HAL
open science

Full simulation coverage for SystemC transaction-level models of systems-on-a-chip

Claude Helmstetter, Florence Maraninchi, Laurent Maillet-Contoz

To cite this version:

Claude Helmstetter, Florence Maraninchi, Laurent Maillet-Contoz. Full simulation coverage for SystemC transaction-level models of systems-on-a-chip. Formal Methods in System Design, 2009, 35 (Number 2 / October, 2009), p. 152-189. 10.1007/s10703-009-0075-z . hal-00429058

HAL Id: hal-00429058

https://hal.science/hal-00429058

Submitted on 19 Feb 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Full Simulation Coverage for SystemC Transaction-Level Models of Systems-on-a-Chip

C. Helmstetter[1,3], F. Maraninchi[1], and L. Maillet-Contoz[2]

[1] Verimag (CNRS, Grenoble INP, UJF), Centre équation - 2, avenue de Vignate, 38610 GIÈRES — France
[2] STMicroelectronics, 12 rue Jules Horowitz, 38019 GRENOBLE — France
[3] INRIA Grenoble - Rhne-Alpes, 655 avenue de l'Europe, 38330 Montbonnot Saint Martin - France

Abstract. Transaction-Level Models (TLM) are used for the early validation of embedded software. A TL model is a virtual prototype of the hardware part of a System-on-a-Chip (SoC). When using SystemC for transaction level modeling, the main parallel entities of the hardware platform (processors, DMAs, bus arbiters, etc.) are modeled by asynchronous processes, which are scheduled at simulation time. The specification of this scheduling mechanism is non-deterministic; the set of all possible schedulings of the parallel activities represents the physical parallelism faithfully. Moreover TL models may contain loose timing annotations (intervals for instance), and the set of all possible values of time in these intervals is also meant to represent the hardware behaviors faithfully.

However, any simulation engine is built on a deterministic scheduler, and at runtime will use specific values in the time intervals. This means that only a very small subset of all the possible schedulings and timings are exhibited during simulation. Some bugs may be missed if they are due to some behaviors of the hardware that are represented by other schedulings or timings.

For a given finite test scenario, the set of valid schedulings and timings of a model is finite, but far too large to be explored fully. We present a solution to cover the set of schedulings and timings efficiently. Our solution is based on dynamic partial order reduction and constraint solving techniques. It gives a complete scheduling and timing set, which guarantees the detection of all local errors and deadlocks for a fixed test scenario.

1 Introduction

1.1 Transactional Models for the Simulation of SoCs

A *System-on-a-Chip* (SoC) integrates many components on the same chip: a processor, several memory components, one or several buses, and specific components like video or audio decoders. A growing part of the functionality is implemented in the software part.

The development of the embedded software for such a dedicated hardware platform requires specific methods and tools, as explained later in this article.

The embedded software can, of course, be executed on the physical chip; this approach is fast and perfectly realistic with respect to the final SoC. However, this approach is not feasible for two main reasons: cost and time-to-market. If executing the software reveals a bug in the hardware, then it is prohibitively costly to correct it. In addition, the execution of software on real hardware offers no detailed debugging capabilities. Finally, because of time-to-market constraints, the embedded software should be ready and tested before the physical chip is delivered, to shorten the integration phase.

The embedded software can also be executed on the RTL description of the hardware. This approach is still precise; if necessary, hardware bugs can be fixed at this stage; all information is available for the debugging (the value of each signal at each clock tick is known). However, the RTL description is available too late with respect to time-to-market constraints. Furthermore, RTL simulation is too slow for complex data treatments: for example, the decoding of one image can take up to one hour.

The solution is to develop abstract models of the real system, with just enough details to be able to simulate the embedded software. This level of abstraction is called *Transaction level modeling* (TLM) [1]. Since the TL model of a system is less detailed than the RTL description, the TL model can be available earlier. TL simulations are much faster than RTL simulations: the decoding of an image takes only a few seconds. These advantages are obtained at the price of precision loss. The most abstract TL models do not allow timing performance evaluation, in particular because the non-functional features, like pipelines, are not modeled. The level of abstraction of TL models implies that automatic synthesis of RTL descriptions from TL models is not possible. Consequently, TL models do not replace RTL descriptions; they are used earlier in the design cycle.

1.2 Faithfulness of Models and Comprehensive Simulation

The systematic use of TL models raises two independent problems, illustrated by Fig. 1.

Faithfulness of Models Since a TL model is an abstraction of the not yet fully known hardware part of the SoC, the question of whether it represents faithfully the possible behaviors of the final chip is very hard to answer. However, it is quite clear that a high level model like the TL model has to be non-deterministic, so as to represent a *set* of possible behaviors.

In the TL models written in SystemC, the non-determinism affects the scheduling of the asynchronous processes representing real hardware entities, as well as the timing constraints used to simulate time.

Although the faithfulness question cannot be stated in formal terms, it is important to understand what elements in a modeling language can be used to represent *sets* of behaviors, and what are the guidelines for writing a

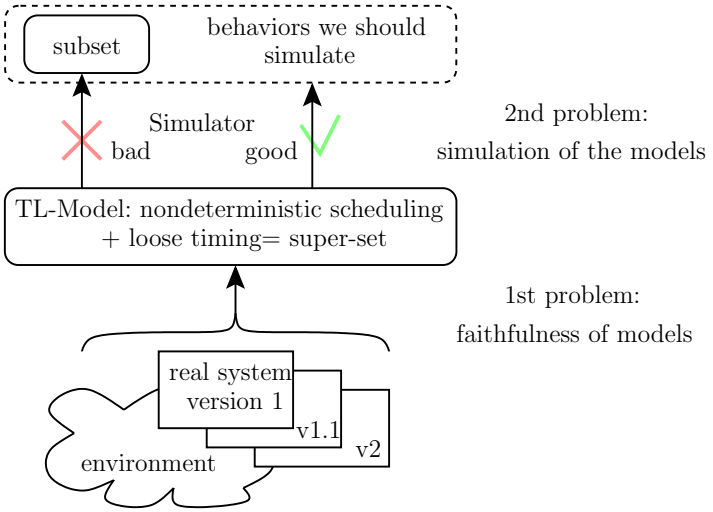


Fig. 1. The 2 problems: Faithfulness of Models and Comprehensive Simulation

faithful model. Section 3 further examines the faithfulness question. It helps understanding the class of models we have to simulate at the transaction level, and why the comprehensive simulation problem appears.

Comprehensive Simulation of Models Because of non-deterministic schedulings and loose timings, a TL model can define a large set S of behaviors. The faithfulness of a TL model with respect to a real hardware platform relies on the fact that the set S is a super-set of all the possible behaviors of the hardware. Validating a property of a SoC, given a TL model of the hardware, amounts to checking this property against all behaviors in S, and it cannot be done with one single (deterministic) simulation run.

Consider one test scenario (i.e., a sequence of data values as it could be generated by an independent tool). One solution would be to run several executions of this test scenario with random choices in the scheduler and for loose timing functions (the original OSCI SystemC simulator does not support random scheduling, but it is open-source and can be patched easily [2]). For any loose timing function call (specifying that the simulation should wait some finite amount of time, in some interval $[a, b]$), the execution engine draws a value in $[a, b]$. If the function call appears within a loop, a new value is drawn for each execution of the function. However, such a random policy slows the simulations without guaranteeing that interesting cases are explored. The resulting coverage is uncertain.

Another theoretical solution would be to systematically try all the choices. In practice, this solution fails due to the combinatorial explosion for any real-size example.

The solution we advocate is the following: for any test scenario, we should try all the choices (schedulings and timings) that "matter" for the result of

the test scenario. Intuitively, we try two different choices only if they are likely to give different test results. In practice, we try two different choices only if we cannot *prove* that they would give the same result.
The rest of the paper explains this principle, and the technical solutions to implement it for SystemC/TL models of systems-on-a-chip.

An important point to notice is the following: when testing a TL model, we are interested in exploring all potential behaviors of the hardware that matter for the software. Since the faithfulness of the TL model with respect to the hardware cannot be stated formally, the best we can do for ensuring this property is to split the problem into two parts: faithfulness is enforced by guidelines; once the TL model is built, we guarantee comprehensive exploration.

It is quite likely that the two problems we identified for SystemC/TL models of systems-on-a-chip are also present in other modeling and simulation contexts. As soon as we need to model the behaviors of physically parallel objects, we need a modeling language that offers some parallel construct, and the faithfulness problem arises if the objects do not behave deterministically. Moreover, the simulation of the model needs some kind of a scheduler: it is quite unlikely that the machines used for simulation offer as many processes as there are physically parallel entities to be simulated. Hence the comprehensive simulation problem also arises.

1.3 Contributions and Structure of the Paper

In this article, we contribute to the two points mentioned above:

- We first show why non-determinism is mandatory to design faithful models of Systems-on-Chip, and we explain the guidelines for writing faithful models.
- We present techniques to *cover* the simulations, i.e., to execute only those simulations that matter for a given test scenario. We are interested in proving properties, so we have to select enough behaviors such that a property which is false on a possible behavior is also false on at least one executed behavior. Since the sets of executed behaviors are far smaller than the sets of all possible behaviors, the method enables us to validate properties on large examples, up to medium-sized industrial case studies.

Concerning the second point, we first adapt the dynamic partial order reduction technique of [3] to SystemC, which enables us to cover the set of schedulings efficiently. We do not reuse directly the algorithm of [3]. We present a new algorithm, based on the same idea, which guarantees the same coverage property: all local errors and deadlocks for a given test scenario are detected. An error is local to a process of the system under test if this error can be checked dynamically by adding an assertion in this process. Deadlocks are not local errors since one needs to consider all the processes to detect them, but the scheduler of the simulator can detect them since it knows if at least one process is active. The correctness of our algorithm is based on the new concept of *scheduling constraint trees*, whereas the correctness of most partial order reduction algorithms, including [3], is based on *persistent sets*.

The idea is to look at the actions performed by the processes, in order to assess whether a change in their order (as what would be produced by distinct scheduler choices) could affect the final state (i.e., the content of the memory when the simulation terminates). If this change could affect the final state, then we generate a new scheduling. We repeat the same analysis on each newly generated scheduling. Successive iterations eventually give a *complete* (but not necessarily *minimal*) set of scheduling directives for the execution engine, which guarantees that all the executions of a test scenario "that matter" are explored. This guarantees the detection of all local errors and deadlocks for a fixed test scenario.

Secondly, in order to validate TL models with loose timing annotations, we also have to cover the set of valid timings too. We present a novel algorithm to solve this problem; it combines dynamic partial order reduction techniques with constraint solving techniques. It generates a set of valid timings that serve as execution directives. It guarantees, again, that all local errors and deadlocks for a fixed test scenario are detected.

Previous versions of these algorithms have been presented at the 6th FMCAD conference [4], at the 11th FMICS workshop [5], and in the PhD thesis [6]. In this article, we complement them such that they now generate *scheduling constraint trees*, which allow to define a mapping from any valid scheduling to an equivalent scheduling that is actually executed by the method.

The algorithms presented in this article have been implemented. The tool has been run on a real TL model provided by STMicroelectronics, which led to the discovery of a synchronization error that had not been found before.

The paper is structured as follows: Section 2 presents the SystemC language and the TLM library; Section 3 details the faithfulness problem and solutions. Section 4 details the simulation coverage problem. Section 5 recalls the theoretical background on partial order reduction techniques, and presents the new concept of the scheduling constraint tree, on which our algorithms are based. Section 6 explains how we cover the scheduling space for models with only fixed durations. The extension to models with loose timing annotations is described in Section 7. We present our implementation and its evaluation in Section 8, related work in Section 9, and we conclude with Section 10.

2 SystemC and the TLM Library

SystemC is a C++ library used for the description of SoCs at different levels of abstraction, from cycle accurate to pure functional models. SystemC comes with a simulation environment, and is becoming a *de facto* standard.

A TL model written in SystemC is based on an *architecture*, i.e., a set of parallel components and connections between them. Each component has typed connection *ports*, and its behavior is given by a set of communicating *processes* that can be programmed in full C++. For managing processes, SystemC provides a *scheduler*, and several synchronization mechanisms: the low-level *events*, the synchronous *signals* that trigger an event when their value changes, and higher

level mechanisms. When a SystemC model is simulated, first the static architecture is built by executing the so-called *elaboration phase* (**ELAB**), which creates components and connections. Then the scheduler starts running the processes of the components, according to the informal automaton of Fig. 2-(a). A simulation of a SystemC model looks like a sequence of *evaluation phases* (**EV**). Signal *update phases* (**UP**) and *time elapse phases* (**TE**) separate them (see Fig. 2-(b)).

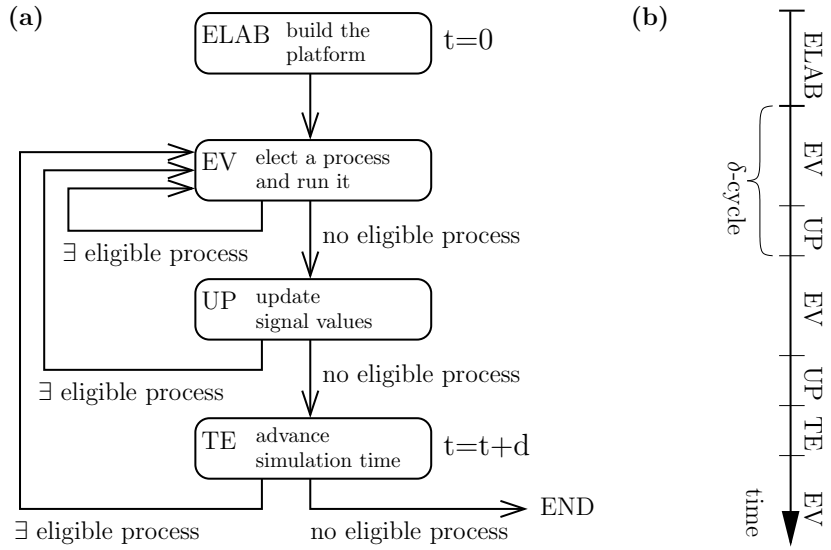


Fig. 2. (a) Automaton of the SystemC scheduler; (b) Diagram of an execution.

2.1 The SystemC Scheduler

The SystemC Language Reference Manual [7] describes the scheduler algorithm. At the end of the elaboration phase **ELAB**, some processes are *eligible*, some others are *waiting*. During the evaluation phase **EV**, eligible processes are run in an *unspecified order*, *non-preemptively*, and explicitly suspend themselves when reaching a *wait* function. A process may wait for some time to elapse, or for an event to occur. While running, it may access shared variables and signals, enable other processes by notifying events, or program delayed notifications. An eligible process cannot become "waiting" without being executed. When there are no more eligible processes, signal values are updated (**UP**) and δ-delayed notifications are triggered, which can wake up processes. A δ-cycle is the duration between two update phases. Since there is no interaction between processes during the update phase, the order of the updates has no consequence. When there is still no eligible process at the end of an update phase, the scheduler lets time elapse (**TE**), and awakes the processes that have the earliest deadline.

A notification of a SystemC event can be immediate, δ-delayed or time-delayed. Processes can thus become eligible at any of the three steps **EV**, **UP** or **TE**.

2.2 The TLM Library

In a TL model, all communications between components are done by transactions, which are implemented by function calls [8]. We distinguish the *initiator* port and the *target* port of a transaction. When a process in a component I (initiator) wants to communicate with another component T (target), it calls a method of one initiator port of I. The initiator port forwards the function call to its associated target port on T, which is linked to the code that implements this method. The initiator process continues when the function call returns. It does not yield back to the scheduler, allowing for atomic sequences of transactions.

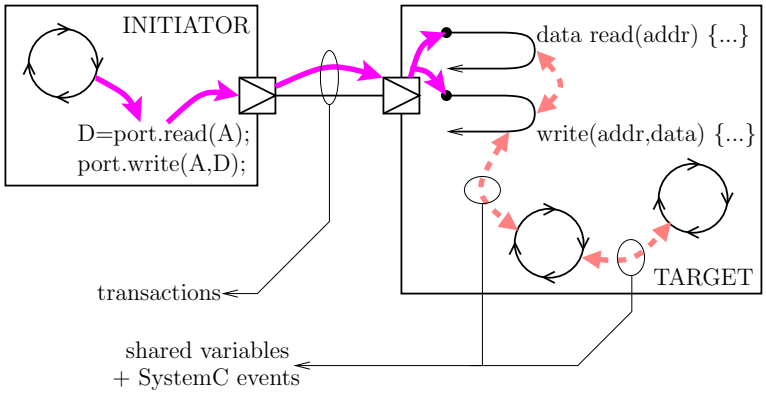


Fig. 3. Communications in TL models. Circles with arrows represent processes; large plain arrows represent the function calls (transactions); dashed arrows represent communications between processes.

In a TL model, processes communicate together inside components. Developers are free to use full C++ code for these communications. Generally, they use SystemC events and shared objects, such as atomic variables, FIFOs or arrays.

In this paper, we study the consequences of scheduling choices. The architecture and the transactions are independent of the scheduling. Hence, we mainly care about communications between processes. However, the problem studied is directly related to the high abstraction level of TL models, specifically to their use of asynchronous parallelism.

3 Faithfulness of TL models

Ideally, the embedded software should behave the same on the real system and on the TL model. However, the behavior of the hardware part considered alone

is non-deterministic, because the physical chip will run in a non-deterministic environment. On the other hand, the physical hardware is not available yet when the TL model is developed, and many technical variants of the hardware may still be investigated. To be faithful, TL models must specify a super-set of the realistic behaviors of the hardware. To do that, designers rely on non-deterministic schedulings and loose timings. Although rarely explained in the literature on transaction level modeling, using asynchronous models of hardware is not new. In 1999, [9] stated that:

> *In microprocessors and memory systems, several actions may occur asynchronously. These systems are not amenable to sequential descriptions because sequentiality either causes over-specification or does not allow for consideration of situations that may arise in a real implementation.*

We illustrate the effects of scheduling and timing on a simplified example, whose architecture is described by Fig. 4. There are a CPU, a memory and a DMA linked by a bus. A signal `running` from the DMA to the CPU notifies the activity of the DMA. Fig. 5 shows a piece of embedded software we want to simulate.

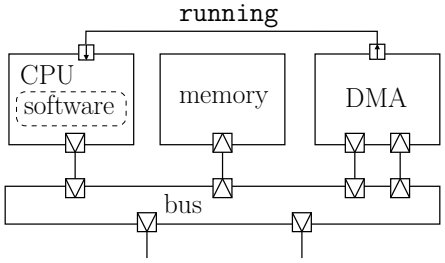


Fig. 4. Part of the architecture of the example

```
s1 //step 1: program and start the DMA
s2   bus.write(addr_DMA+src_reg_offset, image_addr);
s3   bus.write(addr_DMA+dest_reg_offset, dest_addr);
s4   bus.write(addr_DMA+start_reg_offset, 1);
s5 //step 2: do some independent computations
s6   x=bus.read(a); y=f(x); bus.write(a,y);
s7 //step 3: test if the DMA has finished
s8   if (running.read())
s9        {execute A}
s10  else {execute B}
```

Fig. 5. Embedded software executed by a CPU

Fig. 6 describes the DMA behavior, at the cycle-accurate abstraction level. It is here considered as a faithful description of the real system. In practice, we should refer to the RTL description, but for obvious reasons of space, we only give this simple description. It is enough to show the main issues for the design of TL models.

```
c1 wait(start_event);
c2 running.write(1);
c3 wait(clock);
c4 for (a=0; a<size_register; ++a) {
c5    wait(clock); d=bus.read(src_register+a);
c6    wait(clock); bus.write(dest_register+a, d);}
c7 running.write(0);
```

Fig. 6. Cycle-accurate description of the DMA

3.1 Modeling with Fixed Durations

A usual way of modeling a component at the transaction level is to use the SystemC `wait` function with fixed durations. Here, *"fixed durations"* means that the value is given as a constant in the test scenario, and is the same for all executions. Duration values refer to the SystemC simulated time. In our examples, they are expressed in nanoseconds ("SC_NS"). A timed TL model of the DMA is given by Fig. 7.

```
d1 wait(start_event);
d2 running.write(1);
d3 D=bus.read_block(src_register, size_register);
d4 bus.write_block(dest_register, size_register, D);
d5 wait(size_register*period+cst, SC_NS);
d6 running.write(0);
```

Fig. 7. TL model of the DMA with fixed durations

According to the context, an instruction-accurate model of the CPU is not mandatory to simulate the embedded software. We only have to annotate the software with `wait` functions, such as: `{`*step 1* `} wait(`*duration for step 1*`); {`*step 2* `} wait(`*duration for step 2*`); if (running.read()) {`*A*`} else {`*B*`}`.

In the real system, the CPU and the DMA will run concurrently. On the contrary, the simulator has only one processor, so it has to schedule the DMA and the CPU. The observed behavior depends on the order of the two events *"the DMA finishes"* and *"the CPU tests the* `running` *signal"*: either A is executed

or B is executed. To validate the software, both of these possibilities have to be checked. Unfortunately, with fixed durations, only one of them can be observed for a given input (the size of the transfer is fixed by the test scenario). Since the physical chip is still unknown, and the traffic on the bus may depend on non-deterministic factors, timing annotations can only be approximations and so we have no guarantee that the observed behavior will be the same on the real system. The conclusion is that fixed timing annotations reduce the set of observable behaviors to a subset of the realistic behaviors.

3.2 Modeling without Time Information

We can imagine designing TL models without any temporal information. Any `wait` function with a fixed duration needs to be replaced by a function that yields back to the scheduler, but leaves the current process eligible. There are no variants of `wait` functions to do that in SystemC (`wait(SC_ZERO_TIME)` prevents a process from being elected again before the next δ-cycle; `wait()` refers to a statically defined list of events). So a new function is needed; we call it "`yield`". This function allows the simulator to execute other tasks, but the current task remains eligible. If one is not interested in covering all the realistic behaviors, the empty function is a deterministic, fast and valid implementation of the "`yield`" function.

In the DMA TL model, we replace line `d5` by the code below:

d5: `yield();`

We apply the same transformation to all `wait` functions inserted in the embedded software.

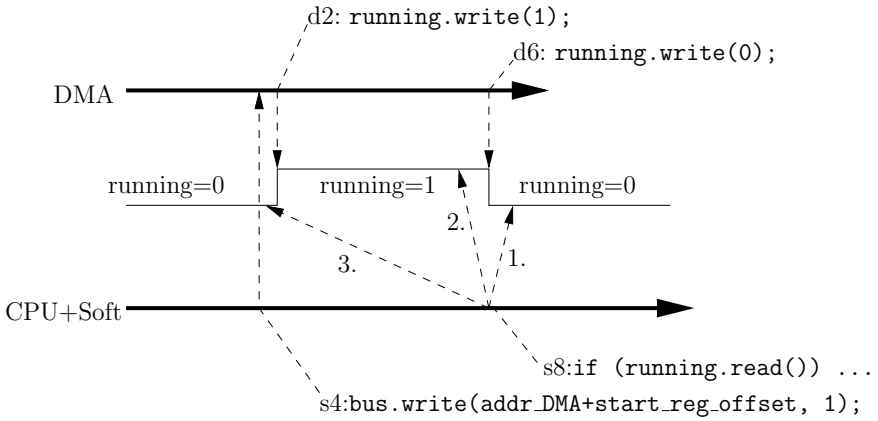


Fig. 8. Diagram of CPU+DMA Possible Executions

This new version allows more schedulings than the previous version with fixed durations. They are three possible behaviors, as shown by Fig. 8:

1. Line `s8` of the software is executed after the DMA has finished (line `d6`)
2. Line `s8` is executed before the DMA has finished but after it has started
3. Line `s8` is executed before the DMA has started (line `d2`)

The first two behaviors are realistic and so the TL model must allow their simulation. Conversely, the third behavior is not realistic. In this last case, the software thinks the DMA has finished whereas it has not started yet. Such non-realistic behaviors are very problematic for engineers using the models: rapidly, they will find more false errors than real bugs. A complete absence of timing information is therefore not suitable: this approach leads to a too large super-set of the realistic behaviors.

3.3 Modeling with Loose Timing Annotations

The solution currently investigated at STMicroelectronics is to use *loose timing annotations*. In place of fixed durations, we specify intervals of time. An instruction `pv_wait(duration, delta, time_unit)` means that the current task waits for a duration belonging to $[\texttt{duration} - \texttt{delta}, \texttt{duration} + \texttt{delta}]$.

Finally, we replace line `d5` of the DMA TL model by the code below:

d5: `pv_wait(size_register*period+cst, delta, SC_NS);`

We apply the same transformation to all `yield` functions inserted in the software. We get as many "delta" variables as `pv_wait` functions. Modifying their values allows to approach the set of realistic behaviors (Fig. 9).

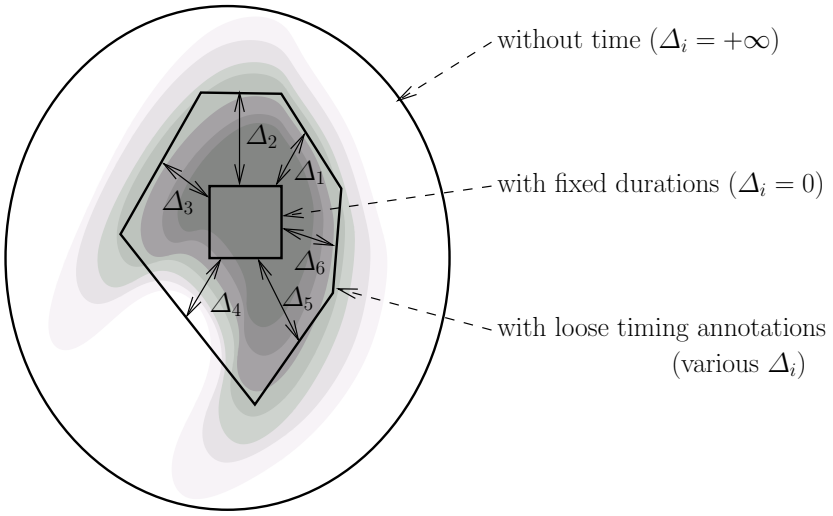


Fig. 9. Realistic SoC behaviors (gray shape, darker means more realistic), and behavior sets covered by untimed models, models with fixed durations and models with loose durations.

The idea of interpreting timing annotations in a loose way is quite natural. This idea was already present in some modeling approaches based on so-called *fuzzy* time (see, for instance, [10]), or based on *time Petri nets* [11].

To summarize, the guidelines for writing faithful models are:

- Avoid fixed durations, since using only fixed durations hides realistic behaviors.
- Do not remove all timing annotations, since untimed models produce too many non-realistic behaviors.
- Add loose timing annotations, because it is, up to now, the best way to tune the model in order to approach a faithful model of the real system.

Notice that we do not provide a formally justified way of choosing the loose timing values. But introducing the loose timing functions, and providing a way to cover the associated variations, are a good support for a test methodology.

In the rest of the paper, we will consider TL models without questioning their faithfulness again. The only remaining question is how to cover the choices allowed by non-deterministic schedulings and loose timings.

4 Comprehensive Simulation: the Effect of Scheduling and Timing

Before presenting our solutions for full simulation coverage, we present SystemC examples that show the effects of scheduling and timing choices on the result of a test.

4.1 Examples with Fixed Durations

```
void top::P() {
 wait(e);
 wait(20);
 if (x) cout << "Ok\n";
 else cout << "Ko\n";}
```

```
void top::Q() {
 e.notify();
 x = 0;
 wait(20);
 x = 1;}
```

Fig. 10. The `foo` example

```
void top::P()
   as in example foo
void top::Q()
   as in example foo
```

```
void top::R() {
  wait(20);
}
```

Fig. 11. The `foobar` example

To illustrate the possible consequences of scheduling choices, let us introduce two small examples of SystemC programs. Since the architecture and the transactions have no consequences for the synchronization analysis, we have fully

abstracted them. Fig. 10 shows the example `foo` made of two processes `P` and `Q`. The example `foo` has three possible executions depending on the scheduling, leading to very different results. We describe them below, with the following notation: an execution is denoted by a sequence of process names (to show which process is elected) and strings of the form "$[t \xrightarrow{d} D]$" that serve to show the **TE** phase of the scheduler; d represents the duration elapsed and D the new global date (these strings can be deduced from other information, but we include them for readability reasons). The three executions are:

- `P;Q;P;`$[t \xrightarrow{20} 20]$`;Q;P`: this scheduling leads to the printing of "Ok".
- `P;Q;P;`$[t \xrightarrow{20} 20]$`;P;Q`: the string "Ko" is printed. It is a typical case of *data-race*: `x` is tested before it has been set to 1.
- `Q;P;`$[t \xrightarrow{20} 20]$`;Q`: the execution ends after three steps only. The notification of event `e` has been executed before any "`wait(e)`" statement. Since events are not persistent in SystemC, process `P` has not been woken up. It is a particular form of *deadlock*.

Testing all executions of the `foo` example is useful because these executions lead to different final states. But consider now the `foobar` example defined in Fig. 11. `foobar` has 30 possible executions, but only 3 different final states. 12 executions are equivalent to "`R;P;Q;P;`$[t \xrightarrow{20} 20]$`;R;Q;P`", 12 to "`R;P;Q;P;`$[t \xrightarrow{20} 20]$`;R;P;Q`" and 6 to "`R;Q;P;`$[t \xrightarrow{20} 20]$`;R;Q`". Our method will generate only 3 executions, one for each final state (or equivalence class).

4.2 Examples with Loose Durations

Fig. 12 presents a new version `foochi` of the `foo` example, with loose durations. To execute this example, we must choose a value for t_1 between `3-d1` and `3+d1`, a value for t_2 between `40-d2` and `40+d2`, etc.

```
void P() {
  pv_wait(3,d1);   // t1
  wait(e);
  pv_wait(40,d2);   // t2
  if (x) cout << "Ok\n";
  else cout << "Ko\n";}
```

```
void Q() {
  pv_wait(6,d3);   // t3
  e.notify();
  x = 0;
  pv_wait(24,d4);   // t4
  x = 1;}
```

Fig. 12. The `foochi` example

If `d1` = `d2` = `d3` = `d4` = 0, then all delays are fixed and there are only two valid and equivalent executions (the index on process names is used to identify the occurrence): $\mathtt{P}_1;\mathtt{Q}_1$ or $\mathtt{Q}_1;\mathtt{P}_1$ followed by $[t \xrightarrow{3} 3];\mathtt{P}_2;[t \xrightarrow{3} 6];\mathtt{Q}_2;\mathtt{P}_3;[t \xrightarrow{24} 30];\mathtt{Q}_3;[t \xrightarrow{16} 46];\mathtt{P}_4$. $\mathtt{P}_1$ and $\mathtt{Q}_1$ occur at $T = 0ns$, $\mathtt{P}_2$ at $T = 3ns$, $\mathtt{Q}_2$ and $\mathtt{P}_3$ at $T = 6ns$. Next $\mathtt{Q}_3$ runs at $T = 24 + 6 = 30ns$. At last, the string "Ok" is displayed by $\mathtt{P}_4$ at $T = 6 + 40 = 46ns$.

Giving non-null values to the variables `d`i allows to explore more cases. If we take `d1` = `d2` = `d3` = `d4` = 2, then it is possible to permute the `wait` function call and the notification of the SystemC event `e`: we choose $t_1 = 5ns$ and $t_3 = 4ns$. With these values, it is still impossible to permute $\mathtt{Q}_3$ and $\mathtt{P}_4$. If we increase `d2` (resp. `d4`) to 10 (resp. 6), then $\mathtt{Q}_3$ and $\mathtt{P}_4$ may occur at the same time $T = 6 + 30 = 36ns$ $(30 = 24 + 6 = 40 - 10)$. Next, playing with the non-determinism of the scheduler allows to execute $\mathtt{P}_4$ before $\mathtt{Q}_3$. We have found the two errors of the `foo` example again.

Our method will generate timings and schedulings automatically, in order to find the executions that lead to these errors.

5 Formal Setting

The method for generating the schedulings and timings that matter for a given test scenario is based on partial order reduction techniques. We first review the main concepts used in this paper, expressing them in the SystemC/TLM context. Subsections 5.1 and 5.2 are only adaptations of existing works. Subsection 5.3 presents our new concept of *scheduling constraint tree*, on which the correctness of the algorithm presented in the next section is based.

5.1 Schedulings and Transitions

In the whole section, the system under test (SUT) is a SystemC program. We suppose that we have an independent tool for generating test cases that only contain the data. We call SUTD the object made of the SUT plus one particular test data[4]. We have to generate a relevant set of schedulings for this data.

When data is fixed, a SUT execution is entirely defined by its scheduling; a scheduling is entirely defined by an element of P^* where P is a set of process identifiers. We define *full states* of a SUTD to be full dumps of the SUTD memory, including the position in the code of each process. The SUTD can be seen as a *function* from the schedulings to the full states. This function is partial: not all the elements of P^* represent possible schedulings of the SUTD (because of the synchronization constraints between processes).

Definition 1 (Schedulings, Valid Schedulings and Terminated Schedulings). *Let M be a SUTD. P_M is the set of its processes; S_M is the set of its reachable full states; $F_M : P_M^* \longrightarrow S_M$ is its associated* function. *$V_M \subset P_M^*$ is the definition domain of F_M.*

- *A* scheduling *is an element of P_M^*.*
- *A* valid scheduling *is an element of V_M.*
- *A valid scheduling u is* terminated *if and only if there is no more eligible process in the state $F_M(u)$ (i.e., u cannot be extended: $\forall p \in P_M, up \notin V_M$); in this case, the state $F_M(u)$ is* final.

[4] Strictly speaking, the SUT includes a data generator, not a single piece of data. But the generator does not depend on the scheduling, hence the distinction is not necessary here.

A *transition* of a scheduling begins when the scheduler elects one process, and ends when this process yields back to the scheduler. Given a scheduling, we identify a transition by a pair: its *process identifier*, indexed by its *occurrence number*. A second index enables the scheduling to be make precise, when useful.

Definition 2 (Identification of Transitions). *$p_{i,u}$ denotes the i-th execution of the process p in the scheduling u.*

For example, in the scheduling $u = pqp$ there are 3 transitions: $p_{1,u}$, $q_{1,u}$ and $p_{2,u}$, in this order. This definition implies that writing "up_i" (i.e., scheduling u followed by the i-th execution of process p) is correct only if the process p is executed exactly $i - 1$ times in the scheduling u.

Note that, given two schedulings u and v of the same SUTD, two transitions $p_{i,u}$ and $p_{i,v}$ may read different values from memory or execute different pieces of code, due to interference with other processes.

A scheduling u defines a *total order* over its transitions. We have $p_{i,u} <_u q_{j,u}$ if and only if $p_{i,u}$ occurs before $q_{j,u}$.

5.2 Equivalence of Schedulings

The theory of partial order reduction relies on the definition of *dependent* transitions [12]. A dependency relation determines whether the permutation of two transitions has a consequence on the behavior. Intuitively, two transitions which do not access the same shared object are independent; a transition accessing only local variables is independent of all transitions of other processes.

An *equivalence relation between schedulings* can be directly defined from a dependency relation between transitions: two schedulings are equivalent if they differ only by the order of independent transitions. One equivalence class can be represented by a partial order, called the *happens-before relation* [13], such that each scheduling of this equivalence class defines a total order over its transitions, which is a linear extension of the happens-before relation.

When a dependency relation is *valid*, two equivalent schedulings lead to the same final state. As a consequence, it is sufficient to execute only one scheduling of each equivalence class to discover all local errors (i.e., errors that can be checked dynamically by adding an assertion in a SystemC process) and deadlocks. The common goal of all partial order reduction algorithms is to simulate at least one scheduling of each equivalence class, without simulating all the valid schedulings. As in [3], the dependency relation on transitions is deduced dynamically from the execution traces. The computation of this dependency relation is based on static rules depending on operation types: for instance, two transitions writing to the same variable are dependent.

The relation that contains all the pairs of transitions is always a valid dependency relation. However, if there are no independent transitions, then each equivalence class contains a single scheduling and the partial order reduction does not reduce the number of schedulings to be simulated. The more transitions we can prove independent, the more efficient the partial-order reduction

will be. Computing an efficient dependency relation for a SystemC program is discussed in section 6.3.

Below we give formal definitions of the dependency-based equivalence relation, the valid-dependency relation and the happens-before relation. These definitions do not provide a way to compute a valid dependency relation, but give the properties that a dependency relation must satisfy in order to guarantee the correctness of the concepts and algorithms presented later. We show how to compute dynamically a valid dependency relation for SystemC in subsection 6.3.

Definition 3 (Dependency-based Equivalence Relation). *Let $\mathcal{D}$ be a binary relation between transitions. The* equivalence relation $\equiv_{\mathcal{D}}$ *between schedulings, associated with $\mathcal{D}$, is the reflexive, symmetric and transitive closure of:*

$$\{(u,v) \in V_M \times V_M | u = u'p_iq_ju'' \wedge v = u'q_jp_iu'' \wedge (p_{i,u}, q_{j,u}) \notin \mathcal{D}\}$$

Definition 4 (Valid Dependency Relation). *A relation $\mathcal{D}$ between transitions is a* valid dependency relation *only if:*
$\forall u \in V_M, \quad (p_{i,u}, q_{j,u}) \notin \mathcal{D}$ *implies the conjunction of the 3 properties below:*

1. $\forall w$ *such that* $uw \in V_M, (p_{i,uw}, q_{j,uw}) \notin \mathcal{D}$
2. $\forall v \in V_M$ *such that* $u \equiv_{\mathcal{D}} v, (p_{i,v}, q_{j,v}) \notin \mathcal{D}$
3. $\forall v \in V_M$ *such that* $u \equiv_{\mathcal{D}} v$ *with* $v = v'p_iq_jv''$,
$$v'q_jp_i \in V_M \wedge F_M(v'p_iq_j) = F_M(v'q_jp_i)$$

Property 1 of this definition implies that the dependency of two transitions does not depend on the future; when transitions are defined statically, this property is implicit. Property 2 implies that if $u \equiv_{\mathcal{D}} v$, then $(p_{i,u}, q_{j,u}) \in \mathcal{D} \Leftrightarrow (p_{i,v}, q_{j,v}) \in \mathcal{D}$; informally, two transitions are dependent in all or none of the schedulings of an equivalence class. As a consequence, *conditional dependency relations* in the sense of [14] are not valid dependency relations according to our definition. Property 3 of this definition corresponds to the classic definition of a valid dependency relation.

At last, we recall the definition of the *happens-before relation*, which is a partial order describing an equivalence class. We note $[u]$ the equivalence class of a scheduling u: $v \in [u]$ if and only if $u \equiv_D v$. The happens-before relation of $[u]$ specifies which transitions cannot be permuted in u without permuting dependent transitions.

Definition 5 (Happens-before Relation). *Let u be a valid scheduling and $\mathcal{D}$ a valid dependency relation. The* happens-before relation *of an equivalence class $[u]$, denoted by "$\prec_{[u]}$", is defined by the property below:*

$$p_{i,u} \prec_{[u]} q_{j,u} \Leftrightarrow \forall v \in V_M \wedge v \equiv_{\mathcal{D}} u, p_{i,v} <_v q_{j,v}$$

5.3 Trees of Scheduling Constraints

The algorithms we present in the next sections are based on the concept of *scheduling constraints*. We use them to describe execution directives and equivalence classes. This concept is specific to the work presented in this paper. *Trees*

of scheduling constraints are used to prove the correctness of the algorithm presented in the next section, in the same way that *persistent sets* are used to prove the algorithm of [3] and many other partial order reduction algorithms.

Definition 6 (Scheduling Constraints). *A scheduling constraint is a tuple* (p, i, q, j)*, noted by "*$p_i < q_j$*". A scheduling* u *satisfies a scheduling constraint* $p_i < q_j$ *if and only if:*

$$q_j \in u \Rightarrow p_i \in u \wedge p_{i,u} <_u q_{j,u}$$

In that case, we denote: $u \models p_i < q_j$*. If* $c = p_i < q_j$*, then* c^{-1} *represents the* inverse *constraint "*$q_j < p_i$*".*

A scheduling satisfies a set of scheduling constraints if and only if it satisfies each constraint of this set.

Note that:

- A scheduling constraint can be interpreted in any scheduling.
- If a scheduling does not satisfy c, then it satisfies c^{-1}.
- A scheduling satisfies both "$p_i < q_j$" and its inverse "$q_j < p_i$" if and only if neither p_i nor q_j are in this scheduling (i.e., p is executed less than i times and q is executed less than j times). In particular, the empty scheduling satisfies all scheduling constraints.

In the following section, some particular cases will require the notion of terminated scheduling assuming a given set of scheduling constraints. A scheduling is said to be terminated assuming some constraints if there are no schedulings that are longer, valid, and satisfy all these constraints.

Definition 7 (Scheduling Terminated Assuming a Scheduling Constraint Set). *A valid scheduling* u *is* terminated assuming a set of scheduling constraints C *if and only if:*

$$u \models C \quad \wedge \quad \forall p \in P_M, \neg(up \in V_M \wedge up \models C)$$

("up" means the scheduling "u" extended by one execution of the process "p")

In particular, "u is terminated assuming $\emptyset$" is equivalent to "u is terminated", according to definition 1. Another direct consequence of this definition is that a scheduling u terminated assuming a set C, is terminated assuming any set D such that $C \subset D \wedge u \models D$.

We will use *complete* sets of scheduling constraints to describe equivalence classes. A set of scheduling constraints is *complete* if it defines the order of all pairs of dependent transitions, whose order is not already fixed (by the program itself or by previous scheduling constraints).

Definition 8 (Complete Set of Scheduling Constraints). *Let* C *be a set of scheduling constraints.* C *is* complete*, according to a dependency relation* $\mathcal{D}$*, if and only if:*

$$\forall u, v \in V_M \text{ with } u \text{ and } v \text{ terminated assuming } C, u \models C \wedge v \models C \Rightarrow u \equiv_{\mathcal{D}} v$$

Our algorithm to cover the schedulings of a test scenario generates a binary *tree of scheduling constraints*. A scheduling constraint tree is used to classify a set of schedulings according to which scheduling constraints are fulfilled by each scheduling. Schedulings are associated with the leaves and constraints are associated with the other nodes. Each tree node tagged by a constraint c has two sub-trees; we put the scheduling satisfying c on the *"true"* side, and we put others on the *"false"* side. Each path from the root to a leaf φ defines a set of scheduling constraints C, such that the scheduling associated with φ satisfies C; if C is complete, then this path defines an equivalence class. We explain below how a scheduling constraint tree can give a compact description of all the equivalence classes of a test scenario.

Definition 9 (Valid Tree of Scheduling Constraints). *A* tree of scheduling constraints *is a binary tree such that:*

- *Each leaf node φ is associated with a valid scheduling u (noted :"$\varphi = \mathrm{leaf}(u)$")*
- *Other nodes β have two children t (*"true" *side) and f (*"false" *side) and are associated with a scheduling constraint c (noted: "$\beta = \mathrm{node}(c, t, f)$")*

Given a sub-tree of root α, we note $\mathrm{nodes_of}(\alpha)$ *its nodes and* $\mathrm{leaves_of}(\alpha)$ *its leaves; formally:*

$$\begin{aligned}
\mathrm{nodes_of}(\beta = \mathrm{node}(c, t, f)) &= \{\beta\} \cup \mathrm{nodes_of}(t) \cup \mathrm{nodes_of}(f) \\
\mathrm{nodes_of}(\varphi = \mathrm{leaf}(u)) &= \emptyset \\
\mathrm{leaves_of}(\beta = \mathrm{node}(c, t, f)) &= \mathrm{leaves_of}(t) \cup \mathrm{leaves_of}(f) \\
\mathrm{leaves_of}(\varphi = \mathrm{leaf}(u)) &= \{\varphi\}
\end{aligned}$$

A tree of scheduling constraints α is valid *if and only if:*

$$\forall \beta = \mathrm{node}(c, t, f) \in \mathrm{nodes_of}(\alpha),$$

$$(\forall \gamma = \mathrm{leaf}(u) \in \mathrm{leaves_of}(t), u \models c) \text{ and } (\forall \delta = \mathrm{leaf}(v) \in \mathrm{leaves_of}(f), v \models c^{-1})$$

(i.e., the schedulings associated with the sub-tree t must satisfy c and the schedulings associated with the sub-tree f must satisfy c^{-1}).

According to this definition, each scheduling associated with a leaf of a valid tree must satisfy one constraint for each node from the root α to this leaf β. We note $C(\alpha, \beta)$ this constraint set. The set $C(\alpha, \beta)$ can be computed using the inductive rules (note that $C(\alpha, \beta)$ is defined even if β is not a leaf):

- $C(\beta, \beta) = \emptyset$
- $C(\mathrm{node}(c, t, f), \beta) = \{c\} \cup C(t, \beta)$ if β is a sub-tree of t
- $C(\mathrm{node}(c, t, f), \beta) = \{c^{-1}\} \cup C(f, \beta)$ if β is a sub-tree of f

As direct consequences of this definition, we have:

- When α is a valid tree, and $\beta = \mathrm{leaf}(u)$ is one of its leaves, we have: $u \models C(\alpha, \beta)$.
- When α is a tree, β a sub-tree of α, and γ a sub-tree of β, we have: $C(\alpha, \gamma) = C(\alpha, \beta) \cup C(\beta, \gamma)$.

Such a tree can be used to map any scheduling to a scheduling associated with a leaf. Given a scheduling u and a tree of root α, we define the mapping $MAP(u, \alpha)$ inductively, as follows:

- $MAP(u, \text{leaf}(v)) = v$
- $MAP(u, \text{node}(c, t, f)) =$ if $u \models c$ then $MAP(u, t)$ else $MAP(u, f)$

The computations $MAP(u, \alpha)$ and $C(\alpha, MAP(u, \alpha))$ are done by visiting the tree following the same path. As a consequence, the lemma below holds:

Lemma 1. $\forall u \in V_M, \forall \alpha, u \models C(\alpha, MAP(u, \alpha))$

Proof. By induction. There are three cases:

- $\alpha = \text{leaf}(v)$: then $MAP(u, \alpha) = \alpha$ and $C(\alpha, MAP(u, \alpha)) = C(\alpha, \alpha) = \emptyset$ so $u \models C(\alpha, MAP(u, \alpha))$
- $\alpha = \text{node}(c, t, f)$ and $u \models c$: then $MAP(u, \alpha) = MAP(u, t)$ and $C(\alpha, MAP(u, \alpha)) = \{c\} \cup C(\alpha, MAP(u, t))$; inductively $u \models C(\alpha, MAP(u, t))$ and by hypothesis $u \models c$ so $u \models \{c\} \cup C(\alpha, MAP(u, t)) = C(\alpha, MAP(u, \alpha))$
- $\alpha = \text{node}(c, t, f)$ and $u \models c^{-1}$: then $MAP(u, \alpha) = MAP(u, f)$ and $C(\alpha, MAP(u, \alpha)) = \{c^{-1}\} \cup C(\alpha, MAP(u, f))$; inductively $u \models C(\alpha, MAP(u, f))$ and by hypothesis $u \models c^{-1}$ so $u \models \{c^{-1}\} \cup C(\alpha, MAP(u, f)) = C(\alpha, MAP(u, \alpha))$

□

Fig. 13 shows an example of a scheduling constraint tree. The wide gray arrow represents the computation of $MAP(p_1q_1q_2p_2p_3, \alpha)$, where α is the root of the tree of scheduling constraints. According to the definition of MAP, $MAP(u, \alpha) = \beta$ implies that u satisfies one scheduling constraint for each node on the path from the root α to the leaf β. By construction, $MAP(u, \alpha) = v$ implies that u and v satisfy one scheduling constraint for each node on the path from the tree root to the leaf associated with v. Here, $p_1q_1q_2p_2p_3$ and $p_1q_1p_2q_2p_3$ satisfy $\{p_1 < q_1, q_2 < p_3\}$. This set of scheduling constraints is complete with respect to the untimed version of the `foo` example, whose code is given by Fig. 14; as a consequence, $p_1q_1q_2p_2p_3$ and $p_1q_1p_2q_2p_3$ are equivalent.

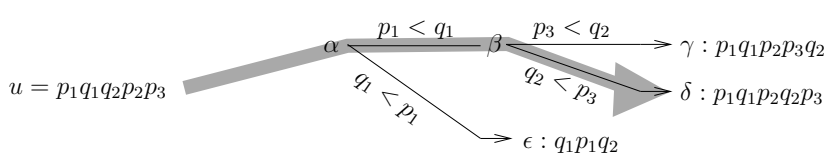


Fig. 13. Scheduling constraint tree for the untimed version of `foo`, and mapping of a scheduling $u = p_1q_1q_2p_2p_3$: $MAP(u, \alpha) = MAP(u, \beta) = MAP(u, \delta) = p_1q_1p_2q_2p_3$.

Trees of scheduling constraints are useful because of the property below:

```
void P() {                    | void Q() {
  wait(e);                    |   e.notify();
  yield();                    |   x = 0;
  if (x) cout << "Ok\n";      |   yield();
  else cout << "Ko\n";}       |   x = 1;}
```

Fig. 14. Untimed version of the `foo` example

Property 1. Let α be the root of a valid tree of scheduling constraints. If for all leaves $\beta = \text{leaf}(u)$, $C(\alpha, \beta)$ is a complete set of scheduling constraints and u is terminated assuming $C(\alpha, \beta)$, then for all valid and terminated schedulings $v \in V_M$, there exists a leaf $\gamma = \text{leaf}(w)$ of the tree whose root is α, such that $v \equiv w$.

Proof. We just have to choose $\gamma = MAP(v, \alpha)$. Indeed, we have:
- $MAP(v, \alpha) \models C(\alpha, MAP(v, \alpha))$ because $MAP(v, \alpha)$ is a leaf of the valid tree whose root is α
- $v \models C(\alpha, MAP(v, \alpha))$ according to the lemma 1

The scheduling v is terminated (assuming $\emptyset$), so it is terminated assuming $C(\alpha, MAP(v, \alpha))$ too (since $\emptyset \subset C(\alpha, MAP(v, \alpha))$). As v, $MAP(v, \alpha)$ is terminated assuming $C(\alpha, MAP(v, \alpha))$, and both v and $MAP(v, \alpha)$ satisfy the complete set of scheduling constraints $C(\alpha, MAP(v, \alpha))$, so they are equivalent. □

6 Covering the Schedulings of a Test Scenario

We describe in this section how to generate a set a schedulings that cover a test scenario. Concretely, we give an algorithm to compute a tree of scheduling constraints that satisfies the hypothesis of the property 1.

6.1 Main Algorithm

Fig. 15 gives an overview of our algorithm. For a given SoC description and a given test scenario (a SUTD as explained in section 5.1), the algorithm generates a set of directives for the execution engine so that it explores all the scheduling variants that matter for this test scenario. The goal is to generate at least one element in each equivalence class as formalized in the previous section.

We start by executing the SUTD without any constraints. We get an execution trace. The trace analyzer computes the dependencies between the transitions of this execution. The associated happens-before relation describes the equivalence class of this execution. The trace analyzer returns a description of this equivalence class in the form of a complete set of scheduling constraints. From these constraints, the generator of execution directives generates new sets of scheduling constraints, which specify uncovered schedulings. Next, we execute the SUTD with each new execution directive and follow the same loop for each

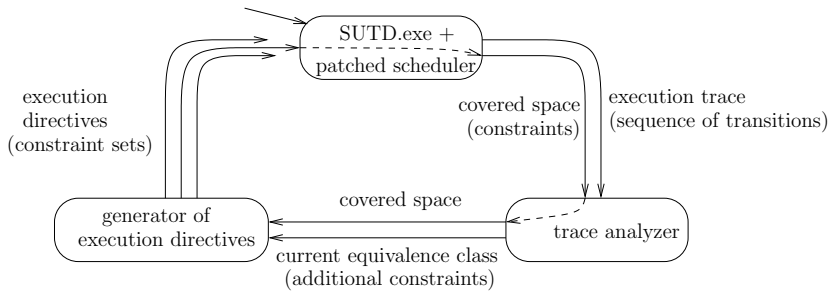


Fig. 15. Main loop for automatic generation of the set of execution directives

generated execution trace. To avoid looping infinitely, we keep at each iteration a set of scheduling constraints that describe what has already been covered.

When the algorithm stops, we get a set of schedulings such that any equivalence class has been covered at least once. Like the number of equivalence classes, the cost of the main algorithm is obviously exponential in the number of transitions.

Fig. 16 provides a formal definition of the algorithm GS. This algorithm requires two external functions:

- A function *execute* which takes one argument: a set of scheduling constraints D (called the *execution directive*). This function returns a scheduling u such that u is terminated assuming D (which implies $u \models D$, according to definition 7).
- A function *analyze* which takes two arguments: a set of scheduling constraints D and a scheduling u terminated assuming D. This function returns a set C such that: $D \cup C$ is a complete set of scheduling constraints and $u \models D \cup C$.

Subsection 6.2 presents how to implement a valid function "*execute*". Subsection 6.3 presents how to implement a valid function "*analyze*".

Implementing the algorithm GS requires choosing a way to pick one element c in C (line 6). The shape and size of the generated tree may vary for different choices, but the properties given below always hold.

GS returns a tree in which each leaf corresponds to a generated and executed scheduling. To prove that these schedulings cover all the equivalence classes, we first need to prove that GS always generates a **valid** tree of scheduling constraints.

Property 2. If the algorithm GS terminates for a given SUTD, then it returns a valid tree of scheduling constraints.

To prove this property, we need to prove the two lemmas below.

Lemma 2. *All calls to $GD_S(D, u, C)$ satisfy the condition $u \models D \cup C$.*

Proof (lemma 2). The function GD_S is called in lines 4 and 7 of Fig. 16. On line 4, $u \models D \cup C$ thanks to the specifications of the *analyze* and *execute*

```
1  GS(constraint set D) =                //initial call: GS(∅)
2  scheduling u = execute(D);
3  constraint set C = analyze(D, u);
4  return GD_S(D, u, C);
```

where the sub-function GD_S (abbreviation of "*generate directives*") is defined by the two inductive rules:

```
5  GD_S(D, u, ∅) = leaf(u)
6  GD_S(D, u, {c} ∪ C') = node(c,
7                              GD_S(D ∪ {c}, u, C'),
8                              GS(D ∪ {c^-1}))      //with c ∉ C'
```

Fig. 16. Main algorithm for the generation of schedulings

functions. If $u \models D \cup (\{c\} \cup C')$ is true on line 6, then $u \models (D \cup \{c\}) \cup C'$ is true on line 7 too. So the lemma is true by induction. □

Lemma 3.

$$\forall D, \forall u \in \textit{schedulings_of}(GS(D)), u \models D \textit{ and}$$

$$\forall D \forall v \forall C, \forall u \in \textit{schedulings_of}(GD_S(D, v, C)), u \models D.$$

Proof (lemma 3). Inductive proof: this lemma holds on line 5 since the previous lemma implies that $u \models D \cup \emptyset$; if this lemma holds on lines 7 and 8, any scheduling of the sub-tree built on line 6 satisfies either $D \cup \{c\}$ (line 7), or $D \cup \{c^{-1}\}$ (line 8), so it always satisfies $D = (D \cup \{c\}) \cap (D \cup \{c^{-1}\})$; if the lemma holds on line 4, then it holds obviously on line 1. □

Proof (property 2). Nodes are created only on line 6, with the constraint c, the *"true"* side $t = GD_S(D \cup \{c\}, u, C')$ and the *"false"* side $f = GS(D \cup \{c^{-1}\})$. The lemma 3 implies that $\forall \gamma = \text{leaf}(v) \in \text{leaves_of}(t), v \models c$ since $c \in D \cup \{c\}$, and $\forall \delta = \text{leaf}(w) \in \text{leaves_of}(f), w \models c^{-1}$ since $c^{-1} \in D \cup \{c^{-1}\}$. It is exactly what the definition of valid trees requires. □

Next we have to prove that $GS(\emptyset)$ satisfies the hypothesis of the property 1.

Property 3. Let $\alpha = GS(\emptyset)$.

$$\forall \beta = \text{leaf}(v) \in \text{leaves_of}(\alpha), C(\alpha, \beta) \text{ is complete.}$$

Lemma 4. *All calls to $GD_S(D, u, C)$ satisfy the condition $D \cup C$ is complete.*

Proof (lemma 4). Inductive proof: this lemma is true on line 4 because of the specification of the *analyze* function; if it is true on line 6 then it is true on line 7 too, since we just move one constraint from D to C. □

Proof (property 3). For each leaf β created on line 5, we note D_β the value of D at the time this leaf is created. Similarly, for each node γ created on line 6, we note D_γ the value of D at the time this leaf is created. In particular, $D_\alpha = \emptyset$ (with $\alpha = GS(\emptyset)$).

For any sub-tree β of α:

- If $\alpha = \beta$ then $C(\alpha, \beta) = \emptyset = D_\beta$
- If $\alpha \neq \beta$ then $\exists \gamma = \text{node}(c, t, f)$ such that:
  - either $t = \beta \wedge D_\beta = D_\gamma \cup \{c\}$, so $D_\gamma = C(\alpha, \gamma) \Rightarrow C(\alpha, \beta) = C(\alpha, \gamma) \cup C(\gamma, \beta) = D_\gamma \cup \{c\} = D_\beta$
  - or $f = \beta \wedge D_\beta = D_\gamma \cup \{c^{-1}\}$, so $D_\gamma = C(\alpha, \gamma) \Rightarrow C(\alpha, \beta) = C(\alpha, \gamma) \cup C(\gamma, \beta) = D_\gamma \cup \{c^{-1}\} = D_\beta$

By induction on the number of nodes between α and β, $C(\alpha, \beta) = D_\beta$.

In particular, for each leaf β, $C(\alpha, \beta) = D_\beta$ and the lemma 4 implies that the set D_β is complete. □

Finally, Property 1 implies that the algorithm GS generates at least one element of each equivalence class, which was our main goal.

6.2 Execution with Respect to Scheduling Constraints

We describe here how to execute a SUTD given a set of scheduling constraints D, called the *directive*. The goal is to provide a function *execute* that returns a scheduling u that is terminated assuming D.

Here is the idea: at the beginning of each execution step, the scheduler receives the list of eligible processes; we remove from this list all the processes whose election would violate a scheduling constraint at the current step.

Definition 10 (Index Function). *Let u be a scheduling and p a process, we note $ind_u(p)$ and we call* index of p in u *the number of occurrences of p in u.*

We use this first definition for the definition below. We can note that $p_i \in u \Leftrightarrow i \leq \text{ind}_u(p)$.

We "freeze" a process p after the execution of a partial scheduling u if the scheduling up (u followed by p) violates at least one constraint we must fulfill.

Definition 11 (Set of frozen processes). *Let $E(u)$ be the set of eligible processes after the execution of a (partial) scheduling u, and C a set of scheduling constraints. The set $F_C(u)$ of the frozen processes is defined by:*

$$F_C(u) = \{p \in E(u) | \exists (q_j < p_i) \in C, i = ind_u(p) + 1 \wedge j > ind_u(q)\}$$

Next, we patch the scheduler to restrict its choice to $E(u) \setminus F_C(u)$ at each step, as formalized by Fig. 17. Evaluating $E(u)$ (line 2) implies running one step of the system under test.

At the end of the execution, we get a scheduling u such that $u \models C$. This scheduling is always terminated assuming C (according to definition 7), and most of the time it is also terminated assuming $\emptyset$ (or according to the definition 1).

```
execute(directive D, prefix u) =
    if ∃p ∈ E(u) \ F_D(u)
    then execute(D, up)
    else u
```

Fig. 17. Algorithm of an *execute* function, which returns a scheduling terminated assuming a directive (if no prefix is given, then u is assumed to be the empty scheduling).

The schedulings which are not terminated assuming $\emptyset$ are not useful for the final set of schedulings that the GS algorithm generates, since they do not represent any equivalence class. The risk of getting only a partial scheduling depends on the scheduling constraints. For example, we always get a terminated scheduling if the constraint set is complete. We will discuss the ratio of these partial and useless executions for the case study (section 8).

6.3 Generation of a Complete Set of Scheduling Constraints

We explain here how to generate a *complete* set of scheduling constraints C (Cf. definition 8) for a given scheduling u, which is *terminated assuming* a directive D (Cf. definition 7). This requires computing the happens-before relation (Definition 5) which represents the equivalence class of u (Definition 3). The algorithm GS will execute the SUTD at least once for each constraint of the generated set. Consequently, we should generate a set as small as possible.

Property 4 below gives a hint on how to generate a complete set. The definition and the lemma below are required to define and prove this property.

Definition 12 (Co-eligible Transitions). *Two transitions are* co-eligible *if there exists a state in which both are eligible; formally:* $p_{i,u}$ *and* $q_{j,u}$ *are co-eligible if and only if:* $\exists v = v'v'' \in V_M, u \equiv_{\mathcal{D}} v \wedge v'p_i \in V_M \wedge v'q_j \in V_M$.

Note that if $p_{i,u}$ and $q_{j,u}$ are co-eligible, then $p_{i,w}$ and $q_{j,w}$ are co-eligible for any scheduling w equivalent to u.

Lemma 5. *Let* $u = u'u''$ *be a valid terminated scheduling such that* $u'p_i$ *is a valid scheduling. Let* $\mathcal{D}$ *be a valid dependency relation. There exists a scheduling* v *such that:* $v \equiv_{\mathcal{D}} u$ *and* $v = u'v'p_iv''$ *and* $\forall q_j \in v'$, $q_{j,v} \prec_{[u]} p_{i,v}$.

Proof (sketched). $u'p_i$ being a valid scheduling means that the process p is eligible after the execution of u' and the process p has been executed $i-1$ times in u'. In SystemC a process cannot be disabled by another process, and the scheduling $u = u'u''$ is terminated, so the process p is executed at least once in u after u'. It means that u'' contains p_i. Thus, the scheduling u is in the form $u = u'xp_iy$.

Next, according to the definition of the happens-before relation, all transitions in x that do not happen before $p_{i,u}$ can be moved after $p_{i,u}$ by successive permutations of independent transitions. □

Property 4. Let u be a valid terminated scheduling and $\mathcal{D}$ a valid dependency relation. The set $C(u) = \{\text{“}p_i < q_j\text{”} | p_{i,u} \prec_{[u]} q_{j,u} \ \wedge \ p_{i,u}, q_{j,u} \text{ are co-eligible}\}$ is complete.

Proof. Let v be a valid terminated scheduling such that $v \models C(u)$. We want to prove that $v \equiv_{\mathcal{D}} u$.

Let v' be the longest prefix of v such that: $\exists w \equiv_{\mathcal{D}} u, w = v'w'$. There may be two cases:

1. If $v' = v$ then w' is empty since v' is terminated, so $v = w$, and $v \equiv_{\mathcal{D}} u$;
2. Otherwise $v = v'p_iv''$. According to the lemma 5, there exists a scheduling x such that: $x \equiv_{\mathcal{D}} w$ and $x = v'x'p_ix''$ and $\forall q_j \in x', q_{j,x} \prec_{[w]} p_{i,x}$. Again, there may be two cases:
   - If x' is empty, then $v'p_i$ is a prefix of v longer than v' that satisfies the same property. As v' is already the longest, this case is not possible.
   - Otherwise let $q_{j,x}$ be the first transition of x'. The scheduling $v'q_j$ is valid (prefix of x) and v is terminated so v contains $q_{j,v}$ (property of the SystemC scheduler), and $q_{j,v}$ is after $p_{i,v}$. $q_{j,x}$ and $p_{i,x}$ are co-eligible, since both are eligible in $F_m(v')$. Moreover, $q_{j,x} \prec_{[w]} p_{i,x}$, so $q_j < p_i \in C(x)$. $x \equiv_{\mathcal{D}} u$ implies $C(u) = C(x)$ since the relations used in the definition of $C(u)$ are constant inside an equivalence class. However, v does not satisfy $q_j < p_i$. It is contradictory with $v \models C(u)$ so this second case is not possible.

   Both cases of 2. are not possible so only case 1. of the first enumeration is valid.

□

The algorithm described by Fig. 18 computes the set $C(u)$. It loops over all of the transition pairs, and adds a constraint “$q_j < p_i$” each time $q_{j,u}$ happens before $p_{i,u}$ and is co-eligible with $p_{i,u}$. Finally, it returns the set $C' = C(u) \setminus D$. The set $C' \cup D$ is complete because of Property 4. This algorithm uses two predicates *dependent* and *may-be-co-eligible*; Fig. 19 and Fig. 20 explain how to compute these two predicates.

We reuse the idea from [3] (Figure 4) to compute the happens-before relation. The happens-before relation is saved in an array LP (“*Last Predecessor*”) of size $size(u) \times size(P)$ where $size(u)$ is the number of transitions in u and $size(P)$ is the number of processes. The element $LP[num(p_{i,u}), q]$ is the number of the last transition of process q which happens before the transition $p_{i,u}$, or 0 if no transitions $q_{j,u}$ satisfies $q_{j,u} \prec p_{i,u}$; i.e.: $q_{j,u} \prec p_{i,u} \Leftrightarrow num(q_{j,u}) \leq LP[num(p_{i,u}), q]$ (num is defined by: $\forall u = vp_iw, num(p_{i,u}) = size(v) + 1$). We do not provide a proof for this algorithm; the correctness of this algorithm can be established using the invariant of line 13 and the property 4. We can optimize this algorithm by considering that $q_{j,u}$ and $p_{i,u}$ cannot be co-eligible if $q_j < p_i \in D$.

The algorithm of [3] does not generate a tree of scheduling constraints as GS, but it computes the happens-before relation associated with each generated scheduling too. The algorithm of Fig. 18 is an adaptation of the corresponding sub-part of the algorithm of [3] (in [3], the code to compute the happens-before

```
analyze(D, u) =
    initially: ∀n∀p, LP[n, p] = 0; C = ∅
    for n = 1 to size(u) do
    |   let p_i be the n-th transition in u
    |   LP[n, p] = n
    |   for m = 1 to n − 1 do
    |   |   let q_j be the m-th transition in u
    |   |   if m > LP[n, q] and dependent(q_{j,u}, p_{i,u}) then
    |   |   |   if may-be-co-eligible(q_{j,u}, p_{i,u}) then
    |   |   |       C = C ∪ {q_j < p_i}
    |   |   |   for all processes r do
    |   |   |       LP[n, r] = max(LP[m, r], LP[n, r])
    |   //here: ∀q∀j∀r∀k with num(q_{j,u}) < num(r_{k,u}) ≤ n,
                    q_{j,u} ≺ r_{k,u} ⇔ num(q_{j,u}) ≤ LP[num(r_{k,u}), q]
    return C \ D
```

Fig. 18. Algorithm for the generation of a complete set of constraints

relation is merged with the whole algorithm). In general, our analyzer generates a scheduling constraint where the algorithm of [3] adds a process identifier to a *backtrack* set. The predicates *dependent* and *may-be-co-eligible* are specific to SystemC, so that is one of our contributions.

The predicate *may-be-co-eligible*$(q_{j,u}, p_{i,u})$ returns ***false*** only in the cases below:

1. p and q represent the same process
2. one transition has been enabled by the other transition, by a notification of a SystemC event
3. they are not in the same δ-cycle, if all durations are fixed (we will talk of SUTDs with loose durations in the next section)

Fig. 19. Rules to compute the *may-be-co-eligible* predicate

Pairs of co-eligible and dependent transitions are implied by non-commutative accesses to a common shared object. As a first approximation, we can consider that two transitions that access a common shared object are always dependent. However, we refine the dependency relation by taking into account the kind of access. For example, two "read"s on the same variable do not imply any dependency.

Since the analysis is dynamic (that is to say, done for each executed scheduling), the use of arrays or pointers is not a problem: we always know which memory location has been actually accessed.

To prove the *dependent* and *may-be-co-eligible* predicates, one needs a formal semantics of C++ and SystemC. Only one particular case is easy to prove: *dependent*$(\ldots)$ always *true* and *may-be-co-eligible*$(\ldots)$ always *true*; if the *dependent* predicate returns always *true*, then GS generates all the valid

The predicate $dependent(q_{j,u}, p_{i,u})$ returns ***true*** only in the cases below:

1. they cannot be co-eligible ($may\text{-}be\text{-}co\text{-}eligible(q_{j,u}, p_{i,u})$ returns *false*)
2. $q_{j,u}$ and $p_{i,u}$ access the same shared variable, and at least one access is a *write*
3. $q_{j,u}$ and $p_{i,u}$ access the same SystemC event, and at least one access is a *notify*

Fig. 20. Rules to compute the *dependent* predicate

schedulings. If the *may-be-co-eligible* predicate returns *true* too often, then GS will generate schedulings which are useless since they are not terminated. If the *dependent* predicate returns *true* too often, then GS will generate schedulings which lead to the same final state and exhibit the same local errors (each "false dependent" may multiply the number of generated schedulings by 2, as shown by the case study).

In this paper, we consider only shared variables and SystemC events, but this list can be augmented. The computation of the dependency relation for abstract types such as FIFOs has already been detailed in existing works [15]. We detail below the dependency cases for the SystemC events.

There are two main operations on SystemC events: *wait* and *notify*. A transition can be enabled by a *notify*. There are two obvious cases of dependency:

- A transition contains a *wait* and the other transition contains a *notify* on the same event (the permutation can lead to a dead-lock, see the examples of section 4.1).
- A transition $p_{i,u}$ enables a transition $q_{j,u}$, so they are dependent but cannot be co-eligible.

However, as shown by the example below, considering these two cases is not enough to get a valid dependency relation.

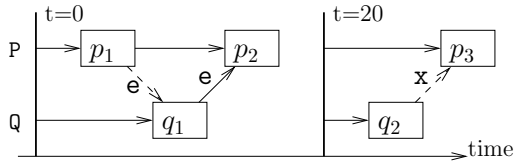
Suppose one runs this three-process model:

- Initial state: process `p` waiting for `e`, processes `q` and `r` eligible
- Process `p`: `cout <<'p';`
- Process `q`: `cout <<'q'; e.notify();`
- Process `r`: `cout <<'r'; e.notify();`

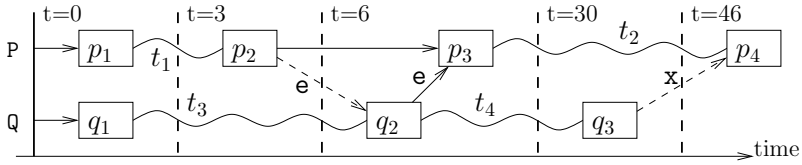
Each process executes exactly once before the execution terminates. Four schedulings are valid: $u = qpr$, $v = qrp$, $w = rqp$ and $x = rpq$. They lead to the same final state. However, there is no valid dependency relation such that these 4 schedulings are equivalent. Indeed, a valid dependency relation $\mathcal{D}$ contains at least $(q_{1,u}, p_{1,u})$ since pqr is not a valid scheduling (constraint 3 of definition 4); so $u \equiv_{\mathcal{D}} w$ would imply $(q_{1,w}, p_{1,w}) \in \mathcal{D}$ (constraint 2 of definition 4), which is contradictory with $w \equiv_{\mathcal{D}} x$ since w and x differ only by the order of p and q. Intuitively, the problem is that we cannot represent the property "p is after q or r" with a partial order. A solution to get a valid dependency relation is to consider that two notifications of the same event are dependent. Thus, the example

above has two equivalence classes: $\{qpr, qrp\}$ and $\{rpq, rqp\}$; the corresponding dependency relation is: $\mathcal{D} = \{(q_{1,u}, p_{1,u}), (q_{1,u}, r_{1,u}), (q_{1,v}, p_{1,v}), (q_{1,v}, r_{1,v}),$
$(r_{1,w}, p_{1,w}), (q_{1,w}, r_{1,w}), (r_{1,x}, p_{1,x}), (q_{1,x}, r_{1,x})\}$.
The analyzer computes only the subset of $\mathcal{D}$ that concerns the transitions of the analyzed scheduling. For the scheduling u of this example, the analyzer computes only that $\mathcal{D}$ contains $\{(q_{1,u}, p_{1,u}), (q_{1,u}, r_{1,u})\}$.

One way to represent an happens-before relation for human reading is to use graphics. Fig. 21-(a) represents the happens-before relation associated with the scheduling `P;Q;P;Q;P`, denoted $p_1q_1p_2q_2p_3$, of the `foo` program of Fig. 10. Fig. 21-(b) represents the happens-before relation associated with an execution of `foochi`. Each horizontal line is a process. Time elapses are represented by plain vertical lines if all delays are fixed, otherwise by dotted vertical lines. The wavy lines represent loose durations. Each box is a process transition. Arrows between boxes indicate that the two transitions are dependent; we draw dashed arrows if the transitions are co-eligible, plain arrows otherwise. We may move some transitions on the horizontal axis, remaining among the *valid and equivalent schedulings*, provided we do not permute two boxes linked by an arrow, nor move a transition through a plain vertical line. To generate a *complete* set of scheduling constraints for a scheduling, we just have to generate one constraint for each dashed arrow. For example, the happens-before relation depicted by Fig. 21-(a) gives the complete set $\{p_1 < q_1, q_2 < p_3\}$.



(a) `foo` with scheduling $p_1q_1p_2q_2p_3$



(b) `foochi` with scheduling $p_1q_1p_2q_2p_3q_3p_4$
and timing: $t_1 \mapsto 3, t_2 \mapsto 40, t_3 \mapsto 6, t_4 \mapsto 24$

Fig. 21. Graphical representations of a happens-before relation

7 Covering the Timings of a Test Scenario

The algorithm of the previous section applies to models with only fixed durations, or no time annotations at all. In this section, we present an extension for models with loose timing annotations.

7.1 The SystemC Models We Consider

First, we need to make the context of our work more precise. We allow the use of temporal information only inside the `pv_wait` function (and the `wait` which is a particular case). As defined in subsection 3.3, an instruction `pv_wait(duration, delta, time_unit)` means that the current process waits for a duration belonging to $[\texttt{duration} - \texttt{delta}, \texttt{duration} + \texttt{delta}]$. We consider that the global date (variable t of Fig. 2) is private and cannot be accessed by processes. This means that the processes cannot use the timing annotations to perform functional effects. This is consistent with the context of several TL models, where the timing annotations are added to a functional model for faithfulness and performance evaluation only.

Moreover, we restrict ourselves to SystemC programs whose executions have only one δ-cycle between two "time-elapse" phases. Indeed, the semantics of δ-cycle delays for abstract models with loose durations is unclear and such delays should not be used in timed TL models.

Any timed model that satisfies these constraints can be translated into an untimed model, whose executions are a super-set of the original model: we have to replace each call to `pv_wait(...)` or `wait(...)` by a call to `yield()`. As a consequence, a valid scheduling of a timed model can be executed without the timing information on this untimed variant of the model, giving the same functional behavior.

7.2 Main ideas

With examples that use only fixed delays, two transitions cannot be permuted if they occur at different dates. This is no longer true for SUTDs with loose delays: an alternative concrete timing may allow or force the permutation of some transitions. Now, for all pairs of dependent transitions such that their permutation is not prevented by explicit synchronizations, we have to determine whether there exist concrete timings allowing their permutation. If such timings do exist, we have to choose one among them and to re-execute the SUTD with it. In the algorithm presented in section 6.1 above, it is the only point that has to be rewritten for the generation of timings; the rest is identical.

For an execution of the SUTD and a set of scheduling constraints, we compute the conjunction of all temporal constraints that must be satisfied. Fortunately, all temporal constraints give linear constraints whose variables are the effective durations associated with the `pv_wait` calls. Consequently their conjunction gives a system of linear constraints S, which can be solved with linear programming techniques. If the system of constraints is built correctly, its solutions are

valid timings which make the given set of scheduling constraints feasible. With the current semantics of the `pv_wait` function, S defines an octahedron [16] (all variable coefficients are in $\{-1, 0, 1\}$) but not an octagon [17] (a constraint may use more than two variables).

7.3 The Temporal Constraints

To define the temporal constraints formally and the way we compute them, we have to introduce some complementary notations and definitions: with each `pv_wait(D,d)` function call present in the source code, we associate an identifier $\omega \in \Omega$, where Ω is an uninterpreted set of identifiers. We note $B(\omega)$ (B stands for "*Bounds*") the interval $[D - d, D + d]$ and $\#_u(\omega)$ the number of times an execution of ω occurs in a scheduling u. A timing T is a function from pairs $(\omega, n) \in \Omega \times [1..\#_u(\omega)]$ to durations $d \in \mathbb{R}^+$. $T(\omega, n) = d$ means that we wait for a duration d when we execute the function call identified by ω for the n-th time. The timing T is valid if and only if $\forall(\omega, n) \in \Omega \times [1..\#_u(\omega)], T(\omega, n) \in B(\omega)$.

There are two types of temporal constraints. First, the solution must correspond to valid timings. So for all $(\omega, i) \in \Omega \times [1..\#(\omega)]$ with $B(\omega) = [a, b]$, we add the two constraints $a < T(\omega, i)$ and $T(\omega, i) < b$. Second, each scheduling constraint implies a temporal constraint.

In order to build temporal constraints implied by scheduling constraints, we need the following definition. With each transition $p_{i,u}$, we associate a *symbolic date* noted $sdate(p_{i,u})$. A symbolic date is a sum of variables $T(\omega, i)$ and constants. We compute the symbolic date of a transition $p_{i,u}$ as follows:

1. If $p_{i,u}$ follows a `wait` with loose duration ($p_{i-1,u}$ ended by a call to `pv_wait`), then: $sdate(p_{i,u}) = sdate(p_{i-1,u}) + T(\omega, n)$ where ω is the identifier of this `pv_wait` function call and n is its occurrence number.
2. If $p_{i,u}$ follows a `wait` with fixed duration ($p_{i-1,u}$ ended by a call to `wait(k)`), then: $sdate(p_{i,u}) = sdate(p_{i-1,u}) + k$.
3. If $p_{i,u}$ has been enabled by an immediate notification from transition $q_{j,u}$, then: $sdate(p_{i,u}) = sdate(q_{j,u})$.
4. If p is initially eligible, then $p_{\mathbf{1}} = 0$.

We illustrate these rules on the example `foochi` with $u = p_1 q_1 p_2 q_2 p_3 q_3 p_4$. Symbolic dates do not depend on the timing. We have $sdate(p_1) = sdate(q_1) = 0$ (rule 4); next $sdate(q_2) = t_3$ and $sdate(p_2) = t_1$ and $sdate(q_3) = t_3 + t_4$ (rule 1). According to rule 3 on immediate notifications, we have $sdate(p_3) = sdate(q_2) = t_3$ and so $sdate(p_4) = sdate(p_3) + t_2 = t_3 + t_2$ (rule 1).

Let "$p_i < q_j$" be a scheduling constraint, so we build the associated temporal constraint as follows: we first evaluate $sdate(p_{i,u})$ and $sdate(q_{j,u})$, which yields two expressions e_1 and e_2; we then add to S the constraint "$e_1 \leq e_2$". With a set of scheduling constraints, we associate the conjunction of the linear constraints associated with each scheduling constraint.

7.4 The Algorithm

Fig. 22 presents the new algorithm. C is a set of scheduling constraints and u a scheduling. S is a linear program and the functions *is_feasible* and *solution_of* can be implemented with the simplex algorithm. The function is_feasible returns true if S is satisfiable; the function solution_of returns one solution of S. On line 2, the timing T may be incomplete, i.e., the value for some `pv_wait` function calls may be unspecified. If it is unspecified, then the simulation engine is free to choose any value in the given interval. Initially we call GT with an empty set of scheduling constraints and an empty timing. Let T_u be the concrete timing of the current scheduling u. In general, T_u is not a solution of the linear system S associated with $D \cup \{c^{-1}\}$ (line 7). However, T_u is always a solution of the system of linear constraints associated with $D \cup C$.

1 $GT(\text{constraint set } D, \text{timing } T) =$ //initial call: $GT(\emptyset, \emptyset)$
2 scheduling $u = execute(D, T)$;
3 constraint set $C = analyze(D, u)$;
4 return $GD_T(D, u, C)$;

where the sub-function GD_T is defined by the two inductive rules:

5 $GD_T(D, u, \emptyset) = \text{leaf}(u)$
6 $GD_T(D, u, \{c\} \cup C') =$
7 linear system $S = \bigwedge_{\text{“}p_i < q_j\text{”} \in D \cup \{c^{-1}\}} sdate(p_{i,u}) \leq sdate(q_{j,u})$;
8 if $is_feasible(S)$
9 then return node(c,
10 $GD_T(D \cup \{c\}, u, C')$,
11 $GT(D \cup \{c^{-1}\}), solution_of(S)$));
12 else return $GD_T(D, u, C')$;

Fig. 22. Main algorithm for the generation of schedulings and timings

We describe the first call to GT on the example `foochi` to illustrate this algorithm. If we ignore the temporal aspects, the analysis of $u = p_1 q_1 p_2 q_2 p_3 q_3 p_4$ generates the set $\{p_2 < q_2; q_3 < p_4\}$. Inductive calls to GD_T suggest two directives D': $\{q_2 < p_2\}$ and $\{p_2 < q_2; p_4 < q_3\}$. We have to check whether there exists at least one valid timing for each of them.

The first set of constraints $\{q_2 < p_2\}$ gives a linear system S' containing only the constraint $sdate(q_2) \leq sdate(p_2)$, which rewrites to $t_3 - t_1 \leq 0$. We must also respect the bounds on variables: $t_1 \in [1, 5]$ and $t_3 \in [4, 8]$. We request a solution from the linear programming library and get the solution $t_1 = t_3 = 4$. Finally, we call $GT(\{q_2 < p_2\}, \{t_1 = 4, t_3 = 4\})$. These scheduling constraints and this timing lead to the first error of `foochi` mentioned at the end of section 2.

The second set of constraints $\{p_2 < q_2; p_4 < q_3\}$ gives the two constraints $t_3 - t_1 \geq 0$ and $t_2 - t_4 \leq 0$. With the bounds $t_1 \in [1, 5]$, $t_2 \in [30, 50]$, $t_3 \in [4, 8]$

and $t_4 \in [18, 30]$, one solution is $t_1 = t_3 = 4$ and $t_2 = t_4 = 30$. Finally, we call GT again with this set of constraints and this timing as arguments. This leads to the second error of `foochi`.

7.5 Elements for the Correctness of the Algorithm

In the general case, GT generates **at least one** representative of each equivalence class, as GS does. On this example, we have generated one element of each equivalence class. First, we do not consider anymore that two transitions $p_{i,u}$ and $q_{j,u}$ can be co-eligible only if $date(p_{i,u}) = date(q_{j,u})$. We call GS' the algorithm GS in which this assumption has been suppressed (i.e., we remove the case 3 of the predicate "*may-be-co-eligible*$(p_{i,u}, q_{j,u})$" in subsection 6.3). Running GS' on the SUTD generates a very large set E' of schedulings which are valid if all bounds of loose durations are extended to $[0, \infty[$. It is equivalent to removing all delays of the SUTD. E' contains at least one element of each equivalence class of this "untimed" version of the SUTD.

Second, we have encoded the temporal constraints into a linear system S. The only difference between GS' and GT is that GT checks the feasibility of S. We know by construction that there exists an execution (u, T) which satisfies a set of scheduling constraints C if and only if the system S built from C is feasible. Hence GT generates all elements of GS' that satisfy the temporal constraints. Fig. 23 represents the sets of executions generated by GS, GS' and GT.

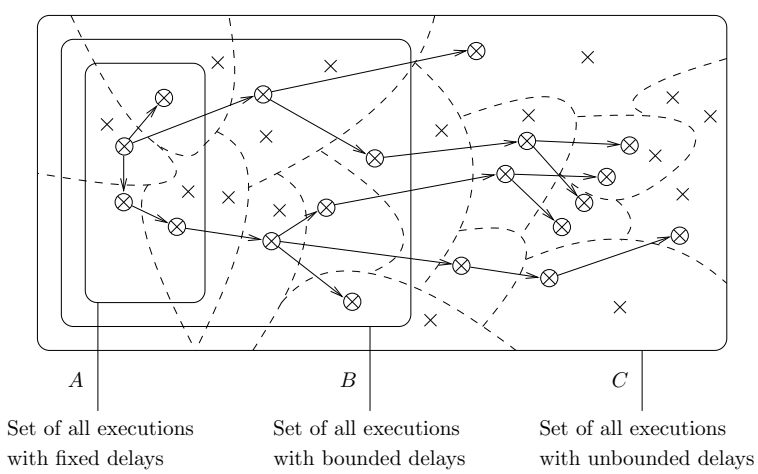


Fig. 23. Sets of all executions of the SUTD. The dashed lines delimit the equivalence classes. The surrounded crosses represent generated executions, with arrows from parent to child. GS returns the surrounded crosses of the set A, GT those of B and GS' those of C.

8 Case Studies

8.1 The Tool Chain

Fig. 24 is an overview of the tools[5]. The **analyzer** implements the algorithm described in section 6.3. It needs to be aware of all communication actions. Some of them can be detected by **instrumenting** the SystemC kernel, some others cannot (like accesses to a shared variable, that are invisible from the SystemC kernel).

For non-atomic shared objects such as SystemC events or mutexes, the easiest solution is to patch the class implementation itself. Detecting the accesses to atomic shared variables is more difficult. We tried two techniques:

1. **instrumenting each access**. For example, consider the instruction `x=y+2` where `x` and `y` are shared variables. The two following instructions are added close to the assignment: `recorder->read(&y); recorder->write(&x)`.
2. **instrumenting variable declarations**. For example, we replace `int x,y` by `probe<int> x,y`. The template class `probe<T>` has two attributes: an id and a value of type `T`. Each operation, such as assignment and conversion to type `T`, is overloaded to: 1) apply the operation on the value, then 2) notify the operation to the recorder.

We developed a prototype for the automatic instrumentation based on the open-source SystemC front-end Pinapa [18]. Our prototype is able to detect all shared variables and their accesses. However, applying the required modifications automatically to the source code still fails on some syntactic structures. With the second technique, few modifications have to be applied, so manual instrumentation is possible.

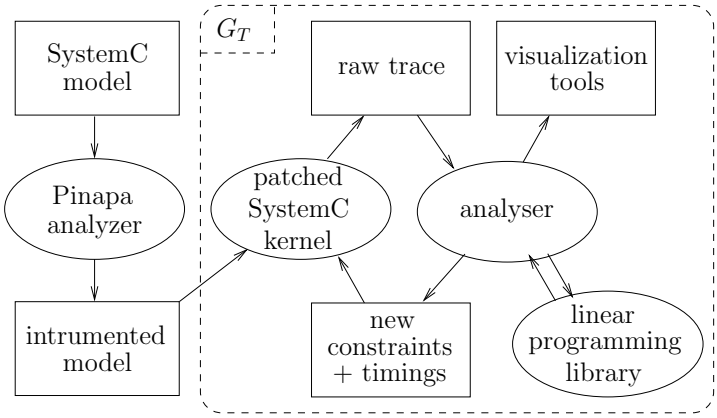


Fig. 24. The Tool Chain Architecture

[5] The tool chain is available on the web: `http://my-trac.assembla.com/scrv/wiki/`.

The instrumented SystemC program is compiled with a **patched SystemC kernel**. The patches are: 1) replacing the election algorithm of the SystemC scheduler by an interactive version, still complying with the SystemC specification; 2) adding code to record the communication actions that we do not detect by instrumenting the code of the processes, and their consequences (e.g., enabling of a process). When we execute the instrumented platform with the patched SystemC kernel, we can detect dependencies dynamically or save a detailed trace and run the analyzer afterwards. In both cases, we get a list of new execution directives to be executed, and a record of the computed dependencies, usable as input for other checkers or visualization tools.

8.2 Benchmarks

This subsection presents the results of our algorithm on two benchmarks taken from previous publications.

The Indexer Example The `indexer` benchmark first appeared in [3]. The original version is described for a preemptive scheduler, so the SystemC version is slightly different in order to preserve all the behaviors the original version may exhibit.

There are n components and one global 128-element array used as a hash table. Each component is composed of two SystemC threads that communicate using a shared variable and an event. Each component writes four messages in the global hash table. This corresponds to schedulings of length $11 \times n$. For $n \leq 11$, there is no collision in the hash table and all schedulings lead to the same final state. For $n \geq 12$ there are collisions hence non-equivalent schedulings. In this example, we generate exactly one scheduling per equivalence class. The number of generated schedulings is far smaller than the number of valid schedulings (at least $3.35E11$ for $n = 2$, and $2.43E25$ for $n = 3$). Results are summarized in table 1. On this example, it appears that our algorithm GS works as well as the algorithm of [3] combined with the sleep set technique.

components	generated schedulings	time
$1 \ldots 11$	1	$\leq$ 0.13 s
12	8	0.27 s
13	64	1.68 s
14	512	13.4 s
15	4096	112 s

Table 1. Results for the `indexer` example

The Chain Benchmark The `Chain` benchmark is a small SystemC/TLM program that appeared first in [19], and has been reused in [20]. It has been

written to evaluate formal verification techniques for SystemC/TLM, but it is a good example for dynamic partial order reduction, because: 1) it is not cyclic; 2) only the scheduling is not deterministic. Consequently, the GS algorithm and formal techniques provide the same information about the correctness of the program: both methods prove that no assertion is violated.

[19] presents a translation of SystemC/TLM into Promela, in order to verify properties of the TL model using SPIN. [20] describes a similar approach using the Lotos language and the CADP toolbox. The translation is partly manual. In this section, we use this benchmark to compare two approaches for the verification of TL models: 1) translation into an input language of an existing model checker; 2) dynamic and direct validation of the TL model without translation.

This benchmark consists of a chain of interrupt transmitter modules, whose length is parametrized by n. Modules communicate through transactions, and threads synchronize with events. Fig. 25 presents this benchmark for $n = 1$. To increase n, one adds a transmitter module between the last transmitter and the module Sink. There are always $n + 2$ SystemC threads (methods named `initiate`, `compute`, and `complete`) and $n+1$ events (attribute `e` of each module except `Source`). All target modules (`Transmitter` and `Sink`) export a method `f` to the previous initiator module (`Source` or `Transmitter`) through a pair of TLM ports.

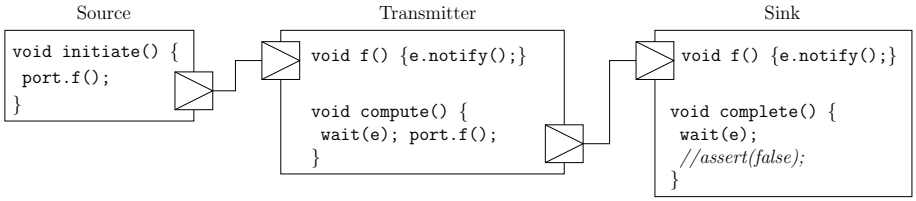


Fig. 25. The `chain` benchmark for $n = 1$

Table 2 recalls the results from [19] and [20] for the model-checking step, and presents the new results obtained with GS ("na" means *not available*). It appears that the number of equivalence classes computed by GS grows linearly whereas the state space grows exponentially. Consequently, on this particular benchmark, the technique presented in this paper scales up better than the approach based on translation and model-checking. A lot of partial order reductions are possible on this benchmark but the translations of SystemC/TLM into Promela or Lotos seem to hide some possible reductions.

8.3 The MPEG Decoder System

We have evaluated the tool on a small industrial case-study. This system has 5 components: a master, a MPEG decoder, a display, a memory and a bus model. There are about 50 000 lines of code and only 4 processes. This is quite

	using Spin [19]	using CADP [20]	*GS*
n=7	< 0.1 s	2.46 s	0.29 s (9 executions)
n=11	1.1 s	4.37 s	0.47 s (13 executions)
n=15	47 s	60.3 s	0.61 s (17 executions)
n=19	na	8293 s	0.78 s (21 executions)
n=23	na	na	0.94 s (25 executions)

Table 2. Results for the `chain` benchmark

common in the most abstract models found in industry, because there is a lot of sequential code, and very few synchronizations. Complete models of SoCs are typically 3 to 6 times bigger than this MPEG decoder. The test scenario provided by STMicroelectronics is stopped after the third decoded image, which corresponds to 150 transitions. One simulation takes 0.39 s. The experiments have been run on a Pentium 4 cadenced at 2.80 GHz.

First, we run the *GS* prototype on a timed version without loose durations. It generates **128 schedulings** in **1 min 08 s**. The total time spent splits into 50 s for running the SUTD 128 times and 18 s for generating the test directives.

Studying the recorded traces shows that each generated scheduling contains 7 pairs of dependent and co-eligible transitions. Such a property does not always hold, but is frequent for real programs which are designed not to depend on the scheduling. When this property holds, that is to say: each scheduling contains the same number n of pairs of dependent and co-eligible transitions, then there are 2^n equivalence classes.

The co-eligible and dependent transitions of the LCMPEG correspond to three different places in the source code:

1. Permutations that have no effect; we see how to avoid these spurious dependencies in subsection 8.4.
2. Permutations that have no functional effect but modify the simulation time and the number of transactions: the master does *polling* on the LCMPEG to test whether it has finished; according to the scheduling, the master can detect the end at the date d, or at the next read transaction at time $d + 10$.
3. A data-race whose participants are the LCMPEG and the display: with some schedulings, the display controller reads the third image instead of the second. This data-race is a real synchronization error since it leads to an incorrect display.

This bug concerning an incorrect display had not been found prior to the validation of this TL model with our new prototype.

The *GS* prototype can also be used on an untimed version. This untimed version is obtained by replacing all timed functions with their corresponding untimed functions. But the prototype fails to run to completion because the scheduling space to explore is far too large. Indeed, removing time constraints allows many new schedulings. For the untimed version, we estimate the number of relevant schedulings to about 2^{32}. Executing these schedulings would take many years. Most of this time would be spent exploring unrealistic schedulings.

The prototype of *GT* allows us to test bounded-delay versions which are intermediate between the fixed-delay version and the fully untimed version. We replace all function calls `wait(d)` by `pv_wait(d,d*r)`. The number of valid schedulings increases when the global variable `r` increases. The goal is to validate the SUTD with `r` as big as possible. We succeeded in validating this MPEG decoder system with `r` = 0.2. The *GT* prototype generates **3584 schedulings and timings** in **35 min 11 s**. One must spend 23 min 18 s to execute this system 3584 times, so the prototype spends 11 min 53 s to generate the test directives. Our goal is to validate the system with `r` = 0.5 but the first attempt shows that our prototype is not fast enough yet.

We said in section 6 that our method can generate useless schedulings for two reasons:

1. The number of equivalence classes depends on the computed dependency relation; the analyzer (Fig. 18) may not compute the most efficient dependency relation.
2. The algorithm *GS* (Fig. 16) may generate many schedulings in the same equivalence class, or schedulings which are not terminated (with respect to definition 1).

On this case study, the algorithm *GS* generates exactly as many schedulings as equivalence classes. That is to say, each generated scheduling is terminated and it corresponds to a distinct equivalence class. Actually, we have observed the generation of more schedulings than equivalence classes only on small examples that we have built for this purpose.

However, the dependency relation computed by the analyzer may be non-optimal in several situations. We illustrate one of these situations in the next subsection.

8.4 Improvement: Persistent Events

This MPEG decoder, as many other TL models, uses a pair (event, variable) to implement a *persistent event* as follows (`x` is initially 0):

Process p runs: `...; x=1; e.notify(); ...`
Process q runs: `...; if (!x) wait(e); x=0; ...`

The two valid schedulings pq and qpq lead to the same final state, but our tool currently generates both schedulings because it cannot prove it. The intuition is that these schedulings are not equivalent according to the dependency relation we compute.

In order to improve the analyzer, we propose to take this kind of structures into account explicitly for the computation of the dependency relation. For all models in which persistent events are needed, we suggest that the designers now use a dedicated structure, instead of encoding it by hand as above. Then this dedicated structure is recognized by our analyzer. Using this structure also improves the readability of the models.

Concretely, we define a new class `pevent` with two private attributes: an event `e` and a Boolean `x`, and two public methods `wait` and `notify`:

```
struct pevent {
  pevent(): x(false) {}
  void notify() {x=1; e.notify();}
  void wait() {if (!x) wait(e); else yield(); x=0;}
....};
```

Adding the `yield` function call in the `wait` method is mandatory because the theory on partial orders does not allow us to consider as equivalent two schedulings of distinct lengths (for example, pq and qpq above). It is a conservative approximation for safety properties as it only allows new schedulings. In practice, this approximation adds very few false errors since the developer must consider that the waiting process can yield back to the scheduler.

The example of code presented at the beginning of this subsection can now be replaced by the code below, where `pe` is an instance of the `pevent` structure, replacing `x` and `e`:

Process p runs: `...; pe.notify(); ...`
Process q runs: `...; pe.wait(); ...`

We upgrade the LCMPEG decoder to use this new dedicated structure and we complement our trace analyzer to take persistent events into account. Next, we run the GS prototype on the timed version (fixed durations) again. The GS prototype needs to generate **32 schedulings** instead of 128, and it takes only **13 seconds** instead of 1 min 08 s. On this program, we removed two "false dependencies", and consequently the number of generated schedulings is divided by 4.

This experiment shows that using persistent events leads to a significant improvement of validation time. Other patterns could be studied. The idea tested here is similar to the idea presented in the example 3.16 of [15]: we refine the dependencies between operations, but to do that we need to merge two objects in one new object.

9 Related Work

Related work can be found in at least three directions: the general problem of writing models that are faithful with respect to some real system; the formal techniques that have been investigated for SoCs and SystemC/TLM, including runtime-verification techniques, static verification, etc.; and the various algorithms that have been proposed for dynamic partial-order reduction.

9.1 Designing Faithful Models

The position paper [21] gives an overview of techniques which are of interest for the verification of SoC models. The authors target the validation of executable *micro-architectural specifications* (MAS), which are very similar to the TL models we are interested in. As they correctly state, there is no full formal equivalence

between the real system (RTL) and its model (MAS). They suggest using formal verification techniques to check the "compatibility" of the model against the real system. Even if we assume that it is technically possible, it is not a panacea. Indeed, such verification requires that both the model and the real system be available. In our context, models are developed and used long before the real system is built. Consequently, if a non-compatibility is found at this step, this requires a modification of the model, and all the work done on the model is invalidated.

As explained in section 3, our point of view is that models should be non-deterministic to specify a super-set of the possible real system behaviors, so as to avoid such a problem. Note that we can always remove non-determinism when we gain knowledge on the real system, without invalidating the work done (in our context: the embedded software).

As far as time is concerned, our notion of *loose* timing is related to the so-called *fuzzy time* modeling (see, for instance, [10]). The common idea is that a time delay in a high-level model can never be a fixed value. The model has to cover several behaviors of the real system, whose precise timing is not known. In all these approaches, tuning the *fuzzy* or *loose* timing annotations is a way of approaching the real system, but this cannot be proven formally.

9.2 Formal Verification Techniques Applied to SoCs and SystemC/TLM

Many works targeting the formal verification of SoC models written in SystemC share the same approach: first the SystemC program is translated into some formal language, such as finite state machines [22], Petri nets [23], Promela [19], or labeled Kripke structures [24]; next the formal model is verified with existing tools. The first step implies abstractions, as formal languages are in general less expressive than full C++. The tool LusSy [25] is able to automatically translate and abstract TL models to synchronous automata with variables [25], for which numerous symbolic verification tools exist. LusSy works on small SystemC models, but in order to scale up, more abstraction is required (this should be automatic; a *manual* abstraction of SystemC programs into some formal language is much too error prone). Partial order reduction techniques should be investigated too (the encoding of [19] does not allow efficient partial order reductions).

The approach described in the present paper is different since we work directly on the SystemC program. It is intermediate between testing and formal verification: we can prove a property for a given set of data, but we cannot prove it for any data. As this approach scales better than verification techniques, it provides a high level of confidence in properties which are too complex to be checked by formal verification techniques. Furthermore, our approach avoids the problem of relating a formal model with the source code, and does not generate false errors (i.e., all the errors detected are present in the TL model).

9.3 Algorithms for Partial-Order Reduction

Our dynamic partial order algorithm GS presented in section 6 is functionally equivalent to the algorithm of [3]: given an acyclic program with fixed data, it covers the space of the valid schedulings such that all local errors and deadlocks are found. Since these algorithms have not been implemented for a common language, there is little experimental data to compare their efficiency. According to the `indexer` benchmark, our algorithm seems as efficient as the algorithm of [3] combined with the sleep set technique of VeriSoft [26]. Both algorithms generate more schedulings than equivalence classes on some particular cases but it does not happen on the same particular cases; examples are available in [6]. More recently, [27, 28] have studied the combination of static analysis and dynamic partial order reduction for the verification of SystemC programs.

The algorithm GT of section 7 extends the concept of dynamic partial order reduction to cover the valid timings in addition to the valid schedulings. We have not found any other extension of dynamic partial order reduction for programs with loose timing annotations. Combining partial order reduction and linear programming for the validation of programs with bounded delays has already been investigated in [29]. They run the formal verifier VINAS-P on a program with bounded delays to get test cases which exhibit "failures". Next, for each failure trace, they generate a system of linear constraints and solve it using an integer linear programming solver, in order to deduce a new timing constraint which prevents this failure. Partial order reduction is used during the formal verification step. Compared to our work, they use static analysis and so static partial order reduction, whereas our approach is based on simulations and dynamic partial order reduction. Moreover, in their work, linear programming is not used to validate the program, but to generate a correct version of the program (i.e., linear programming does not help them in finding the failures).

More recent tools [30, 31] are able to generate relevant sets of data in addition to schedulings, or to manage cyclic programs [32]; integrating these techniques in our tool should be helpful for the validation of TL models.

10 Conclusion and Further Work

10.1 Summary

In this paper, we address the problem of validating functional properties of SoCs, given models of these SoCs at a very high level of abstraction. The problem can be split into two parts: first, one must exploit the characteristics of the modeling language in such a way that the high level models are *faithful* with respect to the (not yet existing) real system; second, the high level models should be explored exhaustively, for a given test scenario.

For the first problem, we explained that the high level models have to be non-deterministic, and we introduced the notion of *loose* timing, which can be used to build models whose possible behaviors are reasonable supersets of the possible behaviors of the final system. Since the faithfulness of a model cannot

be expressed formally, we then assume that guidelines in SystemC are used to guarantee faithfulness.

The second problem is solved by using partial order reduction techniques. We presented a method to explore the set of valid schedulings of a SystemC program and a given data input. Next, we described an extension for the exploration of valid timings. Exploring alternative timings may reveal more synchronization errors such as deadlocks, data-races, or violations of specified temporal constraints. We work directly on the program so all errors found are true errors and not false warnings. The conjoint use of dynamic partial order reduction and linear programming avoids redundant simulations of the system under test. As a result, we are now able to increase the test coverage of real size SoC models. The current implementation is already efficient enough to cover exhaustively small timing variations (about 20%) of medium size SoC models, or parts of full big SoCs.

As a consequence of these two points, if we cover all the specified behaviors of the model when testing a property P, P will hold for any particular implementation of the physical chip. In our context, this enables development of robust embedded software.

10.2 Possible Improvements

The possible improvements can be organized into four categories: implementation improvements, new algorithms, refinements of the dependency relation, and restrictions to a sub-problem.

The current implementation of the prototype is sequential. However, we could improve the validation time by distributing the validation on many processors or computers. Indeed, each time the trace analyzer generates several execution directives, each of them can be treated independently of the others. For large case studies, the improvement factor can approach the number of available computers or processors. A distributed version of the algorithm of [3] has already been presented in [33]. Another possible improvement of the implementation is to use backtrack points to avoid restarting each execution from the beginning, as in VeriSoft [26].

The algorithm presented in this article can generate many equivalent schedulings for a given dependency relation. Like the algorithm of [3], it is not optimal. Examples can be found in [6] (section 7.1). [3] shows that combining the sleep set technique [15] with dynamic partial order reduction can avoid some redundant executions. We should investigate using sleep sets with our algorithm, but the improvement we can obtain is small. Indeed, non-optimal cases are already very rare.

On the other hand, refining the dependency relation has already given significant results. We gave in section 8.4 the example of the persistent events. Other synchronization structures should be studied, such as event queues and hash tables.

Dynamic partial order reduction enables detection of both local errors (assertion failures) and deadlocks. Conversely, net unfolding techniques [34] search only for local errors. Not searching for deadlocks allows to reduce the number

of visited states. Unfolding algorithms exist for Petri nets and for synchronous products of automata [35]. However, it is not clear how these algorithms can be implemented for languages as expressive as SystemC.

10.3 Dependency Analysis for the Simulator Parallelization

The SystemC simulator has been built to run on only one processor. One way to accelerate simulations is to parallelize the SystemC simulator in order to take advantage of multiprocessor machines. Of course, if we have many simulations to run, it is easier to distribute the simulations. Here, we suppose that the developer wants to run only one simulation. The idea is to distribute the SystemC processes on as many operating system processes as there are processors available on the machine that runs the simulator.

A parallel SystemC simulator still has to respect the specification. In particular, the SystemC processes must still behave as if they were run on a non-preemptive scheduler. For example, we must not interleave an instruction `assert(x!=42)` inside an atomic SystemC transition "`x=42; y=(x); x=0; wait();`".

Intuitively, two dependent transitions must not be run in parallel. From a theoretical point of view, an execution d on a parallel simulator can be represented by a partial order "$<_d$", such that $p_{i,d} <_d q_{j,d}$ if and only if transition $p_{i,d}$ ended before transition $q_{j,d}$ started. A parallel simulator is valid only if for any parallel execution d, there exists a valid scheduling u such that $p_{i,u} \prec_u q_{j,u}$ implies $p_{i,d} <_d q_{j,d}$ for any $(p, i, q, j) \in P \times \mathbb{N} \times P \times \mathbb{N}$. Note that the definition of a valid dependency relation has to be slightly adapted to take into account all schedulings allowed by the operating system scheduler.

Consequently, a valid parallel scheduler can elect a transition only if it is independent of all already running transitions. A dynamic analysis is not applicable in this context since the dependency relation must be known before the scheduler elects a new transition. [36] is based on the assumption that two transitions are independent if they start from two different components. Unfortunately, this assumption does not hold for SystemC models using the TLM library. We have worked with Yussef Bouzouzou and Pascal Raymond on a parallel scheduler and a static analysis tool to allow valid parallelization for TL model simulations.

10.4 Other Applications and Extensions

We mentioned in the introduction that it is quite likely that the two problems we identified for SystemC/TLM models of systems-on-a-chip are also present in other modeling and simulation contexts.

In a number of industrial contexts, validating the embedded software early, using a virtual prototype of the execution platform (hardware plus operating system or middleware layers), is a crucial problem. The degree of parallelism that has to be modeled is far greater then the number of processors or machines that can be used for the simulation of a virtual prototype. For instance, if this virtual prototyping approach is to be applied in a avionics context, it may be

the case that the execution platform to be modeled contains several hundreds of processors, several buses, and several dedicated devices. If we think of sensor networks, a useful virtual prototype for the development of an application should model the behavior of several thousands of sensor nodes. Hence it is always the case that the available parallelism for the simulation is far smaller than the physical parallelism to be modeled.

Consequently, the simulation techniques have to rely on *schedulers*, and the coverage problem for which we proposed a solution does occur.

We think that our informal study of the faithfulness notion, together with the techniques developed for full coverage simulations are applicable to other contexts.

References

1. Ghenassia, F., ed.: Transaction-Level Modeling with SystemC. TLM Concepts and Applications for Embedded Systems. Springer (2005) ISBN 0-387-26232-6.
2. Herrera, F., Villar, E.: Extension of the SystemC kernel for simulation coverage improvement of system-level concurrent specifications. In: FDL'06: Forum on Specification & Design Languages. (2006)
3. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM Press (2005) 110–121
4. Helmstetter, C., Maraninchi, F., Maillet-Contoz, L., Moy, M.: Automatic generation of schedulings for improving the test coverage of systems-on-a-chip. FMCAD (2006) 171–178
5. Helmstetter, C., Maraninchi, F., Maillet-Contoz, L.: Test coverage for loose timing annotations. In: 11th International Workshop on Formal Methods for Industrial Critical Systems, Springer-Verlag (2006)
6. Helmstetter, C.: Validation de modèles de systèmes sur puce en présence d'ordonnancements indéterministes et de temps imprécis. PhD thesis, INPG, Grenoble, France (2007)
7. Open SystemC Initiative: SystemC v2.1 Language Reference Manual (IEEE Std 1666-2005). (2005) `http://www.systemc.org/`.
8. Open SystemC Initiative: OSCI SystemC TLM 2.0, draft 1 for public review (2006) `http://www.systemc.org/web/sitedocs/TLM_2_0.html`.
9. Arvind, Shen, X.: Using term rewriting systems to design and verify processors. IEEE Micro **19** (1999) 36–46
10. L. A. Kunzle, R. Valette, B.P.C.: Temporal reasoning in fuzzy time Petri nets. Technical Report 98073, LAAS Toulouse (1998)
11. Merlin, P., Farber, D.: Recoverability of communication protocols–implications of a theoretical study. Communications, IEEE Transactions on [legacy, pre - 1988] **24** (1976) 1036–1043
12. Mazurkiewicz, A.: Trace theory. In: Advances in Petri nets 1986, part II on Petri nets: applications and relationships to other models of concurrency, New York, NY, USA, Springer-Verlag New York, Inc. (1987) 279–324
13. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21** (1978) 558–565

14. Katz, S., Peled, D.: Defining conditional independence using collapses. Theoretical Computer Science **101** (1992) 337–359
15. Godefroid, P.: Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem. Volume 1032. Springer-Verlag Inc., New York, NY, USA (1996)
16. Clarisó, R., Cortadella, J.: The octahedron abstract domain. In Giacobazzi, R., ed.: Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings. Volume 3148 of Lecture Notes in Computer Science., Springer (2004) 312–327
17. Miné, A.: The octagon abstract domain. In: WCRE. (2001) 310
18. Moy, M., Maraninchi, F., Maillet-Contoz, L.: Pinapa: An extraction tool for SystemC descriptions of systems-on-a-chip. In: EMSOFT. (2005)
19. Traulsen, C., Cornet, J., Moy, M., Maraninchi, F.: A SystemC/TLM semantics in Promela and its possible applications. In: 14th Workshop on Model Checking Software SPIN. (2007)
20. Helmstetter, C., Ponsini, O.: A comparison of two SystemC/TLM semantics for formal verification. In: MEMOCODE. (2008)
21. Vardi, M.Y.: Formal techniques for SystemC verification. In: DAC'07: Proceedings of the 44th annual conference on Design automation. (2007) Position Paper.
22. Niemann, B., Haubelt, C.: Formalizing TLM with communicating state machines. In: FDL'06: Forum on Specification & Design Languages. (2006) 285–292
23. Karlsson, D., Eles, P., Peng, Z.: Formal verification of SystemC designs using a Petri-net based representation. In: DATE '06: Proceedings of the conference on Design, automation and test in Europe, 3001 Leuven, Belgium, Belgium, European Design and Automation Association (2006) 1228–1233
24. Kroening, D., Sharygina, N.: Formal verification of SystemC by automatic hardware/software partitioning. In: Proceedings of MEMOCODE 2005, IEEE (2005) 101–110
25. Moy, M., Maraninchi, F., Maillet-Contoz, L.: LusSy: an open tool for the analysis of systems-on-a-chip at the transaction level. Design Automation for Embedded Systems (2006) special issue on SystemC-based systems.
26. Godefroid, P.: Software model checking: The verisoft approach. Formal Methods in System Design **26** (2005) 77–101
27. Kundu, S., Ganai, M., Gupta, R.: Partial order reduction for scalable testing of SystemC TLM designs. In: DAC '08: Proceedings of the 45th annual conference on Design automation, New York, NY, USA, ACM (2008) 936–941
28. Blanc, N., Kroening, D.: Race analysis for SystemC using model checking. In: Proceedings of ICCAD 2008, IEEE (2008) 356–363
29. Yoneda, T., Kitai, T., Myers, C.J.: Automatic derivation of timing constraints by failure analysis. In: CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification, London, UK, Springer-Verlag (2002) 195–208
30. Godefroid, P., Klarlund, N., Sen, K.: Dart: directed automated random testing. In: PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, New York, NY, USA, ACM Press (2005) 213–223
31. Godefroid, P.: Compositional dynamic test generation. In: POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM (2007) 47–54
32. Yi, X., Wang, J., Yang, X.: Stateful dynamic partial-order reduction. In: ICFEM. Volume 4260 of Lecture Notes in Computer Science., Springer (2006) 149–167

33. Yang, Y., Chen, X., Gopalakrishnan, G., Kirby, R.M.: Distributed dynamic partial order reduction based verification of threaded software. In: SPIN. Volume 4595 of Lecture Notes in Computer Science., Springer (2007) 58–75
34. McMillan, K.L.: Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In: CAV '92: Proceedings of the Fourth International Workshop on Computer Aided Verification, London, UK, Springer-Verlag (1992) 164–177
35. Esparza, J., Römer, S.: An unfolding algorithm for synchronous products of transition systems. In: Proc. of CONCUR'99. Number 1664 in Lecture Notes in Computer Science, Springer-Verlag (1999) 2–20
36. Chopard, B., Combes, P., Zory, J.: A conservative approach to SystemC parallelization. In: International Conference on Computational Science (4). (2006) 653–660