

# Test Coverage for Loose Timing Annotations

C. Helmstetter<sup>1,2</sup>, F. Maraninchi<sup>1</sup>, and L. Maillet-Contoz<sup>2</sup>

<sup>1</sup> Verimag, Centre équation - 2, avenue de Vignate, 38610 GIÈRES — France

<sup>2</sup> STMicroelectronics, HPC, System Platform Group.  
850 rue Jean Monnet, 38920 CROLLES — France

**Abstract.** The design flow of systems-on-a-chip (SoCs) identifies several abstraction levels higher than the Register-Transfer-Level that constitutes the input of the synthesis tools. These levels are called *transactional*, because systems are described as asynchronous parallel activities communicating by transactions. The most abstract transactional model is purely functional. The following model in the design flow is annotated with some timing information on the duration of the main components, that serves for performance evaluation. The timing annotations are included as special `wait` instructions, but since the timing information is imprecise, it should not result in additional synchronizations. We would like the functional properties of the system to be independent of the precise timing. In previous work [1], we showed how to adapt dynamic partial order reduction techniques to functional models of SoCs written in SystemC, in order to guarantee that functional properties are scheduler-independent. In this paper, we extend this work to *timed* systems with bounded delays, in order to guarantee timing-independence. The idea is to generate a set of executions that covers small variations of the timing annotations.

## 1 Introduction

The Register Transfer Level (RTL) used to be the entry point of the design flow of hardware systems, but the simulation environments for such models do not scale up well. Developing and debugging embedded software for these low level models before getting the physical chip from the factory is no longer possible at a reasonable cost. New abstraction levels, such as the *Transaction Level Model (TLM)* [2], have emerged. The TLM approach uses a component-based approach, in which hardware blocks are modules communicating with so-called *transactions*. The TLM models are used for early development of the embedded software, because the high level of abstraction allows a fast simulation. SystemC is a C++ library used for the description of SoCs at different levels of abstraction, from cycle accurate to purely functional models. It comes with a simulation environment, and is becoming a *de facto* standard.

As TLM models appear first in the design flow, they become reference models for SoCs. In particular, the software that is validated with the TLM model should remain unchanged in the final SoC. The TLM abstraction level comes with new synchronization mechanisms that often make existing methods for RTL

validation inapplicable. In particular, recent TLM models do not have clocks at all. In this paper, we concentrate on testing methods for SoCs written in SystemC.

The current industrial methodology for testing SoCs in SystemC is the following. First, we identify what we want to test (the *System Under Test*, or SUT), which is usually an open system. We make it closed by plugging *input generators* and a *result checker*, called *oracle*. SCV [3] is a testing tool for SystemC. It helps in writing input generators by providing C++ macros for expressing constraints: `SCV_CONSTRAINT((addr())>10 && addr()< 50) || (addr())>=2 && addr()<= 5);` is an SCV constraint for which the SCV solver will generate random values of `addr` satisfying it. In most existing approaches, the SUT writes in memory, and the oracle consists in comparing the final state of the SUT memory to a reference memory. As usual, the main difficulty is to get a good quality test suite, i.e., a test suite that does not omit *useful* tests (that may reveal a bug) and at the same time avoids *redundant* tests (that can expose the same bugs) as much as possible. Specman [4] is a commercial alternative of SCV which uses the *e* language for describing the constraints.

### 1.1 Partial Order Reduction Techniques for Scheduler-Independence

In [1], we have presented an automatic technique for the exploration of schedulings in the case of SystemC. It is an adaptation and application of the method for *dynamic* partial order reduction presented in [5]. We assume that the choice of relevant data for the testing phase has already been done: we consider a SoC written in SystemC, including the data generator and the oracle. For each of the test data, the system has to be *run*, with a particular *implementation* of the scheduler. Since the *specification* of the scheduler is non-deterministic, this means that the execution of tests may hide bugs that would have appeared with another valid implementation of the scheduler. Moreover, the scheduling is due to the simulation engine only, and is unlikely to represent anything concrete on the final SoC where we have true parallelism. We would like the SoC description, and in particular the embedded software, to be scheduler-independent. Exploring alternative schedulings is a way of validating this property.

Our tool is based on forking executions: we start executing the system for a given data-input, and as soon as we suspect that several scheduler choices could cause distinct behaviors, we fork the execution. We use an *approximate* criterion to decide whether to fork executions. The idea is to look at the actions performed by the processes, in order to guess whether a change in their order (as what would be produced by distinct scheduler choices) could affect the final state. This criterion is approximate in the following sense: we may distinguish between executions that in fact lead to the same final state; but we cannot consider as equivalent two executions that lead to distinct final states. The result is a complete, but not always minimal, exploration of the scheduling choices for the whole data-input.

## 1.2 The Hierarchy of TLM Models

There are several levels of transactional models. The more abstract transactional model is purely *functional*. The following model in the design flow is enriched with some timing information on the duration of the main components, that may serve for performance evaluation during simulation. This timing information is quite imprecise: it may be given by previous measures on existing IPs (IP stands for “*Intellectual Property*”; an IP block is a reusable hardware component). For instance, we may have approximate values for the time it takes to write an image in memory. Note that this kind of loose timing is still very far from the precision of cycle-accurate models, and this is why timed transactional models are interesting: they simulate much faster than cycle-accurate models, but they can already give some hints on the performance of the SoC.

Practically, a SystemC description annotated by timings uses a special instruction `wait (duration)`. The interpretation of this instruction by the simulation engine simulates the amount of time taken by the components. When executing such a SoC description enriched with timings, the SystemC execution engine has to take precise values of the timings. There is a risk of producing spurious synchronizations by interpreting the timings too strictly. In other words, the embedded software will be more robust if it works correctly for slightly distinct timings. It is therefore useful to explore alternative timings during testing. It can be done by choosing a timing randomly within an interval, at execution time. Existing industrial approaches use a new instruction `lwait (duration, delta)`, telling the execution engine to draw a value in the interval `[duration - delta, duration + delta]`. If the instruction appears within a loop, a new value is drawn for each execution of the instruction. However, this slows the simulations without guaranteeing that interesting cases are explored.

## 1.3 Contributions and Structure of the Paper

Ensuring *timing-independence* can also be done in a more systematic way, by generating exactly the set of timings that yield different behaviors of the SoC. In this paper, we generalize the approach of [1] in order to generate alternative schedulings and alternative timings for a given data input. The result is of the same kind: we obtain a complete but not always minimal set of alternative executions, for a given data input. The idea is that, if the software works well for all these alternative executions, it is more robust. This is our notion of scheduler and timing independence.

The paper is structured as follows: section 2 presents an overview of SystemC, and some examples for illustrating the influence of the scheduling and the presence of loose timings. Section 3 recalls the results of [1] and section 4 describes our new algorithm for models with loose timings. We present our implementation and its evaluation in section 5, related work in section 6, and we conclude with section 7.

## 2 SystemC, Scheduling Problems, and Loose Timings

A TLM 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. 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 figure 1-(a). Simulations of a SystemC model look like sequences of *evaluation phases* (EV). Signals *update phase* (UP) and *time elapse* (TE) separate them (see figure 1-(b)).

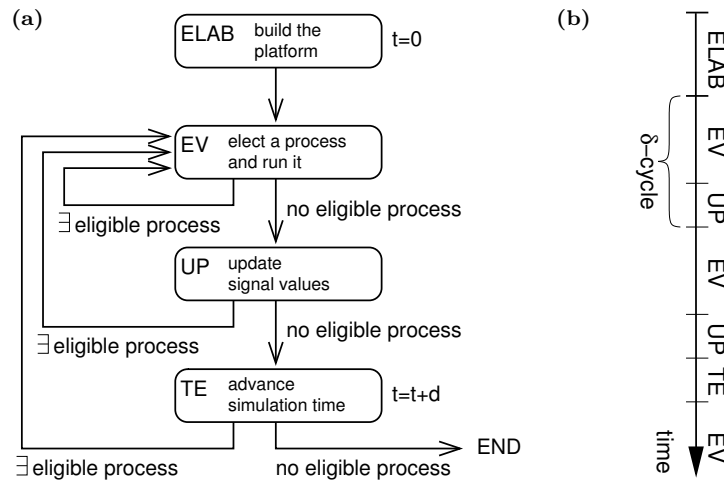


Fig. 1. (a) Automaton of the SystemC Scheduler; (b) Diagram of an execution.

### 2.1 The SystemC Scheduler

The SystemC Language Reference Manual [6] 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* instruction. 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 is no more eligible process, signals values are updated (**UP**) and  $\delta$ -delayed notifications are triggered, which can wake up processes. A  $\delta$ -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,  $\delta$ -delayed or time-delayed. Processes can thus become eligible at any of the three steps EV, UP or TE. Besides events, processes can also communicate using shared variables, and higher level structures built with these two primitives.

## 2.2 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. 2.** The foo example

```

void top::P()
    as in example foo
void top::Q()
    as in example foo
    |
void top::R() {
    wait(20);
}

```

**Fig. 3.** The foobar example

To illustrate possible consequences of scheduling choices, let us introduce two small examples of SystemC programs. Figure 2 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 “`wait(e)`” statement has been executed before any notification of event  $e$ . Since events are not persistent in SystemC, process P has not been woken up. It is a particular form of *deadlock*.

It is useful to test all executions of the `foo` example because they lead to different final states. But consider now the `foobar` example defined in figure 3. `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 for scheduler-independence would generate only 3 executions, one for each final state (or equivalence class).

### 2.3 Examples with Loose Durations

Figure 4 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  $3-d_1$  and  $3+d_1$ , a value for  $t_2$  between  $40-d_2$  and  $40+d_2$ , etc.

```

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

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

```

Fig. 4. The `foochi` example

If  $d_1 = d_2 = d_3 = d_4 = 0$ , 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):  $P_1; Q_1$  or  $Q_1; P_1$  followed by  $[t \xrightarrow{+3} 3]; P_2; [t \xrightarrow{+3} 6]; Q_2; P_3; [t \xrightarrow{+24} 30]; Q_3; [t \xrightarrow{+16} 46]; P_4$ .  $P_1$  and  $Q_1$  occur at  $T = 0ns$ ,  $P_2$  at  $T = 3ns$ ,  $Q_2$  and  $P_3$  at  $T = 6ns$ . Next  $Q_3$  runs at  $T = 24 + 6 = 30ns$ . At last, the string “Ok” is displayed by  $P_4$  at  $T = 6 + 40 = 46ns$ .

Giving non-null values to the  $d_i$  allows to test the robustness of the program. If we take  $d_1 = d_2 = d_3 = d_4 = 2$ , then it is possible to permute the wait 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  $Q_3$  and  $P_4$ . If we increase  $d_2$  (resp.  $d_4$ ) to 10 (resp. 6), then  $Q_3$  and  $P_4$  may occur at the same time  $T = 6 + 30 = 36ns$  ( $30 = 24 + 6 = 40 - 10$ ). Next, playing with the indeterminism of the scheduler allows to execute  $P_4$  before  $Q_3$ . We have found the two errors of the `foo` example again. The algorithm we describe in this paper generates timings and schedulings automatically, in order to find the executions that lead to these errors.

## 3 Relationships for Partial Order Reduction Techniques

In the whole section, the 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. We have to generate a relevant set of schedulings and timings for this data.

### 3.1 Representation of the SUTD

When data is fixed, a SUT execution is entirely defined by its scheduling and its concrete timing. A scheduling is entirely defined by an element of  $\mathcal{P}^*$  where  $\mathcal{P}$  is the set of process identifiers. Not all the elements of  $\mathcal{P}^*$  represent possible schedulings of the SUTD (because of the synchronization and timing constraints between processes). With each `lwait(D,d)` instruction present in the source code, we associate an identifier  $\omega \in \Omega$ ; we note  $B(\omega)$  (B stands for “Bounds”) the interval  $[D-d, D+d]$  and  $\#_u(\omega)$  the number of times an execution of  $\omega$  occurs

in a scheduling  $u$ . A timing  $T$  is a function from pairs  $(\omega, n) \in \Omega \times [1..\#_u(\omega)]$  to durations  $d$ .  $T(\omega, n) = d$  means that we wait for a duration  $d$  when we execute the instruction identified by  $\omega$  for the  $n$ -th time. The timing  $T$  is valid if and only if  $\forall(\omega, n) \in \Omega \times [1..\#_u(\omega)], T(\omega, n) \in B(\omega)$ .

We call *transition* one execution of one process in a particular scheduling. Each transition of a scheduling is identified by its process identifier indexed by the occurrence number of this process identifier in the scheduling. For example, in the scheduling  $pqp$  there are 3 transitions:  $p_1$ ,  $q_1$  and  $p_2$ , in that order. For a particular execution with a specified timing, the *date* of a transition is the value of the variable  $t$  of the scheduler (figure 1-(a)) when the transition occurs.

We will use letters  $p, q, r$  to denote processes,  $p_i, q_j, \dots$  to denote transitions and  $u, v, \dots$  to denote sub-sequences of schedulings. Indexes will be omitted when obvious by context.

### 3.2 Relationships

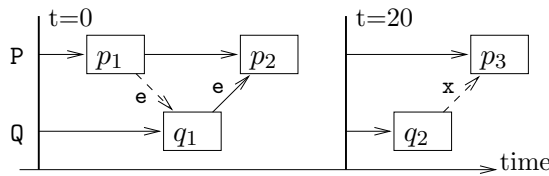
We recall some standard notions from the literature on partial order reduction techniques, that we will use for both scheduling and timing generation.

**Dependent and equivalent transitions** The theory of partial order reduction relies on the definition of *dependent* transitions [7]. Let  $u$  be a valid scheduling, two transitions  $p_i$  and  $q_j$  are independent if not any of them has been enabled by the other, and if permuting them gives a new valid scheduling which still leads to the same final state. In all other cases, we say that  $p_i$  and  $q_j$  are *dependent*. Note that it is correct because, in SystemC, an enabled process cannot be disabled without being executed. We note  $D$  the set of all pairs of dependent transitions.

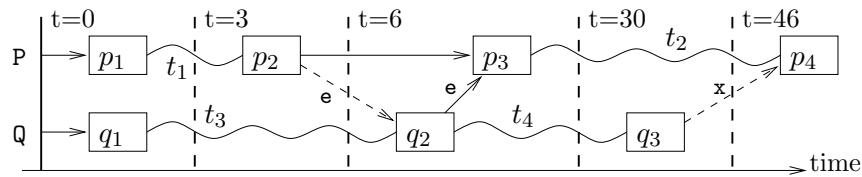
Two schedulings  $u$  and  $v$  are *equivalent*, noted  $u \equiv v$ , if and only if we can transform one into the other by successive permutations of independent transitions. As a consequence of the definition of the dependency relationship  $D$ , two equivalent schedulings lead to the same final state. In our testing approach for SystemC, we include the output checker into the SUT, which means that the detection of an error corresponds to a particular final state. Hence generating one scheduling of each equivalence class allows to detect all errors.

**Causally ordered and permutable transitions** Consider a scheduling  $u$ : we note  $p_i <_u q_j$  if the transition  $p_i$  (the  $i$ -th execution of process  $p$ ) occurs before the transition  $q_j$  (the  $j$ -th execution of process  $q$ ) in  $u$ . We note  $p_i \prec_u q_j$  and say that  $p_i$  and  $q_j$  are *causally ordered*, if we have  $p_i <_v q_j$  for any scheduling  $v$  equivalent to  $u$ . In other words,  $p_i$  and  $q_j$  are causally ordered if we cannot permute them without permuting dependent transitions. Unlike the causal relationship, the *permutability relationship* is not a partial order. Two transitions are permutable if they can be permuted without permuting **other** dependent transitions. We note  $P$  the set of permutable transitions. The transitions  $p_i$  and  $q_j$  are *permutable* in the valid scheduling  $u = u_1 p_i u_2 q_j u_3$ , noted  $(p_i, q_j) \in P$ , if and only if:  $\exists v_1, v_2$  such that  $u_1 v_1 p_i q_j v_2 \equiv u_1 p_i u_2 q_j u_3$  and  $u_1 v_1 q_j$  is a valid scheduling.

**Dynamic Dependency Graph (DDG)** The DDG represents the synchronizations of a particular scheduling. Fig. 5-(a) represents the scheduling  $P;Q;P;Q;P$ , denoted  $p_1q_1p_2q_2p_3$ , of the `foo` program of Fig. 2, and Fig. 5-(b) represents 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 curved lines represent loose durations. Each box is a process transition. Arrows between boxes indicate that the two transitions are causally ordered; we draw dashed arrows if the transitions are permutable, 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.



(a) `foo` with scheduling  $p_1q_1p_2q_2p_3$



(b) `foochi` with scheduling  $p_1q_1p_2p_3q_3p_4$   
and timing:  $t_1 \mapsto 3, t_2 \mapsto 40, t_3 \mapsto 6, t_4 \mapsto 24$

**Fig. 5.** Dynamic Dependency Graphs

*Computation of the relationships* In practice, we can only compute an *approximation* of the dependency relationship: two independent transitions may be considered as dependent, but two dependent transitions are always considered as dependent. Consequently, the only risk is to generate useless schedulings.

We compute the dependency relationship for each new generated scheduling. Doing multiple dynamic computations is more precise than one static computation. For example, for a code like `Tab[h]=42` we know exactly which element of `Tab` is accessed, and whether the new value is different from the old one.

Two transitions are dependent if some reasons prevent their permutation, or else if they contain non-commutative actions on the same shared object (for example: `wait(e)` and `notify(e)`, or `x=0` and `x=x+1`). For the causal order and the relationship  $P$ , we have to compute a transitive closure. The principles of these algorithms are available in [5], or in [1] for SystemC-specific concerns. Here, we consider that  $D$  and  $P$  are computed without taking temporal constraints into account.

### 3.3 Generation of Schedulings

In this section, we rewrite the algorithm of [1] defined for the automatic generation of schedulings, in such a way that the generalization to timing generation becomes possible. The algorithm of [1] works on any SUTD with only fixed delays. First, we execute the SUTD with a random scheduling. Next, for each executed scheduling, we generate a new scheduling for each pair of dependent and permutable transitions. Figure 6 gives a definition of the main algorithm.

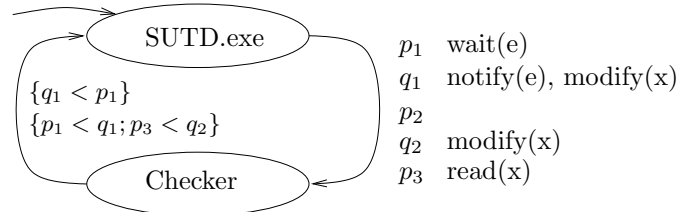
```

 $G_S$ (constraint set  $C$ ):      //initial call:  $G_S(\emptyset)$ 
  execute the SUTD according to  $C$ ;
   $u$  = scheduling of the above execution;
  for all transitions  $p_i$  and  $q_j$  of  $u$  with  $p_i <_u q_j$  such that:
    ( $p_i, q_j$ )  $\in D \cap P$  and
     $date(p_i) = date(q_j)$  and //temporal constraints are treated here (*)
     $\exists v, v \models C \wedge q_j <_v p_i$  do
       $G_S(C \cup \text{"}q_j < p_i\text{"})$ ; //constraint to be satisfied by new schedulings
       $C = C \cup \text{"}p_i < q_j\text{"}$ ; //constraint satisfied by the current scheduling

```

**Fig. 6.** Main algorithm for the generation of schedulings

We generate each scheduling in two steps. First, we build a set of *scheduling constraints* of the form “ $p_i < q_j$ ”. A constraint “ $p_i < q_j$ ” is satisfied by a scheduling  $u$  if and only if the  $j$ -th occurrence of  $q$  does not occur before the  $i$ -th occurrence of  $p$  (formally:  $q_j \in u \Rightarrow p_i \in u \wedge p_i <_u q_j$ ). The scheduling  $u$  satisfies a set of constraints  $C$  (noted  $u \models C$ ) if and only if it satisfies all constraints of  $C$ . Next, we give this constraint set to a patched scheduler that elects processes according to the given constraints. Each new generated scheduling is more constrained than its father scheduling. Consequently, there are fewer and fewer new schedulings at each iteration. When the checker does not generate any new scheduling, we have a complete test suite. If we execute this algorithm until completion, we get at least one scheduling for each equivalence class.



**Fig. 7.** First iteration of the analysis for the `foo` example

Figure 7 describes the first iteration of our tool on the `foo` example. The first execution activates processes  $p$  and  $q$  in the order  $p_1q_1p_2q_2p_3$ . The checker generates two new sets of constraints. One to permute  $p_1$  and  $q_1$  (unordered accesses to event  $e$ , first dashed arrow of figure 5-(a)) and the other to permute  $p_3$  and  $q_2$  (unordered accesses to shared variable  $x$ , second dashed arrow of figure 5-(a)). Following iterations do not generate other schedulings and we get at last 3 schedulings.

## 4 Conjoint Generation of Scheduling and Timings

### 4.1 The SystemC Models We Consider

First, we need to make the context of our work more precise. In this work, we restrict to SystemC programs whose executions have only one  $\delta$ -cycle between two “time-elapse” phases. Indeed, the semantics of  $\delta$ -cycle delays for abstract models with loose durations is unclear and such delays should not be used in timed TLM models. Moreover, for simplicity reasons, we do not consider delayed notifications. Finally, we consider that the global date (variable  $t$  of Figure 1-(a)) 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 TLM models, where the timing annotations are added to a functional model, for performance evaluation only. We discuss this topic in the conclusion.

### 4.2 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 it exists a concrete timing which allows their permutation. If such timings exist, we have to choose one and to re-execute the SUTD with it. In the algorithm presented in section 3.3 above, it is the only point which 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  $T(\omega, i)$  items. 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 `lwait` instruction,  $S$  defines an octahedron [8] (all variable coefficients are in  $\{-1, 0, 1\}$ ) but not an octagon [9] (a constraint may use more than two variables).

### 4.3 The Temporal Constraints

There are two sorts of temporal constraints. First, the solution must correspond to valid timings. So for all  $(\omega, i) \in \Omega \times [1..\#(\omega)]$ , we add the two constraints  $\inf(B(\omega)) < T(\omega, i)$  and  $T(\omega, i) < \sup(B(\omega))$ . 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  $p_i$ , we associate a *symbolic date* noted  $sdate(p_i)$ . A symbolic date is a sum of variables  $T(\omega, i)$  and constants. We compute the symbolic date of a transition  $p_i$  as follows:

1. if  $p_i$  follows a **wait** with loose duration ( $p_{i-1}$  ended by a call to **lwait**), then:  $sdate(p_i) = sdate(p_{i-1}) + T(\omega, n)$  where  $\omega$  is the identifier of this **lwait** instruction and  $n$  its occurrence number.
2. if  $p_i$  follows a **wait** with fixed duration ( $p_{i-1}$  ended by a call to **wait(k)**), then:  $sdate(p_i) = sdate(p_{i-1}) + k$ ;
3. if  $p_i$  as been enabled by an immediate notification from transition  $q_j$ , then:  $sdate(p_i) = sdate(q_j)$ ;
4. if  $p$  is initially eligible, then  $p_1 = 0$ .

We illustrate these rules on the example **foochi** with  $u = p_1q_1p_2q_2p_3q_3p_4$ . 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, we build the associated temporal constraint as follows: we first evaluate  $sdate(p_i)$  and  $sdate(q_j)$ , which yields two expressions  $e_1$  and  $e_2$ ; we then add to  $S$  the constraint “ $e_1 \leq e_2$ ”.

#### 4.4 The Algorithm

Figure 8 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. On line (1), the timing  $T$  may be incomplete, i.e., the value for some **lwait** instructions may be unspecified. In this case, the simulation engine is free to choose any value in the given interval. Initially we call  $G_T$  with an empty set of scheduling constraints and an empty timing. Let  $T_u$  be the concrete timing of the current scheduling  $u$ .  $T_u$  is always a solution of the system of linear constraints  $S$ . In general,  $T_u$  is not a solution of the system built on line (2).

```

 $G_T$ (constraint set  $C$ , timing  $T$ ): //initial call:  $G_T(\emptyset, \emptyset)$ 
  execute the SUTD according to  $C$  and  $T$ ; (1)
   $u =$  scheduling of the above execution;
  linear system  $S = []$ ;
  for all  $(\omega, i) \in \Omega \times [1.. \#(\omega)]$  do
     $S = S \bullet (T(\omega, i) \in B(\omega))$ ;
  for all constraint “ $p_i < q_j$ ” of  $C$  do
     $S = S \bullet (sdate(p_i) \leq sdate(q_j))$ ;
  for all transitions  $p_i$  and  $q_j$  of  $u$  with  $p_i <_u q_j$  such that:
     $(p_i, q_j) \in D \cap P$  and
     $\exists v, v \models C \wedge q_j <_v p_i$  do
    if is_feasible( $S \bullet (sdate(q_j) \leq sdate(p_i))$ ) then (2)
       $T' =$  solution_of( $S \bullet (sdate(q_j) \leq sdate(p_i))$ );
       $G_T(C \cup \{q_j < p_i\}, T')$ ; (3)
     $C = C \cup \{p_i < q_j\}$ ;
     $S = S \bullet (sdate(p_i) \leq sdate(q_j))$ ;

```

**Fig. 8.** Main algorithm for the generation of timings

We describe the first call to  $G_T$  on the example `foochi` to illustrate this algorithm. If we ignore the temporal aspects, the analysis of  $u = p_1q_1p_2q_2p_3q_3p_4$  generates two sets of constraints:  $\{q_2 < p_2\}$  and  $\{p_2 < q_2; p_4 < q_3\}$ .

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 in  $t_3 - t_1 \leq 0$ . We must also respect bounds on variables:  $t_1 \in [1, 5]$  and  $t_3 \in [4, 8]$ . We ask a solution to the linear programming library and get the solution  $t_1 = t_3 = 4$ . Finally, we call  $G_T(\{q_2 < p_2\}, \{t_1 = 4, t_3 = 4\})$  (line (3) of the algorithm). These scheduling constraints and this timing lead to the first error of `foochi` cited at 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  $G_T$  again with this set of constraints and this timing as arguments. This leads to the second error of `foochi`.

#### 4.5 Elements for the Correctness of the Algorithm

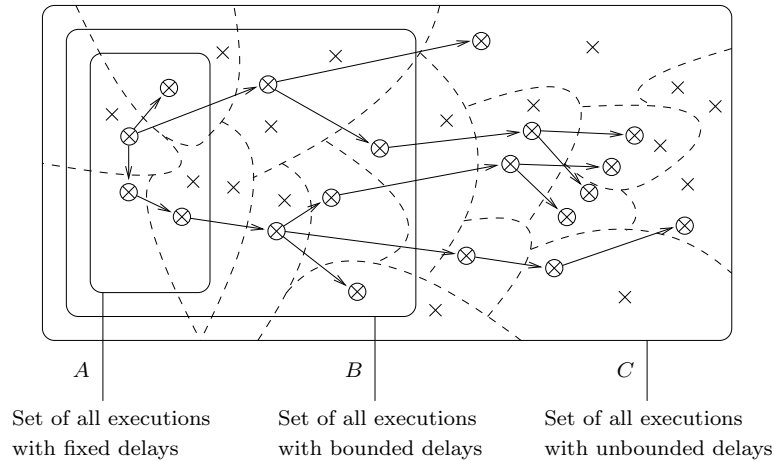
In the general case,  $G_T$  generates **at least one** representant of each equivalence class, as  $G_S$  does. On this example, we have generated one element of each equivalence class. First, we have suppressed the condition “*the transitions  $p_i$  and  $q_j$  are not permutable if  $date(p_i) \neq date(q_j)$ ” (line (\*) of Figure 6). We call  $G'_S$  the algorithm  $G_S$  in which this condition has been suppressed. Running  $G'_S$  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  $G'_S$  and  $G_T$  is that  $G_T$  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  $G_T$  generates all elements of  $G'_S$  that satisfy the temporal constraints. Figure 9 represents the sets of executions generated by  $G_S$ ,  $G'_S$  and  $G_T$ .

## 5 Case Study: The MPEG Decoder System

We have complemented our prototype for  $G_S$  with a prototype for  $G_T$ . Figure 10 gives an overview of this new prototype. We instrument the C++/SystemC source code with the SystemC front-end Pinapa [10] in order to detect the accesses to shared variables dynamically. We have chosen LP\_SOLVE [11] to solve the linear systems.

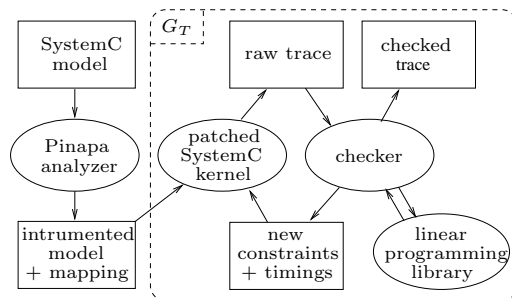
We have evaluated the tool on a small industrial case-study. This system has 5 components: a master, a MPEG decoder, a display, a memory and a bus model. There are about 50 000 lines of code and only 4 processes. This is quite common in the more abstract models found in industry, because there is a lot



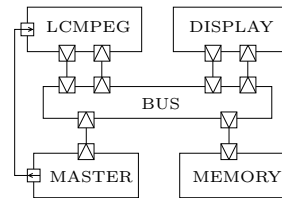
**Fig. 9.** Sets of all executions of the SUTD. The dashed lines delimit the equivalence classes. The surrounded crosses represent generated executions, with arrows from father to children.  $G_S$  returns the surrounded crosses of the set  $A$ ,  $G_T$  those of  $B$  and  $G'_S$  those of  $C$ .

of sequential code, and very few synchronizations. Complete models of SoCs are typically 3 to 6 times bigger than this MPEG decoder. The test is stopped after the third decoded image, which corresponds to 150 transitions. One simulation takes 0.39 s.

First, we run the  $G_S$  prototype on a timed version without loose delays. It generates **128 schedulings in 1 mn 08 s**. No bug is found, which guarantees that this test-case will run correctly on any SystemC implementation. The total time spent splits into 50 s for running the SUTD 128 times and an overhead of 18 s for the additional computations. The experiments have been run on a Pentium 4 cadenced at 2.80 GHz.



**Fig. 10.** The Prototype's Architecture



**Fig. 11.** Architecture of the MPEG decoder system

The  $G_S$  prototype can be used on an untimed version too. This untimed version is obtained by replacing all timed instructions by their corresponding untimed instructions. But the prototype failed to run to completion because the scheduling space to explore is far too large. Indeed, removing time constraints allows a lot of new interleavings. For the untimed version, we estimate the number of relevant schedulings to about  $2^{32}$ . It would take many years to execute them all. Most of this time would be spent exploring unrealistic interleavings.

The prototype of  $G_T$  allows to test bounded-delay versions which are intermediate between the fixed-delay version and the fully untimed version. We replace all instructions `wait(d)` by `lwait(d,d*r)`. The number of valid interleavings increases when the global variable `r` increases. The goal is to validate the SUTD with `r` as big as possible. We succeed in validating this MPEG decoder system with `r = 0.2`. The  $G_T$  prototype generates **3584 schedulings and timings** in **35 mn 11 s**. One must spend 23 mn 18 s to execute this system 3584 times. The overhead is about 11 mn 53 s. Our goal is to validate the system with `r = 0.5` but the first attempts show that our prototype is not fast enough yet.

## 6 Related Work

The idea of interpreting timing annotations in a loose way is quite natural. It was already present in some modeling approaches based on fuzzy time (see, for instance, [12]). However, these approaches are often dedicated to the handling of imprecise functional information, while we focus on non-functional information.

The approach described in [13] has some similarities with ours. 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. Finally, they get new bounds for the delays specified in the program, which avoid failures. Static partial order reduction is used during the formal verification step. Like us, they found that the time spent to solve the generated linear programs is quite small compared with the total time spent. The technique used in our tool differs in two points: first we use *dynamic* POR, second the linear systems are used inside the POR algorithm and not afterwards.

As far as we know, there is no verification tool for SystemC programs with bounded delays yet. However, the tool LusSy [14] is able to translate automatically SystemC programs into synchronous automata for which numerous verification tools exist. Another approach would be to extend LusSy to translate SystemC programs into timed automata which can be verified with tools such as Kronos [15] or Uppaal [16] (this should be automatic; a *manual* translation of SystemC programs into some formal language is too much error prone). The approach described in this paper avoids the problem of relating a formal model with the source code; since it is developed for a testing framework, it scales better than verification techniques.

## 7 Conclusion and Further Work

In previous work, we presented a method to explore the set of valid schedulings of a SystemC program and a given data input. In this paper, we described a generalization to the exploration of valid timings. Exploring alternative timings may reveal more synchronization errors such as dead-locks or data-races, and violations of specified temporal constraints too. 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 allows to avoid redundant simulations of the system under test. As a result, we are now able to increase the test coverage of real size SoC models.

We have implemented this new algorithm. The current prototype is already efficient enough to cover exhaustively small timing variations (about 20%) of medium size SoC models, or parts of full big SoCs. We still have many possible improvements to study. First, using the pre-solve functionality of the LP\_SOLVE library should reduce the overhead due to the computation of timings. Indeed, it seems we perform lots of redundant computations when solving the temporal constraints. Second, we still produce some redundant executions. The dynamic partial order reduction technique is not optimal; we can indeed get two schedulings which are equivalent according to the computed dependency relationship. In [5], P. Godefroid suggests the use of the sleep sets technique to eliminate some of them. In addition, computing a more precise dependency relationship will reduce the number of equivalence classes to cover. The dependency of two transitions depends mainly of the way they communicate. Up to now, we have only considered low level communication items (non-persistent events and shared variables). Higher level communication mechanisms (persistent events, for example) can be globally robust to the scheduler indeterminism although they perform dependent accesses locally. Taking them into account should reduce dramatically the total time spent. With these improvements, we hope to be able to cover wider timing variations, up to 40% or 50%.

Another further work concerns the restrictions imposed on the programs we validate. Currently, we forbid reading of the global date from the processes. For example, the following instruction is not allowed: `if (date()<45) {A} else {B}`, where `date()` returns the current global date. As a consequence of this restriction, the functional behavior of an execution depends only on its scheduling; the timing is only used to know whether the scheduling is valid, and to get an estimation of temporal performances of the final SoC. Instructions as above are not common in the models we have studied, however they might be more frequent or necessary in other domains; therefore extending our tool will be useful. Our idea is the following: first, we add to the scheduling representations virtual transitions of the form  $\chi(t)$  meaning that the global date has just reached  $t$ . Next, we consider that a virtual transition  $\chi(t)$  is: 1) dependent and permutable with all transitions which compare `date()` and  $t$ , and 2) causally ordered with the other virtual transitions. Thus it is possible to treat all expressions of the form `date()<k` without modifying the main algorithm; it could be extended to

multiple clocks with reset instructions but allowing all expressions using `date()` is a harder task.

## References

1. Helmstetter, C., Maraninchi, F., Maillet-Contoz, L., Moy, M.: Automatic generation of schedulings for improving the test coverage of systems-on-a-chip. In: FMCAD, Springer (2006)
2. Ghenassia, F., ed.: Transaction-Level Modeling with SystemC. TLM Concepts and Applications for Embedded Systems. Springer (2005) ISBN 0-387-26232-6.
3. Rose, J., Swan, S.: SCV Randomization (2003)  
[www.testbuilder.net/reports/scv\\_randomization.pdf](http://www.testbuilder.net/reports/scv_randomization.pdf).
4. Kuhn, T., Oppold, T., Winterholer, M., Rosenstiel, W., Edwards, M., Kashai, Y.: A framework for object oriented hardware specification, verification, and synthesis. In: DAC '01: Proceedings of the 38th conference on Design automation, New York, NY, USA, ACM Press (2001) 413–418
5. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: Symposium on Principles of programming languages (POPL), New York, NY, USA, ACM Press (2005) 110–121
6. Open SystemC Initiative: SystemC v2.0.1 Language Reference Manual. (2003)  
<http://www.systemc.org/>.
7. 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
8. 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
9. Miné, A.: The octagon abstract domain. In: WCRE. (2001) 310
10. Moy, M., Maraninchi, F., Maillet-Contoz, L.: Pinapa (2005)  
<http://greensocs.sourceforge.net/pinapa/>.
11. Berkelaar, M., et al.: Lp\_solve (1996)  
<http://www.cs.sunysb.edu/~algorithm/implement/lpsolve/implement.shtml>.
12. L. A. Kunzle, R. Valette, B.P.C.: Temporal reasoning in fuzzy time petri nets. Technical Report 98073, LAAS Toulouse (1998)
13. 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
14. Moy, M., Maraninchi, F., Maillet-Contoz, L.: LusSy: A toolbox for the analysis of systems-on-a-chip at the transactional level. In: International Conference on Application of Concurrency to System Design. (2005)
15. Bozga, M., Daws, C., Maler, O., Olivero, A., Tripakis, S., Yovine, S.: Kronos: A model-checking tool for real-time systems. In: Proc. 1998 Computer-Aided Verification, CAV'98. Volume 1427 of Lecture Notes in Computer Science., Vancouver, Canada, Springer-Verlag (1998)
16. Uppsala and Aalborg Universities: Uppaal (1994-2006)  
<http://www.uppaal.com/>.