreover, checking whether the module has been modified by the last transition is not enough. Even if the module has been modified, it is possible that we already have a copy of its current state. At the end of a transition, we search all the previous states of this module, which are stored in a container (hash table or binary tree). If this module state is encountered for the first time, then it is added to this container, or else we reuse the existing module state.

Here is how the Foo state could be recorded using a hierarchical state:

```
typedef std::set<const State*, StateCmp> state_set;
static state_set foo_states;
size_t Foo::size() const {return sizeof (State*);}
void Foo::store(char *dst) const {
    State *s = new State(); s->set(this);
  std::pair<state_set::iterator,bool> p = foo_states.insert(s);
  if (!p.second) delete s;  // This Foo state already exists,
                                  // so we reuse the previous version.
  *reinterpret_cast<const State**>(dst) = *p.first;}
```

To compare two states of the whole program, we just need to compare the pointers because identical module states are never stored in distinct memory locations.

In some cases, hierarchical states can significantly reduce the memory consumption. Moreover, whereas the OPEN/CÆSAR interface requires the main state to have a fixed size, hierarchical states allow a module to be stored whose size is not statically bound.

Internally, for all objects that are stored automatically, the TLM.open library uses a flat state for all objects except SystemC threads (SC_THREAD). Moreover, storing the state of a thread is done by copying its execution stack. Note that when yielding, the Quick Threads library used by SystemC

pushes the register contents and the program counter on top of the thread stack. As thread stack sizes vary during simulation because stacks may become large, and because at most one thread stack is modified during a transition, the hierarchical state technique here is more efficient than flat states.

4.3. Implementation of the OPEN/CÆSAR Interface

4.3.1. Generation of the Initial State

The generation of the initial state faces a technical problem. Moreover, SystemC and CADP do not use the same control flow:

♦ *A SystemC simulator creates the initial state by calling the function sc_main, which is implemented by the user, and the simulation starts when the user calls back the function sc_start from the sc_main function.*

♦ *A CADP back-end creates the initial state by calling the function CAESAR_START_STATE, which is implemented by the front-end and the verification starts after the function CAESAR_START_STATE returned.*

Thus, the CADP back-end calls the function CAESAR_START_STATE of TLM.open, and this function calls sc_main. The function CAESAR_START_STATE must return when sc_start is called, before sc_main returns. If one returns in advance of sc_start using a return statement or a C long jump or a C++ exception then the modules allocated on the stack are destroyed before they are used. The solution is to execute the sc_main function in a separated thread, which has its own stack and suspends this thread when sc_start is called. As we do not need real concurrency, this thread is implemented using the collaborative Quick Threads library, which is used to implement the SystemC threads too.

4.3.2. Enumerating the Transitions

The key function of the OPEN/CAESAR interface is CAESAR_ITERATE_STATE. This function must enumerate the transitions starting from a given stored state x. A transition is defined by a label s (a C string) and the successor state y.

There is at least one transition per eligible process. Assuming all transitions are deterministic, the TLM.open library behaves as follows:

1. *A SystemC process is selected.*

2. *The simulator is set according to the stored state x, by calling the restore function of each stored object (either a user function for modules, or a TLM.open library function for other SystemC objects).*

3. *The transition is executed, until the elected process yields back to the scheduler.*

4. *The new simulator state is stored in y, by calling the store function of each stored objects. The label s is created using the name of the elected process, and the outputs generated by the user using the puts () function.*

5. *This transition is sent to the back-end.*

6. *If there is another eligible thread, then go back to step 1.*

If no processes are eligible, then TLM.open can let the time elapse until a process is awoken, just like a regular SystemC simulator. In this case, a specific transition is generated with the label "TE". If no process can be awoken by a time elapse, then this means that x is in a deadlock state. In order to simulate inputs or a non determinism, the TLM.open library provides a rand (int MAX) function. From the user point of view, this function returns a number between 0 and MAX. In case of a simulation, an implementation would choose a number randomly. On the contrary, model checking requires an enumeration of all values. In order to generate the full LTS, each time TLM.open encounters a call to rand (), it records that another transition exists for the same process

for which values have already been tried. Thus, the same code will be executed MAX+1 times, generating as many LTS transitions. Because a transition may call rand () many times, TLM.open uses a stack to remember its position in the transition tree. Thus, all input combinations are finally generated (e.g., "x=rand (2); y=rand (3); wait ();" generates 3 × 4 = 12 transitions).

4.4. Features and Limitations

Most SystemC, TLM and C++ features can be used normally. However, some features require special care. As aforementioned, the functions puts () and rand () have a special meaning when used with TLM.open.

4.4.1. The sc_stop Function

SystemC provides a function sc_stop() to stop the simulation. Because all states that can be reached using a simulator must be reached using TLM.open, calling sc_stop() may not stop the generation of the LTS. With respect to the SystemC specifications, the effect of executing sc_stop() in a transition $x \rightarrow y$ is to eliminate all the successors of y. If other transitions are pending, then they are explored normally.

4.4.2. Recording the Current Time

A SystemC simulator, such as the ASI simulator, records the current date. The user can read this data using the function sc_timestamp (). Because this value is stored in the state, the state space becomes infinite for all programs containing a timed instruction in an unbounded loop. An example of such program is:

```
SC_THREAD(compute); ...
void compute() {
while (true) {wait(1, SC_SEC);}}
```

To allow the verification of this program, TLM.open provides an option to record only relative durations. This option disables the function sc_timestamp(). Using this option, the LTS of the program above has only two states and two transitions: a transition with gate "EXEC" leads from the initial state to the second state and another transition with gate "TE" leads back to the initial state.

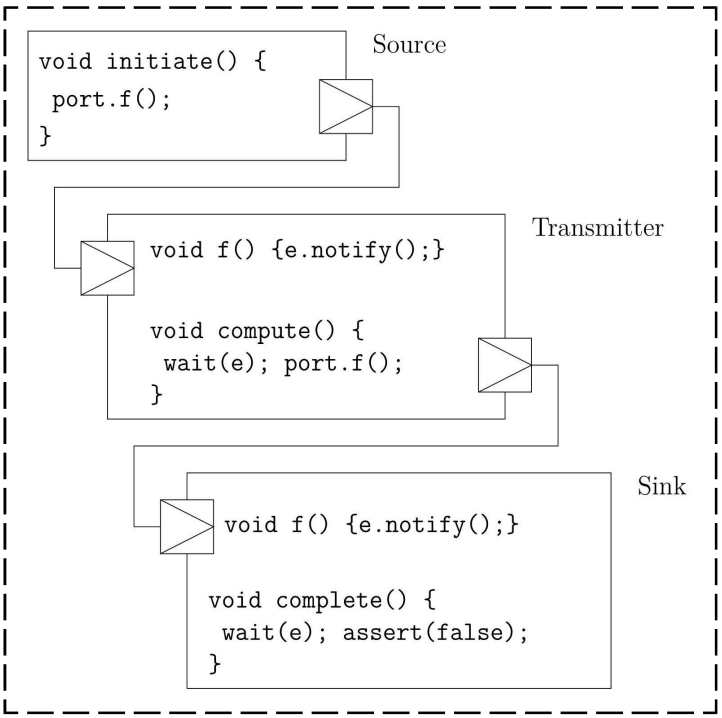

Figure 5. Source code of the chain benchmark for n = 1

4.4.3. Pointers and Dynamic Allocations

It is perfectly safe to use pointers when verifying a SystemC/TLM program with TLM.open. Both the pointer and the pointed value must be stored (respectively restored) when the module is stored (resp. restored).

However, dynamic allocations should not be used because a transition can be executed many times, calling new creates a memory leak (a second object will be created if the transition is executed again), and calling delete corrupts the memory (memory can be freed twice). There is one exception: a new statement can be used safely if the corresponding delete statement is found in the same transition.

If using dynamic allocation is necessary, then the user must define its own memory allocator. Next, the user must provide store and restore functions to manage the state of the memory allocator itself. Thus, when a state is restored, the memory allocator knows which objects are allocated and which memory locations are available.

5. Examples

5.1. The chain Benchmark

We evaluate our new front-end on the benchmark proposed in [27] *and reused in* [16]*. This benchmark consists of a chain of interrupt transmitter modules, whose length is parametrised by n. Modules communicate through transactions, and processes synchronize with events.*

Figure 5 presents the SystemC original benchmark for $n = 1$*. To increase* n*, one adds a transmitter module between the last transmitter and the sink module. There are always* $n+2$ *threads (functions named compute and process) and* $n+1$ *events (private attribute e of each module).*

It is very easy to use TLM.open to verify this benchmark because the modules do not contain any dynamic members. Thus, the store and restore functions can be left empty. The state of the SystemC events and of the SystemC threads (possibly including local variables) are stored automatically.

We have also tried TLM.open on a modified version of this benchmark. The modified version uses the SC_METHOD instead of the SC_THREAD. Using SC_METHOD makes the code more difficult to read, but accelerates the simulation and reduces the memory consumption. When replacing a SC_THREAD by a SC_METHOD, local variables have generally to be replaced by module members, and thus must be stored and restored by the user callback methods.

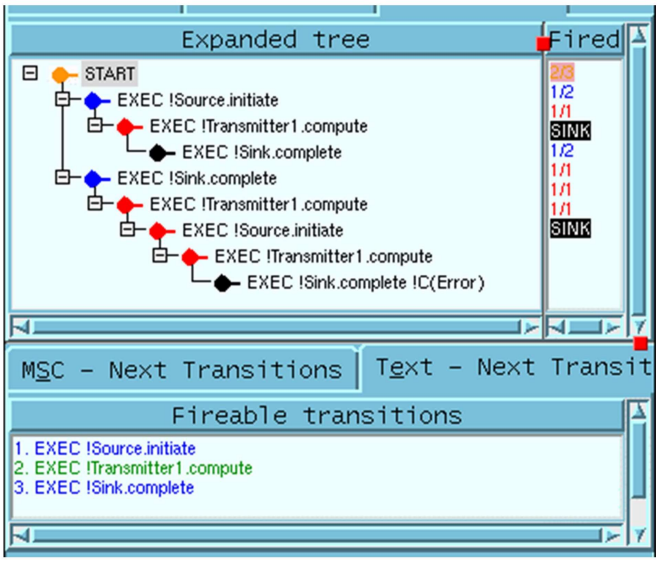

Figure 6. Screen-shot of the OCIS simulator of CADP (chain benchmark, *n* = 1)

n =	3	7	11	15	17	19	21
LTS generation (SC_THREAD)	1.1s	1.2 s	2.3 s	35.3 s	193 s	844 s	4314 s
LTS generation (SC_METHOD)	1.1s	1.1 s	1.5 s	11.8 s	62 s	268 s	1445 s
state number	62	1022	16,382	262,142	1,048,574	4,194,302	16,777,214
state number after minimization	47	767	12,287	196,607	786,431	3,145,727	*n.a.*

Table 1. Results of the experiments using TLM.open

Among the CADP tools that can be used, there is ocis, an interactive simulator with backtracking. Figure 6 provides a screen-shot of this tool. Also, for small values of n, the LTS can be fully generated and displayed (cf. Figure 7).

Table 1 presents the results for the generation of the full LTS, using a Macbook machine with 4 GB of memory. For comparison, [27] *verifies this benchmark up to* $n = 15$ *(47 seconds), and* [16] *verifies this benchmark up to* $n = 19$ (8293 *seconds for* $n = 19$, 60.2 *seconds for* $n = 15$). *These results show a significant improvement compared to the previous approach based on the translation into Promela or LOTOS. The efficiency of TLM.open can be explained by two points:*

♦ *One transition in the LTS corresponds exactly to one SystemC transition (i.e., the execution of a process between two wait statements.) There are no additional transitions used to mimic the behavior of the SystemC scheduler.*

♦ *The memory size of a state is kept as small as possible, allowing the model checker to store more states.*

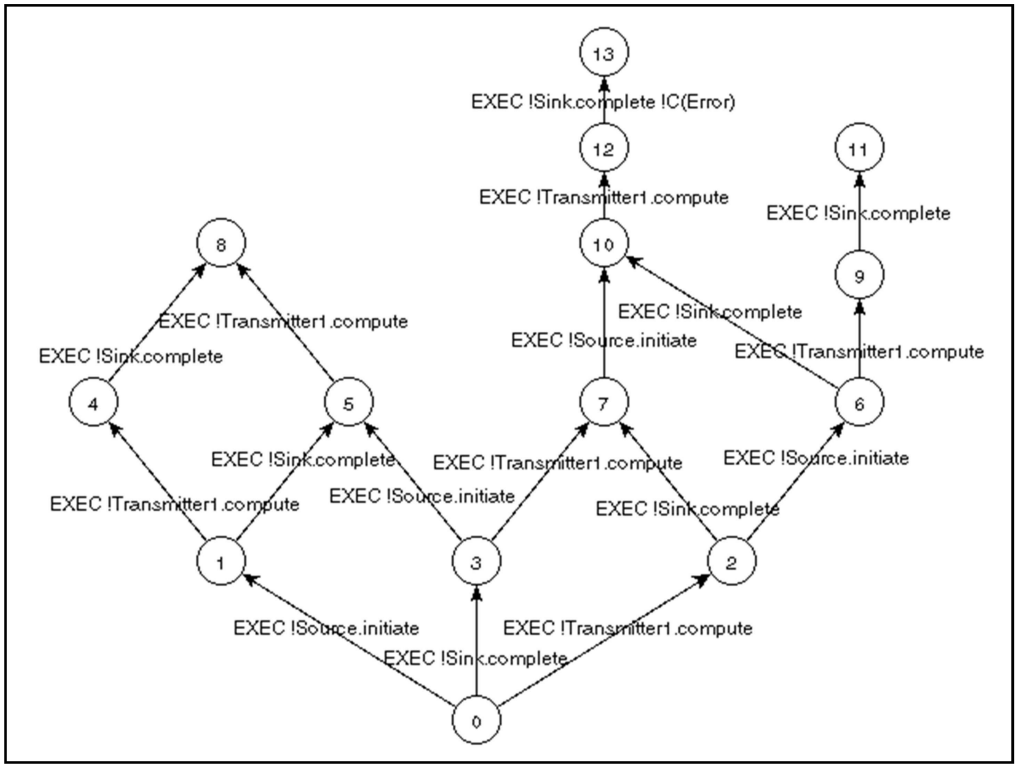

Figure 7. The LTS of the chain benchmark for *n* = 1 (output of bcg_edit)

Journal of Information Organization Volume 14 Number 1 March 2024

The modified benchmark in which SC_THREADs have been replaced by SC_METHODs gives identical LTSes. However, the generation is three times faster, and the memory consumption is reduced: for $n = 19$, generating the LTS for the original benchmark requires 650 MB whereas the modified version requires only 387 MB.

In this experiment, SC_THREADs are stored using hierarchical states. We have tried another implementation using only flat states. Using the original benchmark with SC_THREADs, the flat state technique leads to an explosion of the memory consumption: 700 MB instead of 3.5 MB for $n = 12$ and the LTS generation is about 1.5 times slower.

5.2. The LusSy Benchmark

The thesis [23] describes another SystemC/TLM benchmark, which is similar to the chain benchmark. The main difference is that the LusSy benchmark uses real transactions which are routed by a bus model.

It is worth noting that LusSy has a special interpretation of the timing annotations [23]. TLM.open provides an option to mimic the semantics of LusSy. This allows a greater number of schedules than the official specification, because it considers that all durations are equal.

Using TLM.open, instrumenting this benchmark with store () and restore () functions is trivial for all modules but the bus model because the whole state is contained in SC_THREAD stacks, which are automatically stored and restored. The bus model requires about 40 additional lines of code, used for storing and restoring the list of pending transactions. When using LusSy, no additional code is needed. However, LusSy does not use the bus model code. Indeed, LusSy is currently restricted to a few bus models for which a corresponding automaton model has been manually provided. Modeling a bus using automata requires more work and knowledge than adding store() and restore() functions. Therfore using LusSy is not easier than using TLM.open.

Table 2 provides the results obtained with TLM.open. It appears that TLM.open uses less memory than LusSy combined with SMV. Thus, TLM.open can verify this benchmark up to $n = 18$, whereas LusSy does not work over $n = 13$ (with a common memory limit fixed at 512 MB). For $n = 12$, TLM.open needs only two seconds where LusSy+SMV spends over one hour.

n	Memory Consumption	Time
$n = 15$	30.8 MB	11.3 sec
$n = 16$	64.6 MB	24.1 sec
$n = 17$	136.1 MB	52.6 sec
$n = 18$	289.8 MB	116.3 sec

Table 2. Results for the LusSy benchmark verification using TLM.open

5.3. Application to a Timer

This subsection illustrates the features provided by TLM.open by showing how it can be applied to a simple but realistic example. We consider a timer with two registers; PERIOD and ACK.

- *Writing a non-null value to PERIOD starts the timer.*
- *When enabled, the timer generates an interrupt periodically.*
- *Writing to ACK acknowledges the interrupt.*
- *Writing 0 to PERIOD stops the timer.*

We have 4 TLM models for this timer. The first comes from the SimSoC project [19]; the second

is identical to the first with a bug fix; the third and the fourth were provided respectively by an engineer and a PhD student.

In order to verify the first TLM model, which contains 80 lines of code, we had to write 17 additional lines of code to implement the store () and restore () functions. The timer verification requires the design of an environment modeling the commands generated by the embedded software. For this example, we decided to describe the environment using SystemC code. Here, the code of the environment process:

```
void compute () {
switch (rand(5)) {
case 0: puts("stop"); port->write(Timer::PERIOD_REG_OFFSET,0); break;
case 1: puts("start"); port->write(Timer::PERIOD_REG_OFFSET,5); break;
case 2: puts("ack"); port->write(Timer::ACK_REG_OFFSET,1); break;
case 3: {
std::ostringstream oss;
oss <<"read_period:" <<port->read(Timer::PERIOD_REG_OFFSET);
puts(oss.str().c_str());
break;}
case 4: {
std::ostringstream oss;
oss <<"read_ack:" <<port->read(Timer::ACK_REG_OFFSET);;
puts(oss.str().c_str());
break;}
case 5: puts("wait"); next_trigger(5,sc_core::SC_MS); return;
}
next_trigger(sc_core::IMMEDIATE_WAKE_UP);
}
```

Note that we trigger the timer with only one specific period. Explicit model-checking does not permit the verification of this model for all values of the period. Thus, we have to assume that the presence of bugs does not depend on this particular period.

The last line uses a special feature of TLM.open: the process yields back to the scheduler but remains eligible. This statement is similar to the yield () statement introduced in [15]*. The rationale of this statement is to break critical sections that would exist in the model but not in the real system.*

Firstly, we applied on-the-fly property checking. The property checker of CADP revealed an error in the first version: for some particular schedules, the timer could generate an interrupt after it was stopped. A counter-example was automatically shown, allowing us to fix the bug. Another minor bug was found in the third version.

Secondly, we applied equivalence checking. We generated the LTS of each timer TLM model, we hid the internal transitions and we minimized them according to the branching equivalence. We got the proof that the second and the forth version are bisimilar modulo branching equivalence. It means that if one contains an error, the other contains the same error. Obviously, the first and third versions are not bisimilar, since they contain distinct errors.

5.4. Application to a Basic System

In order to evaluate the behavior of TLM.open on a system made of many modules, we studied a basic system that was originally used for practical work. This system was implemented on FPGA. It contains a MicroBlaze processor, a VGA controller, plus the usual and mandatory peripherals: bus, memory, timers, interrupt controller. In the SystemC/TLM model, the user can model the processor, by either using a native wrapper or an instruction set simulator (ISS). The embedded software compute images and manage the configuration of the peripherals.

For the validation of the embedded software, we decided to use the native wrapper instead of the ISS. On the one hand, there is nothing that prevents us using the native wrapper for this software (i.e., no inline assembly code and no dynamic code loading). On the other hand, using the ISS

would multiply the number of states: 1 state per binary instruction with the ISS instead of one state per explicit synchronization point with the native wrapper.

Thus, we have instrumented all modules with store () and restore () methods. Then, we changed the output functions so that traces are added to LTS labels instead of sent to the terminal. Finally, we made some simplifications: 1. we have disconnected the graphical library used by the VGA module, which means that during model checking we do not display the simulated VGA screen. 2. We have simplified the TLM protocol so that it no longer uses a transaction pool because the transaction pool mechanism is only a trick to make simulations a little faster.

Using TLM.open, we generated the LTS corresponding to this basic SystemC model for one embedded software. Using the bcg_min tool of CADP, the LTS can be reduced to a minimal LTS. This minimal LTS is small enough to be read by human. By observing this LTS, we notice that in some cases the processor was receiving an interruption before any other module raised one. The rationale was a missing dont_initialize() in the SystemC code. Because the occurrence of this error depends on the scheduling, this bug had not been noticed before we used TLM.open. After fixing this bug, we generated the minimized LTS again. This second LTS is represented by Figure 8.

Finally, we tried to add some errors in the embedded software, such as changing an initial value or disabling a register write in order to verify that all errors can be discovered during model checking. For each error, we got either a SystemC error message (such as assertion failure coming from the TLM code) during LTS generation, or an LTS that was not equivalent to the reference once (equivalence checked using the CADP tool bisimulator).

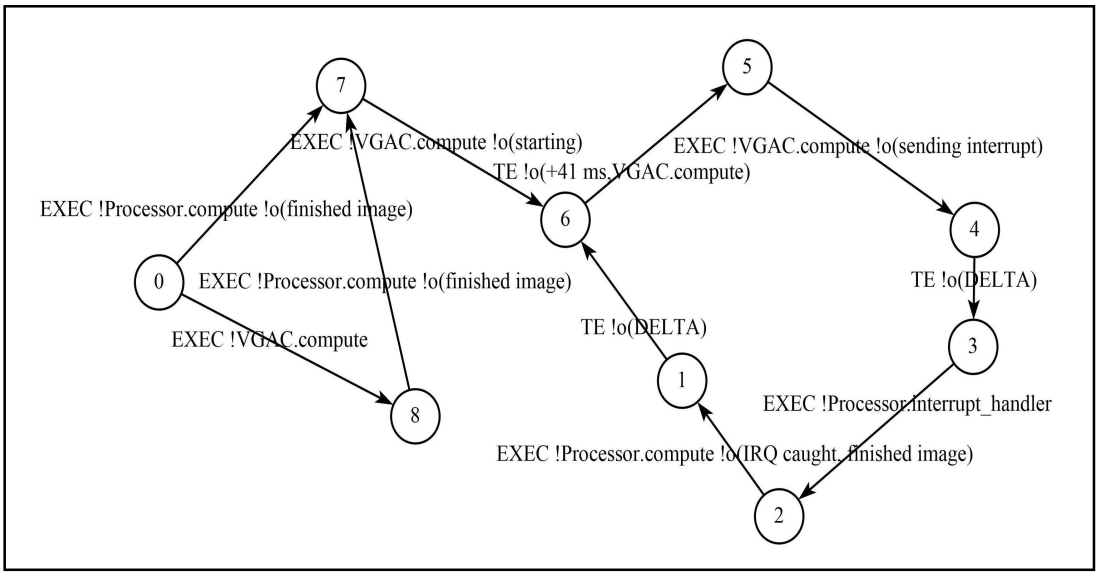

Figure 8. LTS generated for the basic system, after minimization

6. Conclusion

We presented a new framework for the verification of SystemC/TLM programs. Our new System-C/TLM front-end avoids the need to translate the whole SystemC/TLM program into another language. Compared to approaches based on manual translation, the verification using TLM.open is much simpler: there are less lines of code to write and the engineers do not need to learn a new modeling language. Moreover, TLM.open allows better scaling than previous works. Thanks to the numerous tools of CADP, it is now possible to check complex properties and to test the equivalence of two SystemC/TLM programs.

Note that TLM.open can be used with pure SystemC programs also (i.e., programs not using TLM). The rationale of calling our tool TLM.open instead of SystemC.open is related to the abstraction

level: the CADP verification toolbox is optimized for asynchronous processes. SystemC/TLM models use asynchronous processes, but SystemC programs modeling a system at a lower level of abstraction use synchronous processes. In order to verify synchronous processes, symbolic model-checker based on BDD or SAT, are in general more efficient than CADP. Thus, TLM.open can be used for pure SystemC programs, but is not likely to be the most efficient tool. As explained in [7], the most difficult task when verifying a SystemC/TLM program is to extract an abstract model that is simple enough to be formally verified. Thus, the main further work is to integrate TLM.open in the design flow in such a way that this task becomes simple and safe. Additionally, it would help to automatize the generation of the store () and restore methods.

References

[1] Accellera Systems Initiative. (2011). IEEE 1666 Standard: SystemC Language Reference Manual. Retrieved from http://www.accellera.org

[2] Blanc, Nicolas., Kroening, Daniel. (2008). Race analysis for SystemC using model checking. In *2008 International Conference on Computer-Aided Design (ICCAD'08)* (pp. 356–363). IEEE. doi:10.1145/1509456.1509540

[3] Cimatti, Alessandro, Narasamdya, Iman., Roveri, Marco. (2013). Software model checking SystemC. *IEEE Transactions on CAD of Integrated Circuits and Systems, 32*(5), 774–787. doi:10.1109/TCAD.2012.2232351

[4] Drechsler, Rolf., Große, Daniel. (2002). Reachability analysis for formal verification of SystemC. In *2002 Euromicro Symposium on Digital Systems Design (DSD 2002), Systems-on-Chip* (pp. 337–340). IEEE Computer Society. doi:10.1109/DSD.2002.1115387

[5] Flanagan, Cormac., Godefroid, Patrice. (2005). Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLANSIGACT Symposium on Principles of Programming Languages, POPL 2005* (pp. 110–121). ACM. doi:10.1145/1040305.1040315

[6] Garavel, Hubert. (1998). Open/cæsar: An OPen software architecture for verification, simulation, and testing. In *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 – April 4, 1998, Proceedings* (pp. 68–84). Springer. doi:10.1007/BFb0054165

[7] Garavel, Hubert, Helmstetter, Claude, Ponsini, Olivier., Serwe, Wendelin. (2009). 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)* (pp. 46–55). IEEE Computer Society. doi:10.1109/MEMCOD.2009.5185377

[8] Garavel, Hubert, Lang, Frédéric., Mateescu, Radu. (2002). An overview of CADP 2001. *European Association for Software Science and Technology (EASST) Newsletter, 4*, 13–24.

[9] Garavel, Hubert, Mateescu, Radu, Lang, Frédéric., Serwe, Wendelin. (2007). CADP 2006: A toolbox for the construction and analysis of distributed processes. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3–7, 2007, Proceedings* (Vol. 4590, pp. 158–163). Springer. doi:10.1007/978-3-540-73368-3_18

[10] Ghenassia, Frank (Ed.). (2005). *Transaction-Level Modeling with SystemC. TLM Concepts and Applications for Embedded Systems.* Springer. ISBN 0-387-26232-6.

[11] Große, Daniel., Drechsler, Rolf. (2005). CheckSyC: an efficient property checker for RTL SystemC designs. In *International Symposium on Circuits and Systems (ISCAS 2005)* (Vol. 4, pp. 4167–4170). IEEE. doi:10.1109/ISCAS.2005.1465549

[12] Große, Daniel, Le, Hoang M.., Drechsler, Rolf. (2010). Proving transaction and system-level properties of untimed SystemC TLM designs. In *8th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)* (pp. 113–122). IEEE Computer Society. doi:10.1109/MEMCOD.2010.5558643

[13] Helmstetter, Claude. (2007). Validation de modèles de systems sur puce enprésenced' or donnancementsindéterministes et de temps imprécis. PhD thesis, INPG, Grenoble, France. Retrieved from http://tel.archives-ouvertes.fr/tel-00350929

[14] Helmstetter, Claude, Maraninchi, Florence., Maillet-Contoz, Laurent. (2006). Test coverage for loose timing annotations. In *Formal Methods: Applications and Technology, 11th International Workshop, FMICS 2006 and 5th International Workshop PDMC 2006, Bonn, Germany, August 26–27, and August 31, 2006, Revised Selected Papers* (Vol. 4346, pp. 100–115). Springer. doi:10.1007/978-3-540-70952-7_7

[15] Helmstetter, Claude, Maraninchi, Florence., Maillet-Contoz, Laurent. (2009). Full simulation coverage for SystemC transaction-level models of systems-on-a-chip. *Formal Methods in System Design, 35*(2), 152–189. doi:10.1007/s10703-009-0075-z

[16] Helmstetter, Claude., Ponsini, Olivier. (2008). A comparison of two SystemC/TLM semantics for formal verification. In *6th ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2008)* (pp. 59–68). IEEE Computer Society. doi:10.1109/MEMCOD.2008.4547687

[17] Herber, Paula, Pockrandt, Marcel., Glesner, Sabine. (2011). Transforming SystemC transaction level models into UPPAAL timed automata. In *9th IEEE/ACM International Conference on Formal Methods and Models for Codesign, MEMOCODE 2011, Cambridge, UK, 11–13 July, 2011* (pp. 161–170). IEEE. doi:10.1109/MEMCOD.2011.5970523

[18] ISO/IEC. (1989). Lotos – a formal description technique based on the temporal ordering of observational behaviour. *International Standard 8807*. International Organization for Standardization – Information Processing Systems – Open Systems Interconnection, Genève.

[19] Joloboff, Vania., Helmstetter, Claude. (2008). SimSoC: A SystemC TLM integrated ISS for full system simulation. In *Circuits and Systems, 2008. APCCAS 2008. IEEE Asia Pacific Conference on*. IEEE. doi:10.1109/APCCAS.2008.4746381

[20] Kroening, Daniel., Sharygina, Natasha. (2005). Formal verification of SystemC by automatic hardware/software partitioning. In *3rd ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2005), 11–14 July 2005, Verona, Italy, Proceedings* (pp. 101–110). IEEE. doi:10.1109/MEMCOD.2005.1487900

[21] Kundu, Sudipta, Ganai, Malay K.., Gupta, Rajesh. (2008). Partial order reduction for scalable testing of SystemC TLM designs. In *Proceedings of the 45th Design Automation Conference, DAC 2008, Anaheim, CA, USA, June 8–13, 2008* (pp. 936–941). ACM. doi:10.1145/1391469.1391706

[22] Marquet, Kevin, Moy, Matthieu., Jeannet, Bertrand. (2011). Efficient Encoding of SystemC/TLM in Promela. In *Workshop on Design, Analysis and Tools for Integrated Circuits and Systems at the International MultiConference of Engineers and Computer Scientists 2011, DATICS-IMECS* (pp. 1039–1044). Retrieved from http://www.iaeng.org/publication/IMECS2011/IMECS2011_pp1039-1044.pdf

[23] Moy, Matthieu. (2005). *Techniques and Tools for the Verification of Systems-on-a-Chip at the Transaction Level*. PhD thesis, INPG, Grenoble, France. Retrieved from http://www-verimag.imag.fr/~moy/phd/

[24] Moy, Matthieu, Maraninchi, Florence., Maillet-Contoz, Laurent. (2005). 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. doi:10.1007/s10617-006-9044-6

[25] Niemann, Bernhard., Haubelt, Christian. (2006). Formalizing TLM with communicating state machines. In *Forum on specification and Design Languages, FDL 2006, September 19–22, 2006, Darmstadt, Germany, Proceedings* (pp. 285–293). ECSI.

[26] Ponsini, Olivier., Serwe, Wendelin. (2008). A schedulerless semantics of TLM models written in SystemC via translation into LOTOS. In *FM 2008: Formal Methods, 15th International Symposium on Formal Methods, Turku, Finland, May 26–30, 2008, Proceedings* (Vol. 5014, pp. 278–293). Springer. doi:10.1007/978-3-540-68237-0_20

[27] Traulsen, Claus, Cornet, Jérôme, Moy, Matthieu., Maraninchi, Florence. (2007). A SystemC/TLM semantics in Promela and its possible applications. In *Model Checking Software, 14th International SPIN Workshop, Berlin, Germany, July 1–3, 2007, Proceedings* (Vol. 4595, pp. 204–222). Springer. doi:10.1007/978-3-540-73370-6_14