Verification of an Industrial SystemC/TLM Model Using LOTOS and CADP*

Hubert Garavel, Claude Helmstetter, Olivier Ponsini, and Wendelin Serwe
INRIA Grenoble Rhône-Alpes / Vasy
655, avenue de l'Europe, F-38330 Montbonnot St Martin, France

Abstract

SystemC/TLM is a widely used standard for system level descriptions of complex architectures. It is particularly useful for fast simulation, thus allowing early development and testing of the targeted software. In general, formal verification of SystemC/TLM relies on the translation of the complete model into a language accepted by a verification tool. In this paper, we present an approach to the validation of a SystemC/TLM description by translation into LOTOS, reusing as much as possible of the original SystemC/TLM C++ code. To this end, we exploit a feature offered by the formal verification toolbox CADP, namely the import of external C code in a LOTOS model. We report on experiments of our approach on the BDisp, a complex graphical processing unit designed by STMicroelectronics.

1. Introduction

Embedded systems combine on a single circuit several hardware components with embedded software. To cope with the increasing complexity and time-to-market pressure, system level descriptions are increasingly used as reference descriptions. SystemC/TLM [1, 2] is a widely used standard for describing a complex system-on-chip at system level. Due to its fast simulation engines, SystemC/TLM is particularly useful for testing the embedded software, even before the hardware architecture is completely specified.

A system level description generally consists of a set of interconnected modules executing concurrently. Dealing with concurrency is known to be complex due to the many possible interleavings. Because this also holds for a system-on-chip, verification of system level descriptions is required to detect design errors as soon as possible.

The development of a verification tool dedicated to SystemC/TLM is a complex task, because a SystemC/TLM description can contain arbitrary C++ code (SystemC and TLM are actually implemented as C++ libraries). Existing approaches to formal verification of SystemC/TLM translate the complete model into a language accepted by a verification tool (see section 3.2). Such a complete translation requires a significant effort and has the inconvenience that the model under verification is not exactly the same as the reference SystemC/TLM model used for design and test.

In this paper, we present an approach for the validation of a SystemC/TLM description by translation into LOTOS and use of the CADP toolbox [4]. We extend the approach of [3] by reusing as much as possible of the original C++ code. In particular, we reuse a large part of the C++ code (namely the parts corresponding to local computations of processes) to implement LOTOS operations.

We have experimented with our approach on a complex component of an industrial circuit, namely the BDisp graphical processing unit designed by STMicroelectronics (≈25,000 lines of SystemC/TLM). This led to optimizations of our approach, reducing the memory requirements for formal verification.

The remainder of this paper is organized as follows. Section 2 gives an overview of SystemC and TLM. Section 3 presents related work. Section 4 gives an overview of LOTOS. Section 5 presents our approach to the verification of SystemC/TLM. Sections 6 and 7 present the BDisp model and its LOTOS translation. Section 8 reports on our experiments with the verification toolbox CADP. Finally, Section 9 concludes.

2. SystemC and TLM

SystemC [1], a standard published by the *Open SystemC Initiative* (OSCI), is a C++ library providing 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 more *processes* (`SC_THREAD`) describing the behavior of the system. SystemC processes interact using shared memory or communication channels, and are synchronized using SystemC events

*This work has been partially funded by the French government and by Conseil Général de l'Isère as part of the Multival project (pôle de compétitivité Minalogic).

(`sc_event e`, `e.notify()`, `wait(e)`) and timing annotations (`sc_time t`, `wait(t)`).

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 at a time. 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 [2] built upon SystemC provides a *transaction* mechanism that encapsulates communication protocols (data transfer and synchronization) between modules, accelerating both model design and simulation. Using a transaction, a process in an *initiator* module can directly call the methods exported by a *target* module. A process can thus read many values from a memory or set many registers of a peripheral without any costly inter-process synchronization (no context switch is required). Names and attributes of exported functions depend on the *protocol*. In general, a protocol provides at least `read` and `write` functions, but a protocol for a transaction modeling an interrupt might be reduced to only one function without argument. 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.

3. Existing approaches to TLM verification

The OSCI provides an open-source simulator for SystemC/TLM and a library SCV to ease test generation. However, the OSCI does not provide tools for formal verification. Moreover, while the SystemC specification allows many schedulings for a given test case, the OSCI simulator exhibits always the same scheduling. Thus, even if an execution leads to the expected result, another execution with a different scheduling may be erroneous. This issue has been addressed by many publications. Two approaches have been investigated: stateless model checking of a SystemC/TLM program, and translation of a SystemC/TLM program to a language for which a model checker is available.

3.1. Stateless model checking

The first approach is based on the execution of the SystemC/TLM program using an enhanced SystemC simulator capable of exploring each test case with many schedulings. One solution is to implement a random scheduler; this increases error detection but the resulting coverage is uncertain. Another option is to try all valid schedulings, but this technique does not scale up to industrial examples. In contrast, stateless model checking techniques based on dynamic partial order reduction [5, 6, 7] select only a small subset of the valid schedulings, and so scale up to medium sized industrial examples. The generated subset of schedulings is large enough to guarantee that all local errors (i.e. assertion violations) and deadlocks are found.

A SystemC/TLM program can contain loose timing annotations (e.g., an interval of realistic values $[T-d, T+d]$ instead of an exact duration T) if the timing of the real system is uncertain. At runtime, a simulation engine can draw a particular value inside each interval. Again, random simulation does not ensure coverage. Dynamic partial order reduction combined with linear programming techniques can be used to generate a set of schedulings and timings that guarantee to find all local errors and deadlocks for a given test case [8].

All the stateless model checking techniques [5, 8, 6, 7] that have been implemented for SystemC/TLM programs can be applied only to bounded test scenarios.

3.2. Translation-based approaches

For programs that do not terminate, a second approach has been investigated. The idea is to translate the SystemC/TLM program to be verified to 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 [9, 10].

Many translations and languages have been proposed for the validation of transactional models, like [11], which translates SystemC/TLM programs into finite state machines (FSM), or like [12], which describes abstraction techniques and a translation from SystemC/TLM to labeled Kripke structures. Like our approach, most of these translations are manually, a notable exception being the LusSy tool chain [13], 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 small examples have been successfully verified, but industrial examples face the state space explosion problem.

The state space explosion problem appears chiefly because TLM models are mainly asynchronous. Indeed, 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 check-

ers have been specifically optimized to fight state space explosion arising from asynchrony. For example, the SPIN model checker uses partial orders to reduce state spaces; a translation of TLM to Promela is described in [14], allowing the use of SPIN to verify TLM models. Also, we recently proposed a translation of TLM to LOTOS [15] that enables verification of the benchmark of [14] for a slightly greater number of processes than using SPIN; we pursue this line of work in the present paper.

All the translations we have presented in this subsection have given successful verification results only on small examples (a few hundred lines of code). This paper tries to go beyond this limit by addressing larger problems, namely an industrial example of about 25,000 lines of SystemC/TLM code.

4. LOTOS

The ISO standard LOTOS [16] (Language Of Temporal Ordering Specification) is a process algebra used to describe asynchronous concurrent processes communicating and synchronizing by rendezvous on gates. In the following we briefly introduce the main concepts of LOTOS occurring in the examples of this paper; for a complete description of LOTOS, we refer the reader to existing tutorials, such as [17].

LOTOS specifications are composed of a data part and a behavior part. Data values and operations are described by algebraic specifications in the style of ACTONE [18]. Types define a collection of sorts, operations on sorts, and equations describing the meaning of operations.

Behaviors are expressed by terms combining processes with algebraic operators. Figure 1 gives a simplified grammar of LOTOS behaviors; lower case identifiers stand for terminals and upper case identifiers for non terminals (P is a process name, G a gate name, X a variable name, S a sort name, and V a value expression).

As usual in process algebras, the semantics of a LOTOS specification is defined operationally by a translation of LOTOS into a *Labeled Transition System* (LTS). Here, we only sketch the meaning of behavioral operators. A communication "$G\ O_1\ ....\ O_n$; B" on a gate G allows to communicate several values O_i, called offers, either for emission (!) or reception (?); then behavior B is executed. "B_1 [] B_2" specifies a nondeterministic choice between behaviors B_1 and B_2. "B_1 |[$G_1, ..., G_n$]| B_2" specifies the parallel composition of B_1 and B_2 synchronizing on the gates G. Synchronization on gates with offers only occurs if the offers are compatible (same number of offers, same types, and same or compatible values). A behavior B can be guarded by a Boolean expression V: "[V] -> B" specifies that B will only be executed if V is true. "let X:S=V in B" allows to define a variable X of sort S that is initialized to value V and used in B. Finally, a behavior B can be given a name by declaring a process P parameterized by sets of gates (G_i) and of variables (X_i of sort S_i) as follows: "process P [G_1, ..., G_m] (X_1:S_1, ..., X_n:S_n): exit := B endproc".

B ::=	$G\ O_1\ ...\ O_n; B$	*(communication)*
	B_1 [] B_2	*(choice)*
	B_1 |[$G_1, ..., G_n$]| B_2	*(parallel)*
	exit($V_1, ..., V_n$)	*(termination)*
	[V] -> B	*(guard)*
	let $X : S = V$ in B	*(variable definition)*
	P [$G_1, ..., G_m$] ($V_1, ..., V_n$)	*(process call)*
O ::=	$!V \mid ?X : S$	*(offer)*

Figure 1. Grammar of LOTOS (excerpt)

5. Translating a TLM model into LOTOS

To translate an untimed SystemC/TLM model into a LOTOS specification, we extend the translation approach described in [3]. This approach splits a SystemC/TLM model in two parts that are handled differently:

1. communications and synchronizations between SystemC/TLM processes, i.e., the SystemC/TLM specific code; the translation of this code into LOTOS behaviors is described in [3].

2. local computations within a single module, e.g., classes, methods, sequential control, data manipulation; the translation of this code into LOTOS data types is not detailed in [3].

The present paper reduces this translation effort by reusing most of the non SystemC/TLM specific C++ code of the original model. Our approach takes advantage of a particular capability of CÆSAR (the LOTOS to C compiler included in the CADP verification toolbox [4]), which allows the possibility to implement LOTOS sorts (respectively, LOTOS operations) by external C data types (respectively, C functions).

For each SystemC/TLM module described by a C++ class M, we create a corresponding LOTOS model importing C/C++ code, as follows:

1. We define, in C, a compact representation M_{C} of the class M. A value of type M_{C} is called *module state*. This type will be used by the model checker to store and compare the explored states.

2. We define a LOTOS sort M_{LOTOS} corresponding to M_{C}, and we declare a set of LOTOS operations allowing to access or modify a module state (i.e., an element of M_{C}).

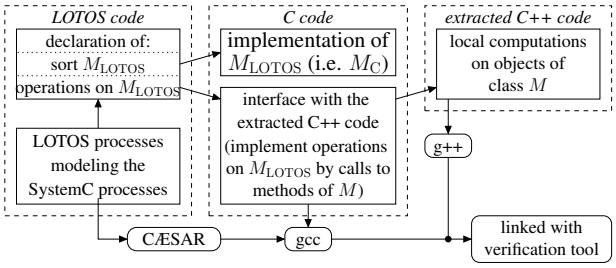

Figure 2. LOTOS model reusing C++ code

3. We translate each SystemC/TLM process of M into a LOTOS process, using the aforementioned LOTOS operations for the local computations, and applying the translation rules of [3] for the communications.
4. We extract from the original SystemC/TLM module the C++ code needed to implement the LOTOS operations.
5. We implement an interface between this C++ code extracted in the previous step, and the LOTOS code (compiled to C using the LOTOS to C compiler CÆSAR).

The following subsections detail these five steps, using the following SystemC/TLM module M as a running example.

```
class M: public sc_module {
 sc_port<...> tlm_port;
 data_t data; ...
 void compute() { /*SC_THREAD*/
  while(!stream.full()) {
   addr+=4; data=tlm_port.read(addr);
   stream<<data; wait();}}
};
```

5.1. Definition of the C type M_{C}

The first step is to define a compact representation M_{C} of the SystemC/TLM module described by the class M. M_{C} is a C structure (`struct` type) M_{C} that contains all the members of M the values of which are not constant. Notice that the C++ class M is aimed at fast simulation, whereas M_{C} is aimed at state space exploration and verification. Therefore, reducing the memory footprint of M_{C} is more important than reducing the access time to each of its fields. Hence, each field of M_{C} should be represented with the minimum number of bits (e.g., no more than one bit for a boolean value) and padding bits should be avoided as far as possible. Furthermore, each value of M_{C} should have a unique memory representation to ease comparing and storing.

In addition to the C structure M_{C}, CÆSAR also requires the user to provide the following C functions:

- a comparison function taking two module states of type M_{C} as argument, used during state space exploration to decide whether a state has already been encountered (e.g., `memcmp()`, if the structure M_{C} contains no padding)
- a printing function for states of type M_{C}.

5.2. Definition of the LOTOS sort M_{LOTOS}

According to the different ways processes operate on the state of a module M, the LOTOS operations to manipulate values of sort M_{LOTOS} have one of the following three profiles, where each T_i is a type used both in LOTOS and C++, such as `addr_t`, `data_t`, or *Bool*:

- $M_{\text{LOTOS}} \rightarrow T_r$: an operation of this profile corresponds to an access to the state without side-effect. The case where $T_r = \mathit{Bool}$ corresponds to the evaluation of a condition.
- $M_{\text{LOTOS}} \times T_1 \times \cdots \times T_{n\geq 0} \rightarrow M_{\text{LOTOS}}$: an operation of this profile corresponds to a modification of the state of the module as a "side-effect" of executing an instruction, as for instance an assignment. Notice that as in any pure functional language, side effects can only be represented by passing the current state before the call as an additional first parameter and return the state after the call as return value.
- $M_{\text{LOTOS}} \times T_1 \times \cdots \times T_{n\geq 0} \rightarrow M_{\text{LOTOS}} \times T_r$: an operation of this profile corresponds to a modification of the state that also returns a value. As a LOTOS operation can return only one single value, it is necessary to define an additional LOTOS sort for each pair $\langle M_{\text{LOTOS}}, T_r \rangle$, together with operations to access the elements of this pair.

At this stage, we just have to declare these operations in LOTOS; they will be implemented in C++ as described in 5.4 and 5.5.

In the running example, we have to declare the four LOTOS sorts M_{LOTOS}, `data_t`, `addr_t`, and $\langle M_{\text{LOTOS}}, \texttt{addr_t} \rangle$. The implementations of `data_t` and `addr_t` of the original SystemC/TLM model can be reused directly. The sort $\langle M_{\text{LOTOS}}, \texttt{addr_t} \rangle$ is implemented as a C structure with two fields (named `state` and `addr`).

We also have to declare the following operations:

- $M_\texttt{stream_full} : M_{\text{LOTOS}} \rightarrow \mathit{Bool}$ evaluates the C++ condition `stream.full()`.
- $M_\texttt{compute_1} : M_{\text{LOTOS}} \rightarrow \langle M_{\text{LOTOS}}, \texttt{addr_t} \rangle$ simulates the process `compute` up to the transaction.
- $M_\texttt{compute_2} : M_{\text{LOTOS}}, \texttt{data_t} \rightarrow M_{\text{LOTOS}}$ simulates the process `compute` from the transaction to the `wait()` statement.

5.3. Translation of the processes of M

Using the operations of M_{LOTOS}, the SystemC/TLM processes can be translated into LOTOS processes according to the rules described in [3]: each SystemC/TLM process is translated to a LOTOS process; the shared variables and the SystemC events are modeled using additional LOTOS processes; finally, all these LOTOS processes are interconnected using LOTOS parallel composition operators.

In the running example, the process M::compute() is translated into the following LOTOS process:

```
process Compute[TLM_READ,WAIT]
              (state:M_LOTOS) : exit:=
  [M_stream_full(state)]-> exit
  []
  [not(M_stream_full(state))]->
    let x:⟨M_LOTOS, addr_t⟩ =
      M_compute_1(state) in
    TLM_READ !get_addr(x) ?d:data_t;
    let new_state:M_LOTOS =
      M_compute_2(get_state(x),d) in
    WAIT;
    Compute[TLM_READ,WAIT](new_state)
endproc
```

5.4. Split of the methods of M

To reuse the code of M, one must first make a copy of class M by removing all occurrences of SystemC/TLM specific types (such as inheritance from `sc_module` and the port `tlm_port`); we still note M this modified class. Then, to reuse the methods of M when implementing the operations of M_{LOTOS}, we extract corresponding fragment methods from each method of M. Note that the code reused does not contain any SystemC/TLM specific statement.

The modified class M is:

```
class M /*: public sc_module */ {
 /* sc_port<...> tlm_port;*/
 data_t data; ...
 addr_t compute_1() {
  addr+=4; return addr;}
 void compute_2(data_t tmp) {
  data=tmp; stream<<data;}
};
```

5.5. Implementation of the LOTOS operations as C++ functions

The operations of M_{LOTOS} are implemented by C++ functions (declared as C functions to enable linking with the C code generated by CÆSAR), which initialize an object of type M according to the module state M_{C}, call the corresponding fragment method of M, and copy the resulting state of M back into a new value of type M_{C}. Therefore, the user must also provide two functions M_{C}_to_M() and M_to_M_{C}(), which convert between a module state (a value of the structure M_{C}) and a state of the module M.

In the running example, M_compute_1() is implemented as follows:

```
extern M module;
⟨M_C, addr_t⟩ M_compute_1 (M_C mc) {
 ⟨M_C, addr_t⟩ tmp;
 M_C_to_M(&mc, &module);
 tmp.addr = module->compute_1();
 M_to_M_C(&module, &tmp.state);
 return tmp;}
```

6. The BDisp and its SystemC/TLM model

In the rest of this paper, we illustrate our verification approach on the SystemC/TLM model of a graphical processing unit that is used in real circuits.

6.1. The BDisp

The BDisp is an IP (*Intellectual Property*) designed by STMicroelectronics. It is a 2D-graphics co-processor implementing *Blit* (*Block Image Transfer*) and numerous graphical operators, e.g., rotations, alpha blending, or Blue Ray disc decoding. A *job* is a sequence of operations that the BDisp must perform to generate a video stream. Both *real-time* jobs and *non-real-time* jobs are supported.

The BDisp is software-controlled by memory-mapped *registers* and *instruction queues* stored in memory. In order to program a job, the embedded software fills an instruction queue describing the graphical operations to be applied. Next, the embedded software starts the job execution by writing into the BDisp registers; then, it can check the status of a queue by reading the BDisp registers.

There are two kinds of queues: *composition queues* (CQ) for real-time jobs, and *application queues* (AQ) for non-real-time jobs. The BDisp processes one queue at a time, scheduling the queues according to their priorities. The BDisp receives interruptions generated by the *Video Timing Generator* (VTG). These interruptions, together with the information contained in instruction queues, controls the execution pace of the composition queues in order to satisfy real-time constraints. Additionally, the CPU can suspend a queue or change its priority by writing to BDisp registers.

The BDisp provides many types of interruption to manage all situations, but the interruptions generated by the BDisp are not relevant for the properties addressed in this paper.

6.2. The BDisp SystemC/TLM model

STMicroelectronics provided us with a SystemC/TLM model of the BDisp. This model is large: it consists of 22 files, totaling to about 25,000 lines of code.

The BDisp SystemC/TLM model is designed at a level of abstraction tailored to functional validation. The code contains a few `wait` instructions with a duration. These timed `wait` instructions are used to allow the other processes to be executed. However, the durations are not precise enough to allow performance evaluation.

Simulating the BDisp requires several additional models: a model of the CPU, a model of the VTG, a model of the memory, and a model of the bus. The CPU model is a so-called *"native wrapper"*: the embedded code, written in C, is included in the SystemC/TLM code, and the accesses to the BDisp registers are instrumented in such a way that each register access generates a transaction. All these models are quite simple compared to the BDisp model.

The SystemC/TLM module describing the BDisp implements several sorts of connections. Communication between the bus and other components (BDisp, CPU, and memory) is modeled using the TAC TLM protocol designed by STMicroelectronics [19]. Communication between the BDisp and the VTG uses interrupts and is modeled by another TLM protocol (the BDisp module exports one method that is called by the VTG process to notify an interruption).

Internally, the BDisp module contains only one SystemC/TLM process, which is an `SC_THREAD`. That does not mean an absence of concurrency inside the BDisp. Indeed, two external processes, the CPU and the VTG, can simultaneously execute code inside the BDisp module by calling its exported methods.

STMicroelectronics provided us with some test scenarios to validate the BDisp. Each test scenario is composed of a piece of C code to program the BDisp, a memory dump containing the instruction queues, and the expected outputs.

All the tests run well and reveal no error. However, as the OSCI simulator is used, each test is executed with only one single scheduling. Moreover, the simulations are driven by particular timing annotations, which may differ from the real system. Modeling the BDisp in LOTOS will allow us to explore more situations and, possibly, detect more synchronization errors than by using the test scenarios.

7. The BDisp LOTOS model

7.1. Overview

Primarily, our goal was to validate the control part of the BDisp, e.g., all issues related to interprocess synchronization. We were interested neither in validating the sequential graphical operators (CADP is primarily aimed at validating concurrent algorithms) nor the embedded software that controls the BDisp (we have no access to it). Consequently, our LOTOS model describes only the features and the external components that are relevant to the control part of the BDisp.

Following the method presented in section 5, we define the C structure type $BDisp_{\mathrm{C}}$ and the LOTOS sort $BDisp_{\mathrm{LOTOS}}$, and we implement the operations on $BDisp_{\mathrm{LOTOS}}$ using C++ functions extracted from the original SystemC/TLM model. To minimize the size of the type $BDisp_{\mathrm{C}}$, we store only the registers that are meaningful to the control. For some registers, we do not need to store all the bits. For example, in the SystemC/TLM state, the register `BLT_CTL` uses 32 bits, but we are only interested in the 31th bit. Thus, the type $BDisp_{\mathrm{C}}$ is implemented using bit fields:
`struct {int BLT_CTL_31:1; ...}` $BDisp_{\mathrm{C}}$`;`
and the conversion from the LOTOS state to the SystemC/TLM state uses bitwise operations:

```
sc_state.registers[BLT_CTL_ID].value =
                lotos_state.BLT_CTL_31<<31;
```

After these optimizations, the C structure $BDisp_{\mathrm{C}}$ occupies only 52 bytes to store the state of a BDisp with 2 composition queues, 4 application queues, and a maximum number of screen lines reduced to 4; instead, the C++ class `BDisp` uses about 40 kilobytes.

Once the LOTOS sorts are defined, we translate the three SystemC/TLM processes (BDisp, VTG, and CPU) in three concurrent LOTOS processes, called `BDisp_process`, `CPU`, and `VTG`. These three LOTOS processes can read and modify the contents of the BDisp module, which we model as a shared variable of type $BDisp_{\mathrm{LOTOS}}$. In our LOTOS model, this shared variable is encapsulated in a process called `BDisp_state`.

Next, we inline the code of the exported functions in the corresponding initiator process, because we have shown in [15] that transaction inlining reduces the number of states. This means that:

- the code of the BDisp module used to set or get the value of each register is inlined in the `CPU` LOTOS process;
- the code of the BDisp module used to simulate the effect of a video synchronization is inlined in the `VTG` LOTOS process.

The BDisp module contains two SystemC events, which we model using a dedicated LOTOS process called `event_handler`, as described in [3].

The architecture of the LOTOS model is shown on Figure 3. There are five LOTOS processes, three of which correspond to the three SystemC/TLM processes

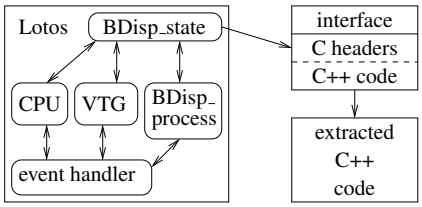

Figure 3. Architecture of the LOTOS model

(`BDisp_process`, `CPU`, and `VTG`). The other two processes are used for communications (`BDisp_state` and `event_handler`).

Our approach allowed us to reuse about 5500 lines of the original SystemC/TLM code (which is, in fact, C++ code), the remaining lines being not related to control. We have written 1000 lines of LOTOS code, and about 2500 lines of C/C++ code for the interface. A complete SystemC/TLM to LOTOS translation, as described in [3], would have required far more work.

7.2. Abstractions and simplifications

The original SystemC/TLM model contains a lot of details that are not relevant for verifying the control part of the BDisp. We present in this section the abstractions and simplifications we applied, still preserving the potential issues.

CPU commands. In the original SystemC/TLM model, sending a command from the CPU to the BDisp is achieved by a TLM write transaction Tr. The address of Tr is obtained by adding the register offset to the base address of the BDisp. The command arguments are encoded in the 32-bit data field of Tr. As we are not interested in this encoding mechanism, we represent all possible BDisp commands by a new LOTOS type `CPU_command`. This brings three benefits:

- this new type requires fewer bits than a numeric offset;
- we can iterate efficiently over the registers (this is used in the CPU to generate a command nondeterministically);
- no model of the bus is required.

Instruction nodes. Each instruction node is significantly reduced by removing anything unrelated to control. For an instruction node of a composition queue CQn, we keep only two Booleans and one screen line number (reduced to the range $0 \ldots 3$). For an instruction node of an application queue AQn, we need only two Booleans. Thus, the instruction nodes are small enough to be enumerated efficiently.

Instruction queues. Once an instruction queue is triggered, the BDisp reads an instruction node from the memory. Since we focus on the verification of the BDisp, we assume that the embedded software controlling the BDisp has correctly written the instructions to the memory before triggering the queue. Thus, we can use lazy evaluation to model the instruction queues. Instead of generating the contents of an instruction queue when it is filled in by the CPU, we wait until the BDisp reads it. Each time a new instruction node is read by the BDisp, we generate its contents nondeterministically. The benefits are twofold:

- we can remove the code that writes instruction nodes to the memory;
- we store at most one node per instruction queue, reducing the memory size of a state.

This abstraction may prevent detection of bugs due to concurrent accesses to a node, but such bugs are possible only with erroneous embedded software, and, as said above, we assume that this software is correct.

Boolean abstraction of counters. After a few experiments, we noticed that two integer variables could increase infinitely, leading to an infinite state space. After examining the code and the test scenarios provided by STMicroelectronics, we concluded that we could safely assume that these variables take only two values, and thus be abstracted by booleans. To check the correctness of this abstraction, we applied the same modification to the original SystemC/TLM model, and we ran the test scenarios provided by STMicroelectronics again. The simulation results are identical, which confirms the correctness of our abstraction.

7.3. Accessing the BDisp state

The contents of the BDisp module are stored in the variable `state` of the process `BDisp_state`. The variable `state` is of sort $BDisp_{\mathrm{LOTOS}}$, which is implemented using the C type $BDisp_{\mathrm{C}}$. Many functions access or modify this variable. For example, the function `ExeSetCommand_CIF` simulates the effect of a CPU command (i.e., a write to a BDisp register); this function takes a BDisp state and a CPU command as arguments, and returns the updated BDisp state.

To model a variable such as `state`, which is shared between several processes, the usual solution in process algebra is based on a READ/WRITE mechanism: the LOTOS process `BDisp_state` storing the variable value has two gates READ and WRITE, used to send the current value and get the new value.

Here is how the CPU can write to a BDisp register using the READ/WRITE mechanism:

```
READ ?state:BDisp_LOTOS
let new_state:BDisp_LOTOS =
   ExeSetCommand_CIF(state,command) in
WRITE !new_state;
```

First, the `BDisp_state` process sends its current state to the CPU process. Next, the CPU process executes the command. Finally, the CPU process sends back the new state to the `BDisp_state` process.

When using the READ/WRITE mechanism, the code of the `BDisp_state` process is quite simple. Moreover, it is easy to implement a lock to prevent concurrent accesses to the BDisp state: a READ rendezvous locks the state, a WRITE rendezvous unlocks it.

However, the BDisp case study revealed a drawback of the READ/WRITE mechanism. The BDisp state is sent to three processes (`CPU`, `VTG`, and `BDisp_process`). Eventually, many processes contain one or several local variables of sort $BDisp_{\mathrm{LOTOS}}$. In a first version of the BDisp LOTOS model, there were 17 such variables, using 52 bytes of memory each, leading to excessive memory consumption for states of the LOTOS model (about 1 kilobyte per state).

We therefore designed a different mechanism, named EXECUTE/RETURN, to avoid this duplication of BDisp state local copies. In this approach, the gates WRITE and READ are replaced by the gates EXECUTE and RETURN. The initiator process (e.g., the CPU) sends an opcode to the `BDisp_state` process using the EXECUTE gate. The `BDisp_state` process decodes the opcode and executes the corresponding function. If the function returns some values (in addition of the new BDisp state), these values are send back to the initiator process using the RETURN gate. The lock mechanism is implemented by adding two boolean offers to each EXECUTE or RETURN communication: the first boolean specifies whether to take the lock or not, the second whether to release it or not.

This EXECUTE/RETURN mechanism requires more LOTOS code than the usual READ/WRITE mechanism, because a new type has to be implemented to store the opcodes, and because the `BDisp_state` process contains specific code for each function of the $BDisp_{\mathrm{LOTOS}}$ sort. But the EXECUTE/RETURN mechanism is very profitable in presence of large shared variables. Indeed, we reduced the state size for the whole LOTOS model from one kilobyte down to 104 bytes, i.e., by an order of magnitude.

However, the EXECUTE/RETURN mechanism might increase the number of transitions in the LTS. Suppose that a process wants to execute an atomic sequence of operations $o_1, \ldots, o_n$ on the BDisp state. Using the classic mechanism, a single READ and a single WRITE transitions are enough. Instead, the EXECUTE/RETURN mechanism requires, for each operation o_i, one EXECUTE transition and possibly one RETURN transition. To avoid this increase of the LTS size, it is possible to aggregate the atomic sequence $o_1, \ldots, o_n$ in a single new operation o'. Thus, only one EXECUTE transition is required, and one RETURN transition if the sequence o' returns some values. Aggregation works as long as the operations of an atomic sequence are not separated by other LOTOS transitions; in particular, this approach worked well on the BDisp case study.

8. Experiments and results

8.1. Interactive and random simulation

The LOTOS model of the BDisp were simulated using the *Open/Cæsar Interactive Simulator* (OCIS). Compared to simulating the SystemC/TLM model with the simulator provided by the OSCI, using OCIS offers two benefits:

- the simulation is interactive: the user can choose the CPU commands, the contents of the instruction nodes, and how the processes are scheduled
- the user can freely backtrack to any previous state, because OCIS stores the tree of explored states.

Another possibility to simulate the LOTOS model would be to use the random simulator tool *Executor* of CADP. Contrary to the OSCI SystemC simulator, Executor chooses the scheduling nondeterministically.

8.2. Generation of the full LTS

Beyond simulation, we aimed at formal verification by generating the labeled transition system (LTS) corresponding to our LOTOS model of the BDisp.

Early attempts indicated that the LTS of this LOTOS model is too large to be generated. On a 64 bit Linux machine (2 GHz Opteron), we interrupted the generation after 3 hours and 20 minutes (at this point, the generation required 15.7 gigabytes of memory). The graph obtained contained 155,377,179 states and 371,146,000 transitions. It is likely that the full graph is many times bigger.

To fight the state explosion, we applied the compositional verification techniques offered by CADP. We could generate the LTS of the CPU (45 states after reduction), the LTS of the VTG (21 states), and the LTS of the BDisp process (124 states). However, we could not generate the LTS of the `BDisp_state` process, which may be as big as the LTS of the full system. Consequently we could not use compositional verification.

8.3. Model checking of verification scenarios

Because we could not generate the full LTS, we decided to concentrate on verification scenarios, that restrict the inputs to some of their most realistic values. For the BDisp

	LTS generation		LTS reduction		property evaluation				
	LTS size	time	LTS size	time	P1	P2	P3	P4	P5
AQ1	100 states	3.3 s	70 states	<1 s	T	T	F	T	F
AQ2	4,077 states	3.4 s	780 states	<1 s	T	T	F	T	F
AQ3	20,459 states	3.5 s	2,005 states	<1 s	T	T	F	T	F
AQ4	73,225 states	4.0 s	3,743 states	<1 s	T	T	F	T	F
CQ1AQ0	11,322 states	3.5 s	1,303 states	<1 s	T	F	T	T	T
CQ1AQ1	410,514 states	7.4 s	15,647 states	17 s	T	F	F	T	T
CQ1AQ2	1,587,961 states	26 s	36,266 states	91 s	T	F	F	T	T
CQ1AQ3	5,153,484 states	114 s	63,486 states	410 s	T	F	F	T	T
CQ1AQ4	15,909,887 states	620 s	97,307 states	1880 s	T	F	F	T	T
CQ1AQ1+RESET	1,660,845 states	28 s	56,051 states	128 s	F	F	F	T	T

Table 1. Results of the experiments

case study, a verification scenario is mainly a sequence of CPU commands that do something useful, like triggering the queues, whereas the control information of the instruction nodes is chosen nondeterministically.

- scenarios "AQn", with $n \in [1..4]$: we trigger n application queues ($n \times 3$ CPU commands), the VTG is disabled since it has no effect on the application queues;
- scenarios "CQ1AQn", with $n \in [0..4]$: we trigger 1 composition queue and n application queues ($n \times 3 + 3$ CPU commands), the VTG is enabled;
- scenario "CQ1AQ1+RESET": we add a soft reset to the scenario CQ1AQ1 (6+1 CPU commands).

For each of these ten scenarios, we generated the LTS (see Table 1) and minimized for branching bisimulation (using the bcg_min tool of CADP).

We expressed five properties using regular alternation-free mu-calculus formulas [20], and evaluated them on the minimized LTS corresponding to the verification scenarios, using the Evaluator 3.5 model checker of CADP.

- P1: if a composition queue is triggered before any application queue, and if the composition queue is not suspended, then no application queue can be executed.
- P2: if a composition queue is triggered but not disabled, then there exists always a path to execute it.
- P3: if an application queue is triggered but not disabled, then there exists always a path to execute it.
- P4: if the CPU sends a RESET command to the BDisp, then the BDisp can still read one node but not two.
- P5: property expressing the absence of deadlock.

The property evaluation results are given in Table 1. The evaluation of all the properties for all the reduced LTS took about 45 seconds. Most of the results were as expected, in particular:

- The scenarios AQn do not satisfy property P5 (no deadlock), because the BDisp stops normally when all application queues are completely executed, and the VTG is disabled.
- The property P1 is false when a reset is possible. Indeed, if the reset occurs after CQ1 triggering but before AQ1 triggering, then AQ1 is executed first.

However, one would have expected properties P2 and P3 to be satisfied by all scenarios AQn and CQ1AQn, but for instance P2 does not hold for scenario CQ1AQ1. The diagnostic generated by Evaluator exhibited a scheduling leading to a wrong behavior of the BDisp. Indeed, if the CPU is scheduled too often before the BDisp, the latter may miss an event notification, and thus not execute properly. Using an interactive SystemC scheduler, we noticed that this scheduling is reproducible on the *untimed* version of the SystemC/TLM model (i.e., replacing delayed notifications by immediate notifications), but not on the *timed* version (nor on the actual circuit itself). We proposed a modification of the SystemC/TLM model so that the property holds also in an untimed context.

9. Conclusion and future work

We presented an approach to apply model checking to SystemC/TLM models. In our approach a large part of the C++ code present in these models is reused, while the remaining part of the SystemC/TLM code is translated into LOTOS according to the systematic rules of [3]. Contrary to other approaches, our approach avoids the complete translation of the SystemC/TLM model into the particular input language of the model checker considered. Thus, formal verification is better integrated in the design flow, the translation effort is reduced, and the confidence in the results is increased.

We applied this approach to a large industrial case study, namely the BDisp designed by STMicroelectronics. Although we were not able to explore the entire state space of the whole system, we managed to analyze large subsets of it and to prove several correctness properties on realistic verification scenarios. Using the verification tools of CADP [4], we discovered an issue that makes the original timed SystemC/TLM model incompatible with a semantics for untimed TLM models (e.g., the one of [3]).

As regards future work, a first line of research would be to elaborate guidelines to write SystemC/TLM models so as to reduce the amount of work required to obtain a verifiable LOTOS model. For instance, to enable partial order reductions and compositional verification of the LOTOS model, it seems that the SystemC/TLM models should be split into many small processes rather than a big monolithic one.

As a second line of research, we are currently developing a LOTOS/SystemC library that helps reusing even more of the original C++ code, including SystemC/TLM specific code.

Acknowledgments.
We are grateful to L. Ducousso, D. Hermitte, and R. Hersemeule (STMicroelectronics) for providing us with the BDisp case study and for their help with this work.

References

[1] *SystemC v2.2.0 Language Reference Manual (IEEE Std 1666-2005)*, OSCI, http://www.systemc.org/.

[2] F. Ghenassia, Ed., *Transaction-Level Modeling with SystemC. TLM Concepts and Applications for Embedded Systems*. Springer, June 2005.

[3] O. Ponsini and W. Serwe, "A schedulerless semantics of TLM models written in SystemC via translation into LOTOS," in *FM'08*, LNCS 5014. Springer, 2008.

[4] H. Garavel, F. Lang, R. Mateescu, and W. Serwe, "CADP 2006: A toolbox for the construction and analysis of distributed processes," in *CAV'2007*, LNCS 4590, pp. 158–163. Springer, Jul. 2007.

[5] C. Helmstetter, F. Maraninchi, L. Maillet-Contoz, and M. Moy, "Automatic generation of schedulings for improving the test coverage of systems-on-a-chip," *FMCAD*, pp. 171–178, 2006.

[6] S. Kundu, M. Ganai, and R. Gupta, "Partial order reduction for scalable testing of SystemC TLM designs," in *DAC'08*, pp. 936–941. ACM, 2008.

[7] N. Blanc and D. Kroening, "Race analysis for SystemC using model checking," in *ICCAD 2008*, pp. 356–363. IEEE, 2008.

[8] C. Helmstetter, F. Maraninchi, and L. Maillet-Contoz, "Test coverage for loose timing annotations," in *FMICS'06*. Springer, August 2006.

[9] R. Drechsler and D. Große, "Reachability analysis for formal verification of systemc." in *DSD*, pp. 337–340. IEEE Computer Society, 2002.

[10] D. Große and R. Drechsler, "CheckSyC: an efficient property checker for RTL SystemC designs," in *ISCAS*, vol. 4, pp. 4167–4170, May 2005.

[11] B. Niemann and C. Haubelt, "Formalizing TLM with communicating state machines," in *FDL'06*, pp. 285–292, September 2006.

[12] D. Kroening and N. Sharygina, "Formal verification of SystemC by automatic hardware/software partitioning," in *MEMOCODE'05*, pp. 101–110. IEEE, 2005.

[13] M. Moy, F. Maraninchi, and L. Maillet-Contoz, "LusSy: an open tool for the analysis of systems-on-a-chip at the transaction level," *Design Automation for Embedded Systems*, 2006.

[14] C. Traulsen, J. Cornet, M. Moy, and F. Maraninchi, "A SystemC/TLM semantics in Promela and its possible applications," in *SPIN Workshop*, LNCS, July 2007.

[15] C. Helmstetter and O. Ponsini, "A comparison of two SystemC/TLM semantics for formal verification," in *MEMOCODE'08*, Jun. 2008.

[16] ISO/IEC, "LOTOS — a formal description technique based on the temporal ordering of observational behaviour", Genève, International Standard 8807, 1989.

[17] T. Bolognesi and E. Brinksma, "Introduction to the ISO specification language LOTOS," *ISDN* 14(1), pp. 25–59, 1988.

[18] H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specification 1 — Equations and Initial Semantics*, ser. EATCS Monographs on Theoretical Computer Science, vol. 6. Springer, 1985.

[19] STMicroelectronics - SPG, "TAC Package," 2004, http://www.greensocs.com/projects/TACPackage.

[20] R. Mateescu and D. Thivolle, "A model checking language for concurrent value-passing systems," in *FM'08*, LNCS 5014, pp. 148–164. Springer, 2008.