Unité Mixte de Recherche 5104 CNRS - INPG - UJF
Centre Equation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
http://www-verimag.imag.fr

Formal and Executable Contracts for Transaction-Level Modeling in SystemC

Tayeb Bouhadiba,Florence Maraninchi, Giovanni Funchal

Verimag Research Report n^o TR-2009-7

May 2009

Formal and Executable Contracts for Transaction-Level Modeling in SystemC

Tayeb Bouhadiba,Florence Maraninchi, Giovanni Funchal

May 2009

Abstract

Transaction-Level Modeling (TLM) for systems-on-a-chip (SoCs) has become a standard in the industry, using SystemC. With SystemC/TLM, it is possible to develop an executable virtual prototype of a hardware platform, so that software developers can start writing code long before the actual chip is available. A hardware model in SystemC/TLM is very abstract, compared to the detailed RTL model. It is clearly component-based, with guidelines defining how components should be designed for use in any TLM context. However, these guidelines are quite informal for the moment. In this paper, we establish a structural correspondence between SystemC/TLM and a formal component-model for embedded systems called 42, for which we have defined a notion of *control contract*, and an execution mode for systems made of components' contracts. This is a way of formalizating SystemC/TLM principles. Moreover, it allows the combined use of SystemC/TLM components with 42 components. Demonstrating that such a combined use is possible is key to the adoption of formal components' definitions in the community of TLM users.

Keywords: Transaction-Level-Modeling, Formal Component Models, Systems-on-a-Chip, Virtual Prototyping

Reviewers: Nicolas Halbwachs

Notes:

How to cite this report:

@techreport { ,
title = { Formal and Executable Contracts for Transaction-Level Modeling in SystemC},
authors = { Tayeb Bouhadiba,Florence Maraninchi, Giovanni Funchal},
institution = { Verimag Research Report },
number = {TR-2009-7},
year = { },
note = { }
}

1 Introduction

The Register Transfer Level (RTL) used to be the entry point of the design flow of hardware systems, including systems-on-a-chip (SoCs). However, the simulation environments for such models do not scale up well. Developing and debugging embedded software for these low level models before getting the physical chip from the factory is no longer possible at a reasonable cost. New abstraction levels, such as the *Transaction Level Model (TLM)* [7], have emerged. The TLM approach uses a component-based approach, in which hardware blocks are described by *modules* communicating with so-called *transactions*. The TLM models are used for early development of the embedded software, because the high level of abstraction allows a fast simulation. This new abstraction level requires that SoCs be described in some non-deterministic asynchronous way, with new synchronization mechanisms, quite different from the implicit synchronization of synchronous circuit descriptions.

SystemC is a C++ library used for the description of SoCs at different levels of abstraction, from cycle accurate to purely functional models. It comes with a simulation environment, and has become a *de facto* standard. SystemC offers a set of primitives for the description of parallel activities representing the physical parallelism of the hardware blocks. The TLM level of abstraction is implemented in SystemC by defining specific TLM libraries and templates. TLM 2.0 has been standardized by the Open SystemC Initiative (OSCI).

The success of SystemC comes from its C/C++ part: it is widely known, it is a general-purpose programming language, so there are no restrictions on what the designer can write, and it makes it possible to build tools for the co-simulation of SystemC code with C, assembly code, RTL models, etc. The dark side of the picture is that SystemC models are quite hard to analyze formally (SystemC being based on C++ has no formal semantics, and SystemC models may implement relatively simple things by complex integer manipulations, which make simple properties undecidable in the general case). Therefore the main activity with a SystemC model is simulation.

On the other hand, in the embedded systems community, there is a lot of work on formal models of components, some of them equipped with contracts, and amenable to formal verification or enhanced simulation techniques like run-time verification. But these models rely on dedicated formal languages (to ease the connection to verification tools) and ignore the quite dirty parts of real-life SystemC models (like the interface with RTL models, the wrappers that are needed for the embedded software to execute exactly the same on the virtual prototype of the hardware, or on the final chip, the low-level code needed for Hardware-in-the-Loop simulations, etc.). Therefore they are far from being usable in practice for industrial-size SoC descriptions (which can be 100 SystemC kLoc, and require the power of a general-purpose language).

1.1 Related Work

There has been a lot of work on the formalization of SystemC, as a parallel programming language. The idea is to extract formal models from SystemC programs, so that tools like model-checkers can be applied. In [8], the formalization uses timed automata, with a connection to Uppaal [10]. In [16, 13] we describe several formalizations and connections to SMV [14], SPIN [9], etc. However, all these formalizations include some form of a *global* model for the SystemC scheduler, and the TLM principles in SystemC are not formalized, in the sense that there is no clear definition of what a *TLM component* is. There only exist *guidelines* for the developers of TL models.

Related work should also be searched for in TLM design for SoCs, not necessarily in SystemC. This could help in identifying the principles of TLM, independently of SystemC. Unfortunately, the SystemJava attempt seems to have disappeared; SystemVerilog is a modeling language for heterogeneous systems, at various levels of abstraction, but it does not come with a methodology for TLM design. Tools like Ptolemy or Metropolis can be used to describe SoCs, at the same level of abstraction than TLM, but do not help in reusing SystemC existing modules, which is our primary goal.

1.2 Contributions and Structure of the Paper

In this paper, we investigate the combined use of: (i) standard SystemC/TLM models; (ii) a formal component model for embedded systems called 42 [11], inspired a lot by Ptolemy [4]. The idea is to take advantage of the formal aspects of 42, while retaining the advantages of existing SystemC components.

Both SystemC/TLM and 42 describe systems of components representing parallel entities, and animated by a scheduler (in SystemC) or a so-called *controller* (in 42). The SystemC scheduler is specific, while the 42 notion of a controller is *generic*; consequently, the effect of the SystemC scheduler can be mimicked by a dedicated 42 controller. The core of the paper is the description of the structural correspondence between a SystemC/TLM model, and a 42 model that has two properties: (i) it has exactly the same component structure; (ii) animating it with a dedicated 42 controller produces the same observations as the SystemC standard executions. For the latter point, we will recall and extend the *control contract execution mode* of 42 as defined in [2]. To our knowledge, this is the first formalization of standard SystemC/TLM components, i.e., we do not only formalize the SystemC language, we also formalize what a TLM component is (interface, contract, and the model of computation and communication corresponding to TLM). Moreover the formalization can be used without leaving the SystemC world, which is key to the adoption of formal models in the SystemC/TLM community.

This structural correspondence has several applications. First, we can extract 42 *control contracts* from the components of a SystemC/TLM model, and animate them by a 42 controller; this produces a superset of the SystemC observations, because extracting control contracts abstracts away the behavior of all the data; this allows to reason on the synchronizations only. Second, we can extract the contracts from some of the components of a SystemC/TLM model, and provide hand-written 42 contracts for the other components; all of them can then be animated, and the synchronization validated before the entire SystemC code is available. Finally, once the SystemC/TLM code is provided for the missing components, the set of contracts can still be animated in 42, and executed together with the actual SystemC code; this allows to check dynamically the conformance between a component and its contract, and to observe more errors.

The contributions of this paper are: (i) A structural correspondence between a SystemC/TLM model and a 42 model; (ii) A generalized execution mode for systems of 42 components given by their contracts only (in [2] we had only one level of hierarchy); (iii) A method for executing the 42 model together with the real SystemC code (the SystemC code for threads and functions is unchanged, but it is animated by the 42 controllers instead of the SystemC scheduler).

Section 2 presents the principles of transaction-level-modeling (TLM) in SystemC; Section 3 is an overview of the component model 42; Section 4 establishes the correspondence between a SystemC/TLM model and a 42 model; Section 5 presents typical uses of our approach, on a case-study; Section 6 is the conclusion.

2 TLM in SystemC

We explain briefly the notion of transaction-level modeling, and the way it is implemented in SystemC. SystemC [17] is a C++ library, plus a simulation environment based on a non-preemptive scheduler. A TL model is based on an *architecture*, *i.e.*, a set of components that expose *ports*, and connections between them, as shown on Fig. 1. The components, also called *modules*, represent some physical entities that behave *in parallel* in the real system to be modeled (typically: a bus, a CPU, a memory, etc.). The direction of the arrow shown on a port determines the role of the port. On Figure 1, `p2` in `module1`, and `p1` in `module2`, are *initiator* ports; `p1` in `module1`, and `p2` in `module2`, are *target* ports.

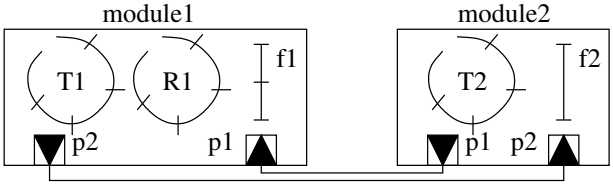

Figure 1: Example of SystemC/TLM (architecture)

The behavior of a component is given by a set of *threads* (represented by circles with "steps" on Fig. 1), and a set of *functions* (represented by straight lines with steps), both programmed in full C++ (see Figure 2). `module1` has two threads `T1` and `R1`, and one function `f1`, while `module2` has a single thread `T2` and a function `f2`. The threads are *active* code, to be scheduled by the global scheduler; the functions are *passive* code, offered to the other components, and that will be called from a thread of another component; the functions are attached to the target ports of the module (`f1` is attached to `p1` in `module1`). Inside such a module, the processes and the functions may share *events* in order to synchronize with each other. Events can be notified, or waited for. In `module1` (Fig. 2) there are three events `e1, e2, e3`.

All the threads are managed globally. The SystemC scheduler is non-preemptive: a running thread has to yield, by performing a wait on an event (or on *time*, see comments in the conclusion, but we will use untimed models throughout this paper, for sake of simplicity). For instance, for the thread `T1` of `module1`, the only point where the thread yields is `wait(e1)`. The execution of `a++; e2.notify(); a++; e3.notify();` is therefore atomic.

```
 1: void module1::T1(){
 2:  int a = 0;
 3:  while(true){
 4:   wait(e1);
 5:   a++;e2.notify();
 6:   a++;e3.notify();
 7: }}
 8: void module1::R1(){
 9:  int b = 0;
10:  while(true){
11:    b++;wait(e2);
12:    b++;p2.f2(b);
13: }}
14: void module1::
15:          f1(int x){
16:   cout << x ;
17:   e1.notify();
18:   wait(e3);
19: }// ** module1 **
```

```
30: void module2::T2(){
31:  int c;
32:  while(true){
33:   c++;  p1.f1(c);
34:   c++;  wait(e4);
35: }
36:}
37:
38: void module2::
39:          f2(int x){
40:   cout<< x ;
41:   e4.notify();
42: }// ** module2 **
```

Figure 2: Example of SystemC/TLM (code)

According to the TLM guidelines, communications between modules cannot use the event mechanism, because this would be meaningless w.r.t. the physical parallelism to be modeled. The only possible communications are called *transactions*, implemented by blocking function calls; the link between a caller and the callee is established through the architecture. On Figure 2, the thread `T2` of `module2` initiates a transaction on its port `p1` (written `p1.f1(c)`). This is a call to the function `f1` in `module1` (which is attached to the target port `p1` of `module1`), because the initiator port `p1` of `module2` is connected to the target port `p1` of `module1`. When the call is executed (`T2` being running), the control flow is transferred to `module1`, until `f1` terminates; then the control flow returns to `module2`, and the execution continues until the next yielding point (`wait(e4)` in the example). Since function `f1` in `module1` waits for the event `e3`, it yields if `e3` is not present; this means that thread `T2` in `module2` may yield because of a `wait` statement in the function it has called in another module. An example atomic sequence is the following: the scheduler elects thread `R1`, which is at line 12; it executes `b++` and then calls `f2` via the port `p2`; the body of `f2` in `module2` is executed entirely, and the control returns to `module1`; the thread `R1` loops, executes `b++` at line 11, and stops on `wait(e2)` at line 11.

3 The Component Model 42

The original definition of 42, and various examples (synchronous, asynchronous, GALS), can be found in [11, 12]. 42 enriched by control contracts, used to simulate a system of components given by their

contracts, can be found in [2].

3.1 Basic and Assembled Components

Figure 3 shows a 42 component. It is a black-box that has input and output *data* ports, and input and output *control* ports. The input control ports are used to ask it to perform a computation *step*. A step corresponds to a terminating (non-necessarily deterministic) piece of code. A component has some internal memory. The input and output data ports are used to communicate data between the components. The output control ports will be used by the components to send information to the controller (see below). The output control ports have enumerated types, indicated on the side of the picture.

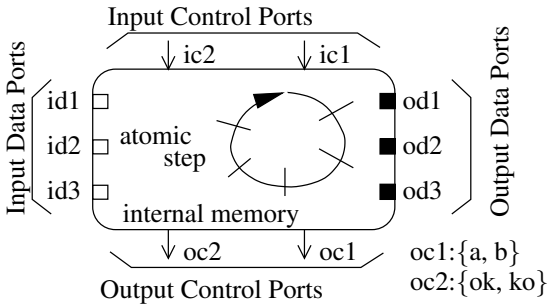

Figure 3: A Basic 42 Component

Components are connected by directed *wires* (see Figure 4). An input data port can be connected to an output data port of the same type (we will assume this is always true in the sequel). When the input and output ports connected together have the same name, we write the name on the wire. The control ports are connected to the *controller*, not directly to each other. In the sense of Ptolemy [4], a 42 *system* (see Fig. 4) is made of components connected by wires (the *architecture*) plus a *controller* that activates the components and decides what happens on the wires. The model is hierarchic: an architecture plus a controller form a new *component*. It exposes new input and output data and control ports. The global output data ports (e.g., O) are connected to output data ports of sub-components (e.g., $B.o$), and the global input data ports (e.g., I) are connected to input data ports of the sub-components (e.g., $A.i$).

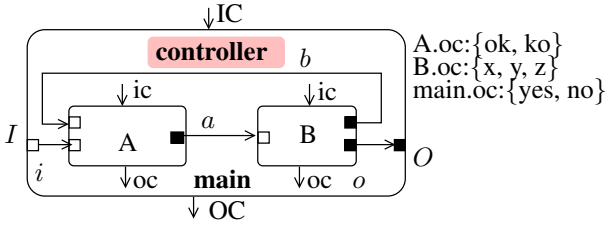

Figure 4: Assembling Components

3.2 Controllers

Figure 5 is an example controller for the system of Figure 4, given in some simple imperative style. The controller defines how the components behave together. It is in charge of *translating* an activation request on one global control input port (e.g., `IC`, also referred to as a *macro-step* in the sequel), into a sequence of activations of the sub-components, and data exchanges between them (also called *micro-steps*). It defines what the *Model of Computation and Communication (MoCC)* is, at this level. To achieve this, the controller may use some temporary variables explicitly associated with the wires (e.g., `m_a`), whose lifetime is limited to the macro-step. It uses simple primitives of two forms: activation of sub-components (e.g., `A.ic`) and data management: `a.put` moves the value from the port `A.a` to the memory associated with the wire a, whereas `a.get` moves the value stored from the wire to the right port `B.a`. The controller also reports on

the activity of sub-components through global output control ports (e.g., `OC`). As in Ptolemy, heterogeneous systems are build by using a hierarchy of controllers that implement different MoCCs.

```
1 Controller is
2 x : int := 1; alpha: { ok, ko }
3 for IC do  {                              //Global IC
4 m_a, m_b, m_i, m_o : int;
5  if (x==2){
6     i.put; i.get;                         //Global ID
7     A.ic; alpha:=A.oc;                    //Read A.oc
8     if(alpha==ok){ a.put; a.get; B.ic;}   //Activate B
9     x:=1;
10   }else{
11    B.ic; o.put; o.get;                   //Global OD
12    b.put; b.get; A.ic;  x:=2;}
13  main.OC := "yes";                       //Global OC
14 }
```

Figure 5: Example 42 Controller

3.3 Control Contracts

The distinction between control and data ports makes it easy to define rich *control contracts* for the 42 components. Such contracts specify two things: (i) the language of correct sequences of activations (input control ports) for the component; this is given as an explicit automaton in the sequel; (ii) for each activation, which data inputs are required, and which control and data outputs are produced. Figure 6 shows a contract for the component C of Figure 3. Each transition has a label of the form:

`{data req} control input / control outputs {data prod}`

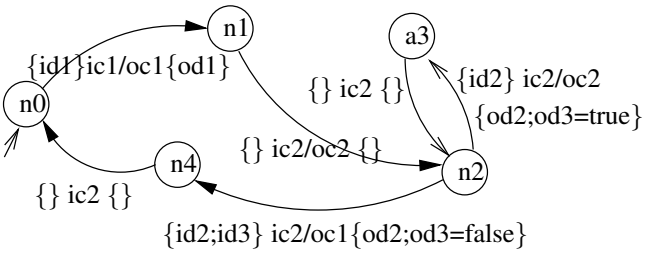

Figure 6: Contract for the Component of Fig. 3

part expresses data dependencies; for instance `{id2;id3}` means that the transition needs `id2` and `id3`. The part `{data prod}` is built similarly, and expresses which outputs are indeed produced. For some of the data ports, which are used for synchronizations (e.g., an interrupt) the contract may also specify which value is indeed produced (e.g., `od3 = true`). The `control input` part is a single control input. The `control outputs` part gives the control outputs that are indeed produced. Initially, the contract is in its initial state (`n0`), and then it evolves according to the sequence of activations produced by the controller in response to a sequence of macro-steps. Each macro-step is considered to start in the state where the contract was, at the end of the previous macro-step. In [2] we presented a more general version of contracts, with internal variables and conditional data dependencies, but this simple version is sufficient here.

3.4 Executing Contracts

Observe Figure 7. It is a 42 model of an asynchronous system for which we want to perform simulations. The control ports `op` are introduced in order to define an *asynchronous simulation controller*. Each component has a single control input `op`, corresponding to the execution of a piece of behavior that can be considered as *atomic* with respect to the other components.

The asynchronous simulation consists in repeatedly activating the global system via its control input `op`, which is then non-deterministically translated into `A.op` or `B.op`, respecting the data dependencies (a component whose contract specifies that input data `id` is needed, cannot be activated if `id` is not available, i.e., has not been produced by another component). We could write this controller directly, but in [2], we

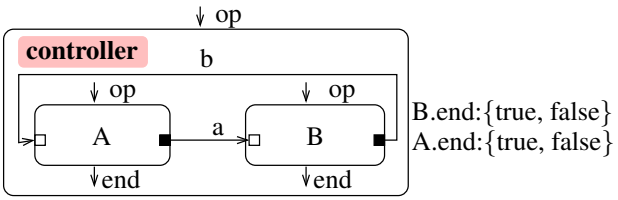

Figure 7: Asynchronous simulation

described how to obtain it automatically from the components' contracts. Figure 8 gives possible contracts for the two components. The controller is simply a contract *interpreter*, which produces one trace in the asynchronous product of the components' contracts. Figure 9 describes part of this asynchronous product.

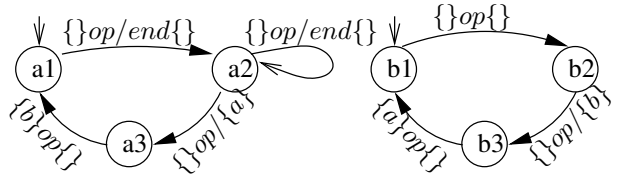

Figure 8: Contracts for the components of Fig. 7

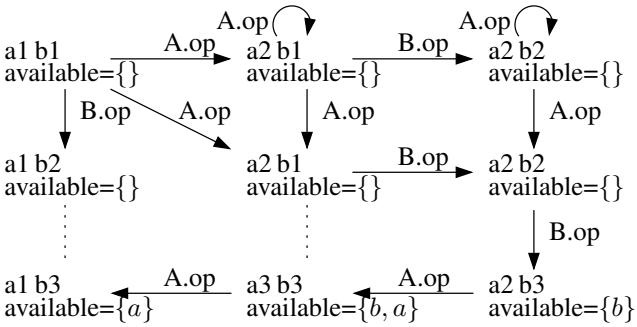

Figure 9: Example Executions

Each location shows: the states of the components' contracts, and the availability of the data exchanged between components. For each global `op`, the controller is in a particular location of this graph, chooses randomly one of the outgoing transitions (say, t), executes it, updates the set of available data (removing what t needs, and adding what t provides), and reaches a new location.

4 The Correspondence Between 42 and SystemC/TLM

In this section, we use a toy example to establish the structural correspondence between the TLM components, as defined informally by the TLM guidelines, and the 42 components. The interesting uses of this approach are explained in section 5, together with the case-study. The SystemC example (denoted by SCTLM in the sequel) is that of Figures 1 and 2. The corresponding 42 model will be denoted by 42M in the sequel. Its architecture is given by Figure 10; example contracts for its components are given by Figure 11. For sake of clarity, we will suppose that we cannot have simultaneously two active calls of a given function. Our full encoding of SystemC into 42 forbids recursion and relies on code duplication when a function may have several parallel callers.

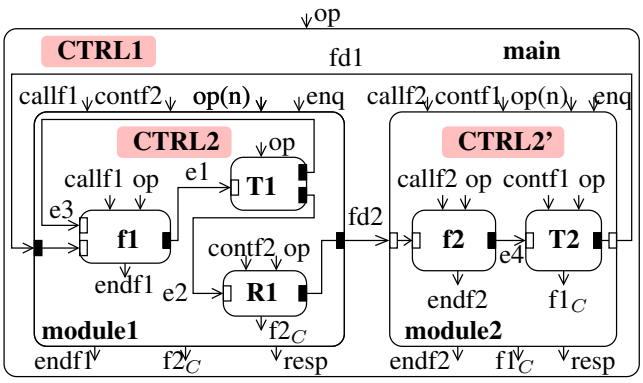

Figure 10: 42 Architecture for the system of Figs. 1 and 2

4.1 Architecture and Ports

The SystemC model SCTLM is made of several *modules* that have *ports*; each module contains threads and functions. The 42 model 42M is hierarchic; at each level of the hierarchy, it is made of a set of components, and a controller. The architecture of 42M (Fig. 10) is built by using one 42 component per module in SCTLM, at the highest level of hierarchy; moreover each 42 component corresponding to a module M is itself built as a 42 system, with one component per thread of M, and one component per function of M; The main component only has a control input `op` for asynchronous simulation.

4.1.1 Highest Level

The module ports in SCTLM are used to route function calls. At the highest level of the 42 hierarchy, these function calls of SCTLM are encoded by separating the *control* effect, and the *data* exchanged. For instance, the fact that thread `T2` in `module2` may call `f1` of `module1` via the ports `p1` (Fig. 2), corresponds to: (i) the data wire `fd1` from `module2` to `module1` on Figure 10; (ii) the output control port $\texttt{f1}_C$ (for "*f1 is called*") of `module2`; (iii) the input control port `callf1` of `module1` (to activate the code of `f1`); (iv) the output control port $\texttt{endf}_1$ (for "*f1 has finished*") of `module1`; (v) the input control port `contf1` of `module2` (to continue the execution of the thread that has called `f1`, when `f1` is finished).

Each module has output data ports corresponding to the parameters of the functions that can be called from inside the module, and input data ports corresponding to the parameters of the functions it provides (we assume a *union* type for these ports, representing all the functions parameters). It also has one input control $\texttt{callf}_i$ for each function it provides, and one control input $\texttt{contf}_i$ for each function that may be called from it. Each module also has a control input `enq` (for "*enquire*") and a control output `resp` (for "*response*"). The use of all these ports by CTRL1 is further detailed below. Finally each module has a *parametrized* control input `op` for asynchronous simulation; the integer parameter will be used, in conjunction with the `enq`/`resp` mechanism, in order for CTRL1 to choose a transition in a module.

4.1.2 Deepest Level

At the deepest level, the 42 components represent threads and functions that may communicate through events. The events are encoded as data wires[1]. A thread or function that may notify (resp. wait for) the event `e` has an output (resp. input) data port `e`. The controllers CTRL2 and CTRL2' ensure the connection between the control ports of the modules, and the control ports of the functions and threads.

Each component associated with a function has an input data port for the parameters (connected to the corresponding input data port of the module); it also has a control input $\texttt{callf}_i$ (used to start the function), an a control output $\texttt{endf}_i$ (produced when the function terminates); finally the control input `op` is used

[1] This produces the effect of *persistent* events, in the sense that an event is memorized until "consumed" by another thread executing a `wait`); in SystemC, standard events are *not* persistent: they are lost if no eligible process waits for them. We could encode non-persistent events in 42 as well.

to re-activate the function code, when it had stopped on a `wait`, and still has something to do before it terminates. If the function calls another function, the component also has the ports related to functions, explained below for threads.

Each component associated with a thread has a control input `op` to perform one execution step, from where it had stopped last time it was activated, until the next `wait` or function call. For each function `f` the thread may call, the component has an output control port $\texttt{f}_C$ to signal that the function is called, and an output data port to send the actual parameters; this data port is connected to the output data port of the module. A thread that can call a function `f` also has a `contf` control input, to resume its execution when the function terminates.

4.2 Contracts

Figure 11 gives example contracts for the threads and functions `T1, R1, T2, f1, f2`. The contract of a function `f` always starts with a transition triggered by `callf`, and needing the data input corresponding to the parameters; it is always a loop (to allow for successive executions of the function). If the body of the function does not wait, nor call other functions, the contract is simply a loop on the initial state, producing `endf`. In the example, this is the case for `f2`. If the body of the function has `wait` statements, the contract has additional states, with outgoing `op` transitions, which need the data input corresponding to the event which is waited for (this is the case for `f1`). The contract of a thread is made of transitions triggered by `op` or the $\texttt{contf}_i$ corresponding to the functions $\texttt{f}_i$ it may call. Each transition that produces $\texttt{f}_{iC}$ also provides the data output corresponding to the parameters, and reaches a state whose only outgoing transition is triggered by $\texttt{contf}_i$.

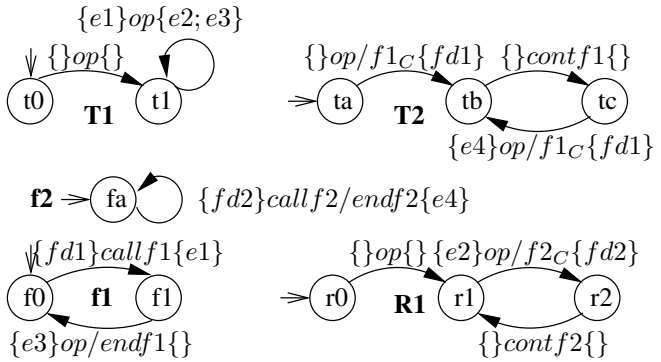

Figure 11: Example 42 Contracts for the Components of Fig. 10

To better understand the relation between 42 contracts and SystemC code, notice that contracts can be *extracted* automatically from SystemC code (see section 4.5). Figures 12 and 13 give an example SystemC thread, and the corresponding contract. The states of the contract correspond to points where the SystemC code waits for events, or calls functions (the notation ⓐ is added for the example, it is not part of SystemC). A transition `op` corresponds to a piece of code that stops on a `wait`, or on a function call. Events are translated into data dependencies. The tests on data produce non-determinism. For instance, from state ⓑ, there are three possible paths, depending on `(x < 42)` and `(y<5)`, leading to states ⓑ, ⓒ or ⓔ.

```
ⓐwhile(true){
  x++; e3.notify(); e4.notify(); ⓑwait(e1);
  if(x < 42){ ⓒp.f(x); ⓓp.g(x); }
  y=0;
  while(y<5){ y++; ⓔwait(e2); }
}
```

Figure 12: A SystemC thread

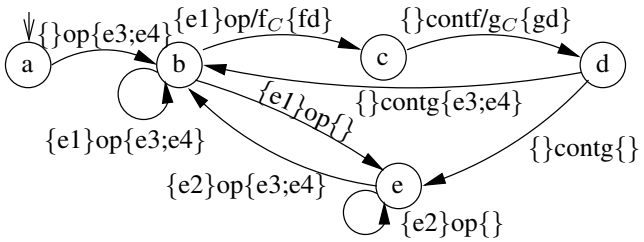

Figure 13: Non-Deterministic Contract for Fig. 12

4.3 Controllers

The control ports `op` in 42M are introduced for asynchronous simulation: a simulation is a sequence of activations of `main` with input `op`; CTRL1 translates each of these into an activation of `module1` or `module2`, through its parametrized input `op(n)`. CTRL1 is very similar to the contract interpreter of [2], as recalled in section 3.4. However, in [2], we had only one level of hierarchy, and the controller could simply interpret a set of *explicit* contracts, while `module1` and `module2` have no explicit contracts. Another difference with [2] is that CTRL1 also produces the effect of function calls.

Only the leaf components, corresponding to threads and functions, have *explicit* contracts (Fig. 11). We could first compute statically the contract of a module from the contracts of its threads and transitions, and then apply the method of section 3.4. But this product could be quite big.

The solution we adopted for SystemC in 42 is to build the contracts of the modules dynamically. The idea is as follows: CTRL1 *asks* the components `module1` and `module2` to expose the transitions that are possible in the current state of their contracts, before choosing which of them to activate through the parametrized `op`. For this, we add two control ports to `module1` and `module2`: the input `enq` (for "*enquire*") is used CTRL1 to ask the components to reveal their possible transitions; the output `resp` is used by the component to answer this question. At the deepest level, CTRL2 and CTRL2' know the explicit contracts of the threads and transitions, and are in charge of answering the question `enq` with an output `resp`. The data type for this exchange between CTRL1 and CTRL2 or CTRL2' is a set of indexed transitions of the form <n $\in \mathbb{N}$, t $\in$ `Trans` >; `Trans` is a contract transition, i.e., a structure $\{req \in 2^{ID}, ic \in IC, ocs \in 2^{OC}, prod \in 2^{OD}\}$, where ID, OD, IC, OC are respectively the sets of data input, data output, control input and control output ports.

4.3.1 The controller CTRL1

Figure 14 is a sketch of the code for CTRL1. It produces the effect of function calls, passing the control flow from the caller to the callee and back. `arch` describes the relation between an output control `f`$_{iC}$ (resp. `endf`$_i$) of a component `C`, and the corresponding input control `Callf`$_i$ (resp. `contf`$_i$) of a component `C`$'$. For instance, `arch(<module1,f2`$_C$`>)=<module2,Callf2>` and `arch(<module2,endf2>)=<module1,contf2>` describe the function call from `module1` to `module2`.

CTRL1 translates a global `op` into a sequence of activations of the modules. First (line 7), it asks all the modules for the possible transitions in the current states of their contracts (with the control input `enq`). Each module answers with a set of indexed transitions, which are stored in `TT`. Then CTRL1 selects randomly a transition t whose control input is `"op"` (line 8). Then it activates the component C to which t belongs (lines 9, 10). The activation uses the control input `op(n)` where `n` is the index of the chosen transition. The transitions in `TT` with the control input `''op''` come from the contracts of threads and functions; all the data they need because of events are provided locally in modules. This is why the execution of such an `''op''` by CTRL1 does not need data inputs.

The `while` loop of CTRL1 implements the general effect of calling, and returning from, a function. As long as the activation of the current component (`C.op(n)` (line 10) or `C.(t.ic) (line 22))` produces a control output *oc*, CTRL1 selects from `arch`, the corresponding component `C`$'$ (and its transition t') (line 14) and activates it immediately (line 17). Notice that, during the execution of the loop, the global

```
1 controller1 is{
2 TT ⊂ ℕ× Trans // set of indexed transitions
3 CC ⊂ ⋃Component // highest level components
4 arch : (CC × OC)→(CC × IC)
5
6 for op do{
7   for all C ∈ CC do { C.enq; TT := TT ∪ C.resp }
8   select t | ∃ n. <n,t> ∈ TT ∧ t.ic="op"
9   let C ∈ CC | t ∈ C.resp
10  C.op(n) ;
11  while (t.ocs != ∅){ // |ocs| ≤ 1
12    let oc ∈ t.ocs
13    let t′, C′ ∈ CC |
14      ∃ n′.<n′,t′> ∈ TT ∧ arch(<C,oc>)=<C′,t′.ic>
15      t := t′; C := C′;
16      for all id ∈ t.req { id.put; id.get;}
17      C.(t.ic) ;
18 } }
```

Figure 14: Sketch of the code for controller 1

`op` is not terminated, which means the execution is *atomic*, as it should be w.r.t. to SystemC.

4.3.2 The Controllers CTRL2 and CTRL2'

The controllers at the deepest level (CTRL2 and CTRL2') perform an on-the-fly production of the contract of a module, from the contracts of its threads and functions. They compute an asynchronous product, restricted by the effect of event-based communication between the threads and functions. Figure 15 is a sketch of the code for CTRL2 and CTRL2'. It contains code for the control input `enq`, for the parametrized `op`, and for the $\texttt{callf}_i$s and the $\texttt{contf}_i$s.

For `enq`, consider a state of `module1` in which the available inputs are `AA`={`e1,e2`}, the contracts' states of `T1, R1` and `F1` are `SS`=<`t1, r1, f0`>. When `module1` is activated with `enq`, it outputs on its port `resp` the set `RET`={<0,{}op{}>, <1,{}op/f2$_C${fd2}>, <2,{fd1}callf1{}>}. The indexed transitions in `RET` are obtained from transitions in the contracts of threads and functions, whose local required inputs (i.e., `t.req` $\setminus$ `P`) are available (line 12). Only the global inputs/outputs (`t.req` $\cap$ `P`, `t.prod` $\cap$ `P` and `t.ocs` $\cap$ `P`) are still visible in t' stored in `RET` (line 14). When returning a set of indexed contract transitions, CTRL2 and CTRL2' also recall in variable `WHO` (line 15) that the transition t' number n corresponds to the original transition `t` and belongs to the contract of the component `C`. Several transitions may come from the contract of the same thread or function.

CTRL2 and CTRL2' also translate an input `op (n)` (line 20) into the appropriate activation of a thread or function with input `op` (line 23) after providing it with the required inputs (line 22). The component is selected according to the information stored in `WHO` (line 21).

Finally, CTRL2 and CTRL2' route the activations $\texttt{callf}_i$ (line 29) and `contfi` (line 31) to the appropriate component. Conversely, they copy the control outputs $\texttt{endf}_i$ and $\texttt{f}_{iC}$ of the threads and functions into the corresponding outputs of the module (for instance at line 26, but possibly also in the code of lines 29, 31). It is similar for the data ports in `P`.

After each activation of a thread or function (lines 23, 29, 31) the set of available data `AA` is updated (as in line 24). The consumed ones (`t.req`) are removed and the newly produced (`t.prod`) are added.

4.4 Comments

The correspondence between SystemC/TLM and 42 may seem complex. In particular, there are more ports and connections between components in the 42 version, and mimicking the effect of the SystemC scheduler with two levels of controllers is more complex than flattening the structure and relying on function calls, as in SystemC. However, we think 42 should not be blamed for that; it only makes explicit the complex

```
controller2 is{
WHO⊂ (ℕ × ⋃Component × Trans)
CR ⊂ (⋃Component × ⋃Contract)
AA ⊂ ⋃ID // set of available input data
P ⊂ ID ∪ OD ∪ OC ∪ IC //module's ports
SS ∈ ΠContracts_States

for enq do{
 n : int := 0 ;
 for all t:Trans |
    ∃ <C,R> ∈ CR ∧ <s,t,s'> ∈ R ∧ s ∈ SS
    ∧ t.req\P ⊂ AA do{
  let t':Trans |
   t'={t.req∩P, t.ic, t.ocs∩P, t.prod∩P}
  WHO := WHO ∪ <n,C,t>
  RET := RET ∪ <n++,t'>
 } resp:=RET;
}

for op (n) do{
 let C,t | <n,C,t> ∈ WHO
 ∀ id ∈ t.req {id.put; id.get;}
 C.op;
 AA := (AA\t.req)∪(t.prod)
 ∀ id ∈ (t.prod ∩ P) {id.put; id.get;}
 let oc ∈ (t.ocs ∩ P) oc:= C.oc;
}

for callfi do{ ... C.callfi ...}

for contfi do{ ... C.contfi ...}
```

Figure 15: Sketch of the code for CTRL2 and CTRL2'

synchronization patterns of TLM components that are hidden in the SystemC execution engine. In other words, the complexity of a component in its 42 version is a better representation of its intrinsic complexity, than the SystemC/TLM version.

The 42 version exposes exactly the information that a user should have in mind when trying to reuse a TLM component. If we want to use SystemC/TLM modules in other contexts (42 or another one), we need to *expose* functions offered to the other modules, including the control dependencies they imply. Conversely, the two levels of controllers allow to make the effect of events as *locally hidden* as it should be; this makes it possible to use TLM modules written in other languages, with a local managing of internal synchronization, with events or another mechanism.

The equivalence between an original SystemC model and its 42 version cannot be proved formally, unless SystemC is given a formal semantics first. In fact the 42 hierarchy of controllers, as an encoding of the SystemC execution engine, *is* a formalization of SystemC, not far from what was used in [16] with synchronous automata or in [8] with timed-graphs. As in [16] the correctness of the translation can be assessed by intensive simulation of SCTLM and 42M for the same inputs. The advantage of the 42 version is the *componentization* of the semantics: a TLM module can be given a meaning in isolation.

4.5 Implementation

42 is implemented in Java. It provides an architecture description language (ADL) for building systems from components, a simple imperative language for controllers, and its interpreter. The basic components can be written directly in Java (implementing a predefined interface) or be defined as Java wrappers on existing code compiled from other languages, and respecting a given set of functions (entry points in the object code). For instance, the modular code produced by the compilers of synchronous languages like Lustre [1] can be wrapped into 42 components, and executed by a dedicated synchronous 42 controller. The contract execution mode is a particular case of an asynchronous controller, which is not written by hand, but results from the interpretation of the contracts.

For this paper we defined a method for extracting 42 contracts automatically from SystemC code (section 4.2), using the SystemC front-end Pinapa [15]. Pinapa can also be used to generate automatically the architecture of the 42 model from the architecture of the SystemC model, and from an analysis of its code, to determine which events are waited for, and which functions are called (this is very similar to what we did for building the architecture of TLM timed models from untimed models in [6]).

Finally we developed an execution mode for 42 contracts together with SystemC code. First, we had to solve the general problem that appears when executing *non-deterministic* contracts together with *deterministic* code: how to establish the correspondence between the states of the contracts, and the control point in the actual execution? A solution to this intrinsic problem exists when the implementation (here, the SystemC code) is written by somebody who knows the contract; the programmer has to relate explicitly the code he/she writes to the states of the contract. This can be done by using a special annotation function `state`. Figure 16 is an example implementation of the contract of Fig. 13, where the correspondence is made (the initial state does not need to be specified with a `state` annotation).

```
while(true){
  x++; e3.notify(); e4.notify();
  state(b) ; wait(e1);
  if(x < 42){
     state(c);p.f(x);state(d);p.g(x);}
  y=0;
  while(y<5){y++; state(e); wait(e2);}
}
```

Figure 16: Annotations in SystemC code implementing the contract of Fig. 13

If we cannot rely on these annotations, there is no way to extract the correspondence automatically, even with sophisticated program analysis techniques. The points corresponding to states are easy to discover, but we cannot decide which of the contract states they correspond to.

The complete implementation of the mixed execution mode, with wrappers, can be found in [3]. We explain the principles below. A SystemC module can be compiled separately, yielding an object code with entry points for threads and functions. Each 42 component corresponding to a thread or function is a Java wrapper for the corresponding entry point in this object code, connected to it via a JNI interface.

We provide re-implementations of the `event` class, and of the `wait` and `notify` methods. Each time a piece of SystemC code calls a `wait (e)`, or a `e.notify()` this is intercepted by the wrapper, so that the 42 world knows about the events exchanged. We also have to intercept the function calls. This is done by reimplementing the class `port` of SystemC. The special function `state` we have just introduced is intercepted by the wrapper, which can inform the 42 controllers that interpret contracts. When CTRL1 executes an `op` (one simulation step), the wrapper lets a SystemC thread or function executes, until it reaches a `wait` of function call. The control will not return to the SystemC code until the effect of the `wait` or function call, which has been intercepted by the wrapper, is terminated.

The general effect of the execution with wrappers is that the original SystemC code of the threads and functions is now animated by the 42 controllers. The events are dealt with by CTRL2 and CTRL2'. The function calls and the scheduling are dealt with by CTRL1.

5 Example Use of the Approach

5.1 The System Under Study

The system of Figure 17 is made of: a DMA (Direct-Memory-Access) component, a CPU, a memory, and a bus. The behavior is as follows: the embedded software running on the CPU first writes something to the memory from address `a1` to address `a2`; then it programs the DMA to perform a transfer of this portion of the memory to another place (say, between addresses `a3` and `a4`). The advantage of using a DMA is that the CPU can then do something else while the memory transfer is performed. When it it finished, the DMA sends an interrupt to the CPU, which can repeat the same behavior.

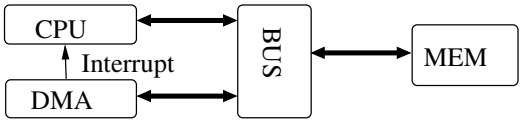

Figure 17: An Example System-On-a-Chip

In SystemC/TLM, the model is that of Figure 18. The four hardware elements are components, communicating via transactions. The communications from the CPU to the memory or the DMA, and the communications between the DMA and the memory, are transactions through the bus. The interrupt is a direct transaction from the DMA to the CPU. When the CPU writes to the memory, this is modeled as follows: the thread of the CPU calls a function of the bus, which itself calls the function `write` of the memory. The 42 model of Figure 19 is built as explained in section 4. We do not give all the details.

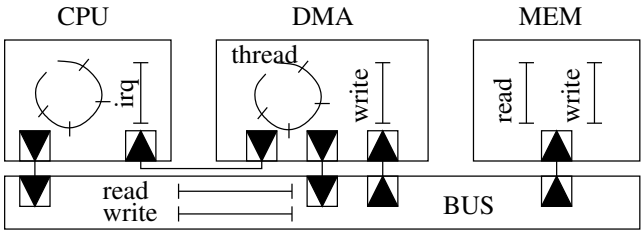

Figure 18: SystemC/TLM Model of the Figure 17

5.2 Uses of the Approach

When developing such systems in SystemC/TLM, it is often the case that there exist models for usual components like the DMA, the bus, and the memory. The model of a memory is simple, but the model of a real DMA can be quite complex, because it can allow several transfers at the same time, and/or it can

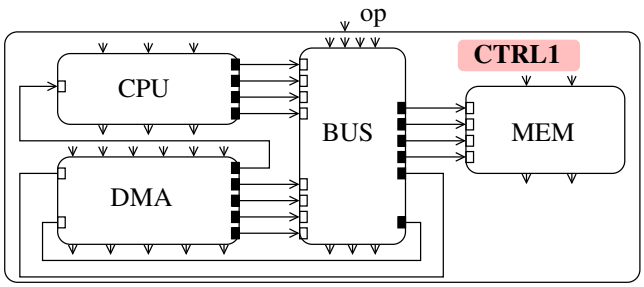

Figure 19: 42 model for the system of Figure 18

offer support for suspending/resuming a transfer, etc. The developer has to build a system by connecting those existing components, and then to write embedded code to be executed by the CPU (with native simulations, or by using an instruction-set-simulator, or ISS). Then he/she can simulate the whole system, and debug the embedded software. This may be complex because the debugging session deals with all the information of the components like the DMA. Even if we only want to check the synchronization effects between the embedded software and the DMA, we have to observe the full simulation through carefully chosen breakpoints. The correspondence we have defined allows the following alternative approach.

First, we extract a detailed architecture and control contracts from the SystemC code of the existing components. For instance, for a real DMA model written in SystemC, we would extract a detailed architecture like that of Figure 20 (showing that the DMA is made of a function `write` and a thread), and the contracts of Figure 21. This DMA is quite simple (only one transfer at a time, no suspension), but the SystemC code is already 50 to 100 lines.

In the contract of the thread, non-determinism comes from the abstraction of data in conditionals. State `t3` corresponds to the state of the DMA where it either terminates the whole transfer, or starts the transfer of a new word.

In the contract of the function `write`, we see conditions on specific values of a data input (e.g., ad_W=08). The DMA has three local registers at offsets `00, 04, 08`, where the CPU should write the start and end addresses of the memory transfer, and the transfer command, respectively. The first two ones are *data* registers; the last one is a *control* register, in the sense that writing to it triggers some behavior. This is visible in the contract of the function `write`: if the input address is `08`, the transition produces the event `e` that triggers the thread to start the transfer. One may wonder how such a contract may be extracted automatically from a piece of SystemC code. In most of the cases, the function is written as a `switch` statement with explicit constants for the cases, which makes the extraction feasible. In other cases, we could try to exploit the information on which registers are *control* ones, as specified in semi-formal interface definitions like IPXact [5].

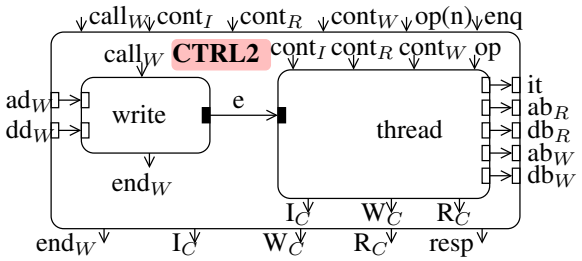

Figure 20: 42 model of the DMA component

Second, we write from scratch a non-deterministic contract for the CPU component (it is in fact a contract of the CPU, plus the embedded software that will run on it); we execute this CPU+SW contract together with the extracted contracts, in the 42 world; we see only the synchronization effects, not the general complexity of the SystemC code. Moreover, since the 42 model exposes clearly the control effects hidden in function calls, the points where the system has to be observed to debug the synchronization effects are already built in the model. The simulation speed can be quite good, also, because of the simplicity of

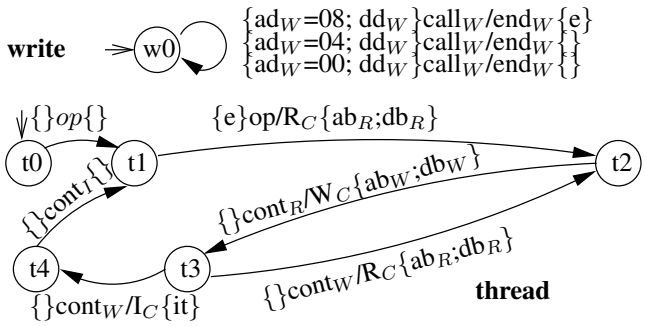

Figure 21: 42 contracts of the DMA component

the model.

Once we are happy with the contract of the CPU+SW, having played with it in the context of a complete SoC, we start developing the real embedded software (that will be executed by a SystemC component which is an instruction-set-simulator of the CPU). At the end, we may still execute the set of 42 contracts, but now, *together* with the actual SystemC code of the CPU (which, itself, executes the embedded software). This is a way of checking, dynamically, that the embedded software, together with the ISS, conform to the abstract CPU+SW contract. Notice that, in order to match the states of the contracts with the control points of the SW, we need to apply the mechanism explained in section 4.5 with the `state` keyword.

6 Conclusion

We have presented a method for combining standard SystemC/TLM components, as used in the industry, with 42 components. It is based on a structural correspondence between SystemC and 42. This provides a formalization of SystemC/TLM components, in the sense that it identifies the *data and control interface* of TLM components, and the TLM MoCC (encoded by the controllers of the 42 version). The effort of expressing in 42 the way SystemC/TLM components behave together, clearly separates the TLM semantics from the execution mechanics, which is not the case in SystemC. We are also able to animate a set of SystemC/TLM components using 42 controllers instead of the SystemC scheduler. The real SystemC code is executed by 42, thanks to wrappers.

Further work will be devoted to the following aspects. First, we will generalize our correspondence scheme and the associated implementation of the SystemC wrappers, to allow shared variables in a module, and timed models. This is a bit more complex than the version presented here, but follows the same lines (the global time keeper is at the highest level of the hierarchy). Second, we will investigate the idea of translating a 42 contract into a SystemC/TLM component, so that the contract can be "executed" together with ordinary SystemC/TLM components, in the SystemC world (for the moment, 42 contracts and SystemC/TLM components can be executed together, being animated by a 42 machinery, not by the regular SystemC scheduler). This is a way of improving simulation speed. Third, we will investigate formal verification for contracts extracted from SystemC/TLM designs (we have to build a model which is the product of all the extracted contracts; this model will be much smaller than the ones we use in [16], which contain all the details of the SystemC code). Finally, in the context of the OpenTLM project of the Minalogic competitiveness pole in Grenoble, we will apply the full approach on a large case-study provided by STMicroelectronics, in order to quantify the benefits of debugging simple control contracts first, either before having to deal with full SystemC code, or together with the full SystemC code of some components, but not all.

References

[1] J.-L. Bergerand, P. Caspi, N. Halbwachs, D. Pilaud, and E. Pilaud. Outline of a real time data-flow language. In *Real Time Systems Symposium*, San Diego, Sept. 1985. 4.5

[2] T. Bouhadiba and F. Maraninchi. Contract-based coordination of hardware components for the development of embedded software. In *COORDINATION'09*, Lisbon, Portugal, June 2009. 1.2, 3, 3.3, 3.4, 4.3

[3] T. Bouhadiba, F. Maraninchi, and G. Funchal. Formal and executable contracts for transaction-level modeling in systemc. Technical Report TR-2009-7, Verimag, May 2009. `www-verimag.imag.fr/index.php?page=techrep-list`. 4.5

[4] J. Buck, S. Ha, E. Lee, and D. Messerschmitt. Ptolemy: a framework for simulating and prototyping heterogeneous systems. *International Journal of Computer Simulation*, 4:155–182, April 1994. Special Issue on Simulation Software Development. 1.2, 3.1

[5] The Spirit Consortium. IP-XACT 1.4 specification. `www.spiritconsortium.org/releases/1.4`. 5.2

[6] J. Cornet, F. Maraninchi, and L. Maillet-Contoz. A method for the efficient development of timed and untimed transaction-level models of systems-on-chip. In *DATE*, Munich, Germany, Mar. 2008. 4.5

[7] F. Ghenassia. *Transaction Level Modeling With SystemC: TLM Concepts And Applications for Embedded Systems*. Springer-Verlag, 2005. 1

[8] P. Herber, J. Fellmuth, and S. Glesner. Model checking systemc designs using timed automata. In *CODES/ISSS '08*, pages 131–136, New York, NY, USA, 2008. ACM. 1.1, 4.4

[9] G. J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. 1.1

[10] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, Oct. 1997. 1.1

[11] F. Maraninchi and T. Bouhadiba. 42: Programmable models of computation for a component-based approach to heterogeneous embedded systems. In *ACM GPCE*, Salzburg, Austria, Oct. 2007. 1.2, 3

[12] F. Maraninchi and T. Bouhadiba. 42: Programmable models of computation for the component-based virtual prototyping of heterogeneous embedded systems. Technical Report TR-2009-1, Verimag, Jan. 2009. `www-verimag.imag.fr/index.php?page=techrep-list`. 3

[13] F. Maraninchi, M. Moy, C. Helmstetter, J. Cornet, C. Traulsen, and L. Maillet-Contoz. Systemc/tlm semantics for heterogeneous system-on-chip validation. In *NEWCAS/TAISA*, Montreal, Canada, June 2008. 1.1

[14] K. L. Mcmillan. The SMV system, Nov. 06 1992. 1.1

[15] M. Moy, F. Maraninchi, and L. Maillet-Contoz. The extraction tool for systemc descriptions of systems-on-a-chip. In *EMSOFT*, New-York, USA, Sept. 2005. 4.5

[16] 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*, 10(2-3), Sept. 2006. 1.1, 4.4, 6

[17] Open SystemC Initiative. *IEEE 1666 Standard: SystemC Language Reference Manual*, 2005. `http://www.systemc.org/`. 2

A Implementation of the wrappers for SystemC code in 42

A.1 Code of module1

This is the unmodified version of the SystemC module which is being wrapped. This module has two ports (a target `P1` and an initiator `P2`) and three events and two threads. The boolean variables `b1, b2, b3` are used to implement persistent events.

A.1.1 module1.h

```
#ifndef MODULE1_H
#define MODULE1_H

#include "ports.h"
#include "systemc.h"
using namespace std;

SC_MODULE(module1), public P1_if{
    P2_port_initiator p2;
    P1_port_target P1;

    SC_HAS_PROCESS(module1);
    explicit module1
      (sc_core::sc_module_name name): sc_module(name), P1(*this),
      b1(false), b2(false), b3(false){
       SC_THREAD(T1);
       SC_THREAD(R1);
    }
 public:
    module1();
    void T1();
    void R1();
    void f1(int x);
    sc_core::sc_event e1;
    sc_core::sc_event e2;
    sc_core::sc_event e3;
    bool b1,b2,b3;
};

#endif
```

A.1.2 module1.cpp

```
#include "module1.h"
#include "systemc.h"
#include <iostream>


void module1::T1(){
  int a = 0;
  while(true){
    if (b1==false)wait(e1);b1=false;
    a++;e2.notify();b2=true;
    a++;e3.notify();b3=true;
  }
}

void module1::R1() {
  int b = 0;
  while(true){
    b++;if(b2==false)wait(e2);b2=false;
    b++;p2.f2(b);
  }
}

void module1::f1(int x) {
  cout << x ;
  e1.notify();b1=true;
  if(b3==false)wait(e3);b3=false;
}
```

A.2 SystemC classes

To be able to load a SystemC module in isolation, we redefine SystemC classes needed by the module. Some of the classes are redefined only because the module may reference them, and others are re-implemented to log module's activity and report it to the Java part through the `wrapper` methods.

A.2.1 systemc.h

```
#ifndef SYSTEMCX_H
#define SYSTEMCX_H



#define SC_MODULE(name) \
      struct name : sc_core::sc_module

#define SC_HAS_PROCESS(name) \
      typedef name __THIS__MODULE__

#define SC_THREAD(name) \
      sc_42::create_behavior<__THIS__MODULE__>(#name, \
            this, &__THIS__MODULE__::name)

namespace sc_core {

   class sc_object {
      protected:
      sc_object();
      sc_object(const char*);
      virtual ~sc_object();
   };

   class sc_module_name {
      public:
      sc_module_name(const char*);
      sc_module_name(const sc_module_name&);
      ~sc_module_name();
      operator const char*() const;
      private:
      sc_module_name();
      sc_module_name& operator=(const sc_module_name&);
   };

   class sc_event {
      public:
      sc_event();
      ~sc_event();
      void notify();
      private:
      sc_event(const sc_event&);
      sc_event& operator=(const sc_event&);
   };

   void wait(const sc_event&);

   class sc_module : public sc_object {
      public:
      virtual ~sc_module();
      protected:
      explicit sc_module(const sc_module_name& n);
      sc_module();
      void wait(const sc_event&);
      private:
      sc_module(const sc_module&);
   };

   typedef sc_module sc_behavior;

}


namespace sc_42 {
   class scheduler {
      public:
      static void create_behavior(const char *name, sc_core::sc_module *host,
            void (sc_core::sc_module::*method)());
   };

   template<typename T>
   inline void create_behavior(const char *name,
         sc_core::sc_module *host, void (T::*method)()) {
      scheduler::create_behavior(name, host,
            static_cast<void (sc_core::sc_module::*)()>(method));
   };
}
#endif
```

A.2.2 systemc.cpp

```
#include "systemc.h"
#include "wrapperx.h"
#include <iostream>

namespace sc_core {
  // sc_object
 /*----------------------------------------------------------*/
  sc_object::sc_object() {}
  sc_object::sc_object(const char*) {}
  sc_object::~sc_object() {}

  // sc_module_name
 /*----------------------------------------------------------*/
  //creation of a module
  sc_module_name::sc_module_name(const char* name1) {

  }
  sc_module_name::sc_module_name(const sc_module_name&) {}
  sc_module_name::~sc_module_name() {}
```

```
  sc_module_name::operator const char*() const {
    return "";
  }

  // sc_event
 /----------------------------------------------------
  // Declartion of a new event
  // wrapper_new_event is implmenented in sc_42.cpp
  sc_event::sc_event() {
     wrapper_new_event(this);
  }
  sc_event::~sc_event() {}

  // The notifiction of an event is rooted to the java part
  // wrapper_notify_event is implemented in sc_42.cpp
  void sc_event::notify() {
    wrapper_notify(this);
  }

  // Waiting and event is rooted to the java part
  // wrapper_wait is implemented in sc_42.cpp
  void wait(const sc_event& e) {
     wrapper_wait(&e);
  }

  // sc_module
 /----------------------------------------------------
  sc_module::sc_module(const sc_module_name& n) {}
  sc_module::~sc_module() {}

  void sc_module::wait(const sc_core::sc_event& e) {
  }

  // sc_start
 /----------------------------------------------------
  void sc_start() {}
}


namespace sc_42 {

   void scheduler::create_behavior(const char *name,
         sc_core::sc_module *host, void (sc_core::sc_module::*method)()) {
     declare_thread(name, host, method);
   }

}
```

A.3 Re-definition of ports

The SystemC part presented above catches the activity of a module with regard to synchronization with events. Function calls are directly routed to the corresponding module. Function calls are performed by the module through its initiator ports. Re-implementing the ports is necessary to redirect the calls to the 42 Java part through the `wrapper` methods.

A.3.1 P1 port headers file

```
#ifndef PORTS_H
#define PORTS_H
#include <cstdlib>

class P2_if {
   friend class P2_port_initiator;
   public:
   virtual ~P2_if();
   private:
   virtual void  f2(int x)= 0;
};

class P2_port_target {
   friend class P2_port_initiator;
   public:
   explicit P2_port_target(P2_if& host);
   private:
   P2_port_target();
   P2_if& _host;
};

class P2_port_initiator{
   public:
   P2_port_initiator() : _target(NULL) {}
   void operator()(P2_port_target & target);
   void  f2( int x);
   private:
   P2_if *_target;
};

#endif
```

A.3.2 Initial implementation of P1

```
#include "ports.h"


P2_if::~P2_if(){}
P2_port_target::P2_port_target(P2_if& host): _host(host) {}
void P2_port_initiator::operator()(P2_port_target& target) {
   _target = &target._host;
}
void P2_port_initiator::f2(int x) {
   _target->f2(x);
}
```

A.3.3 modified implementation of P1

```
#include "ports.h"
#include "sc_42.h"


P2_if::~P2_if(){}
P2_port_target::P2_port_target(P2_if& host): _host(host) {}
void P2_port_initiator::operator()(P2_port_target& target) {
   _target = &target._host;
}
void P2_port_initiator::f2(int x) {
   wrapper_call_java_methode("f2");
}
```

B SystemC Wrapper

The SystemC/42 interface is implemented by the wrapper. This wrapper is composed of two parts. A `C++` part and a `Java` part. The two parts communicate with function calls using the Java Native Interface (JNI).

B.1 C++ side of the wrapper

It allows to load the specified module, retrieve function pointers (SystemC threads and modules functions) so that it is possible to call them from the 42 components (written in `Java`). Conversely, the execution of a thread or function may wait for an event, notify it or call a function. This is reported to the `Java` side by the classes redefined above using the wrapper functions. The `C++` side of teh wrapper implements two headers files. A header (`wrapperx.h`) for the methods called from the C++ side, and header for the native methods called from the `Java` side, directly obtained with JNI tools.

B.1.1 C++ Header

```
#ifndef WRAPPERX_H
#define WRAPPERX_H
#include "systemc.h"


void wrapper_wait(const sc_core::sc_event*);
void wrapper_notify(const sc_core::sc_event*);
void wrapper_new_event(const sc_core::sc_event*);
void wrapper_call_java_methode(const char*);
void declare_thread(const char *name,
                    sc_core::sc_module *host,
                    void (sc_core::sc_module::*method)());

#endif


. .
```

B.1.2 JNI Header

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class wrapper */

#ifndef _Included_wrapper
#define _Included_wrapper
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     wrapper
 * Method:    wrapper_load_module
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_wrapper_wrapper_1load_1module
  (JNIEnv *, jobject);

/*
 * Class:     wrapper
 * Method:    wrapper_call_methode
 * Signature: (Ljava/lang/String;)V
 */
JNIEXPORT void JNICALL Java_wrapper_wrapper_1call_1methode
  (JNIEnv *, jobject, jstring);

/*
 * Class:     wrapper
 * Method:    attach_to_me
 * Signature: (Ljava/lang/String;Ljava/lang/Object;)V
 */
JNIEXPORT void JNICALL Java_wrapper_attach_1to_1me
  (JNIEnv *, jobject, jstring, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

B.1.3 Implementation

```
#include <jni.h>
#include <map>
#include "module1.h"
#include <cstring>
#include "wrapper.h"
#include "wrapperx.h"
module1 * module;

JavaVM *jvm = NULL;
jobject wrapper_java;


jmethodID addEvent;
jmethodID waitEvent;
jmethodID notifyEvent;
jmethodID callaFunction;
JNIEnv * myJniEnv;
bool firstcall = true;

map<std::string,JNIEnv*> envMap;
map<std::string,jobject> objMap;



void wrapper_wait(const sc_core::sc_event *e){
  long int ref = (long int) e;
  (*myJniEnv).CallVoidMethod(wrapper_java, waitEvent, ref);

}

void wrapper_notify(const sc_core::sc_event *e){
  long int ref = (long int) e;
  (*myJniEnv).CallVoidMethod(wrapper_java, notifyEvent, ref);

}


void wrapper_new_event(const sc_core::sc_event *e){
  long int ref = (long int) e;
  (*myJniEnv).CallVoidMethod(wrapper_java, addEvent, ref);

}

void wrapper_load_module(){
  module=new module1("module1Y");
}

void wrapper_call_methode(const char* fun){
  if(! module){
    return;
  }
  // std::cout<<"wrapper.cpp : calling the methode : "<<fun<<"\n";
  if(strcmp(fun,"T1")==0)
    module->T1(); else
  if(strcmp(fun,"R1")==0)
    module->R1(); else
  if(strcmp(fun,"f1")==0)
    module->f1(42);
}

void wrapper_call_java_methode(const char* fun){
  int ref = 4242;
  (*myJniEnv).CallVoidMethod(wrapper_java, callaFunction,ref);
}



//**** implementatokn for the java side
JNIEXPORT void JNICALL Java_wrapper_wrapper_1load_1module
(JNIEnv * env, jobject obj){

  myJniEnv = env;
  if (jvm == NULL) myJniEnv->GetJavaVM(&jvm);
  int res;
#ifdef JNI_VERSION_1_2
  res = (*jvm).AttachCurrentThread( (void**)&myJniEnv, NULL);
#else
  res = (*jvm).AttachCurrentThread(jvm, &myJniEnv, NULL);
#endif
  if (res < 0) {
    fprintf(stderr, "Attach failed\n");
    return;}

  wrapper_java = obj;
  jclass cls = (*myJniEnv).GetObjectClass(wrapper_java);
  addEvent=(*myJniEnv).GetMethodID(cls, "declare_event", "(I)V");
  waitEvent=(*myJniEnv).GetMethodID(cls, "wait_event", "(I)V");
  notifyEvent=(*myJniEnv).GetMethodID(cls, "notify_event", "(I)V");
  callaFunction=(*myJniEnv).GetMethodID(cls, "callaFunction", "(I)V");

  wrapper_load_module();
}


JNIEXPORT void JNICALL Java_wrapper_wrapper_1call_1methode
(JNIEnv * env , jobject obj, jstring fun){
  const char * funC =(*env).GetStringUTFChars(fun,0);
  std::string funs =(*env).GetStringUTFChars(fun,0);

  myJniEnv = env;

  envMap.erase(funs);
  objMap.erase(funs);
  envMap.insert(pair<string, JNIEnv*>(funs,myJniEnv));
  objMap.insert(pair<string, jobject>(funs,obj));


   wrapper_java = obj;
   jclass cls = (*myJniEnv).GetObjectClass(wrapper_java);
   addEvent=(*myJniEnv).GetMethodID(cls, "declare_event", "(I)V");
   waitEvent=(*myJniEnv).GetMethodID(cls, "wait_event", "(I)V");
   notifyEvent=(*myJniEnv).GetMethodID(cls, "notify_event", "(I)V");
   callaFunction=(*myJniEnv).GetMethodID(cls, "callaFunction", "(I)V");
   wrapper_call_methode(funC);
}


JNIEXPORT void JNICALL Java_wrapper_attach_1to_1me
(JNIEnv *env , jobject obj, jstring fun, jobject obj1 ){
  const char * funC =(*env).GetStringUTFChars(fun,0);
  std::string funs=funC;
  myJniEnv = envMap[funs];
  wrapper_java = objMap[funs];
  jclass cls = (*myJniEnv).GetObjectClass(wrapper_java);
  addEvent=(*myJniEnv).GetMethodID(cls, "declare_event", "(I)V");
  waitEvent=(*myJniEnv).GetMethodID(cls, "wait_event", "(I)V");
  notifyEvent=(*myJniEnv).GetMethodID(cls, "notify_event", "(I)V");
  callaFunction=(*myJniEnv).GetMethodID(cls, "callaFunction", "(I)V");

  // std::cout<<"warperr.cpp : attaching jenv to thread "<<funs<<endl;
}
```

B.2 Java Side of the wrapper

The C++ side of the wrapper is responsible for loading modules, storing their function pointers, and intercepting the event manipulations and function calls. The Java side of the wrapper is responsible for executing the code of modules step by step. A step of a SystemC thread or function, is the execution of a piece of code of it between two waits/function calls. After executing a step, the thread (or function) should stop execution. The Java wrapper knows about the ending of a step, when the C++ wrapper reports to it that the function or thread being executed has made a wait on event, or a function call. Next time this thread/function is reactivated, the wrapper should start from the point where it stopped. The solution adopted is that each thread

or function is run on a Java thread, which is stopped when the module calls a function or waits for an event.

The Java wrapper is implemented as a *Singleton*, a unique instance is used by 42 components to call the corresponding piece of C++ code corresponding to it.

```
import java.util.Map;
import java.util.TreeMap;


public class wrapper {
    private  static wrapper instance;
    private  native void wrapper_load_module();
    private  native void wrapper_call_methode(String str);
    private  native void attach_to_me(String s, Object o);
    private Object finished;
    Map threadsMap = new TreeMap();
    Map componentsMap = new TreeMap();
    Map  eventsMap = new TreeMap();

    private wrapperThread current;

    static{
        System.loadLibrary("moduleloader");
    }

    private wrapper(){
        finished=new Object();
        wrapper_load_module();
    }

    public static wrapper getInsance(){
        if(instance==null) instance = new wrapper();
        return instance;
    }

    public void endExecution(){
        synchronized(finished){
            finished.notify();
        }
        threadsMap.remove(current.name);
        current = null;
    }

    public void suspend(){
        synchronized(finished){
            finished.notify();
        }
        synchronized(current.monitor){
            try{
                current.monitor.wait() ;
                attach_to_me(current.name, current);
            }
            catch (Throwable t){t.printStackTrace () ;}}
    }



    public void op(){
        next_step();
    }


    public void wait_event(int event){
        suspend();
    }

    public void notify_event(int e){
    }

    public void callaFunction(int f){
        suspend();
    }

    public void declare_event(int e ){
    }

    public void call(String s){
        this.wrapper_call_methode(s);
    }

    //METHODE TO BE CALLED FROM  COMPONENTS
    public void create_new_thread(String name, BasicComponent42 cpt){
        componentsMap.put(name , cpt);
        threadsMap.put(name, new wrapperThread(name, false, null));
    }

    public void create_new_function(String name, BasicComponent42 cpt){
        threadsMap.put(name, new wrapperThread(name, true, null));
        componentsMap.put(name , cpt);
        next_step(name);
    }



    public void next_step(String cptName){
        current = (wrapperThread)threadsMap.get(cptName);
        synchronized(finished){
            if(!current.isAlive()){
                current.start();
            }
            else{
                synchronized (current.monitor){
                    current.monitor.notify();
                }
            }
            try{finished.wait();}catch(InterruptedException e){}
        }
    }


    public class  BasicComponent42{

    }

    public class wrapperThread extends Thread {
        private String name;
        private String monitor;
        private boolean function;
        private int [] params;
        private boolean running=false;

        public wrapperThread(String threadName, boolean isFunction, int[] param){
            name = threadName;
            monitor = new String(name+"_monitor");
            function = isFunction;
            params = param;
        }

        public void run() {
            call(name);
            endExecution();
        }

    }
}
```