Automatic Generation of Schedulings for Improving the Test Coverage of Systems-on-a-Chip

C. Helmstetter*† , F. Maraninchi*, L. Maillet-Contoz† and M. Moy*
*Verimag, Centre équation - 2, avenue de Vignate,
38610 GIÈRES — France
†STMicroelectronics, HPC, System Platform Group.
850 rue Jean Monnet, 38920 CROLLES — France

***Abstract*— SystemC is becoming a de-facto standard for the early simulation of Systems-on-a-chip (SoCs). It is a parallel language with a scheduler. Testing a SoC written in SystemC implies that we *execute* it, for some well chosen data. We are bound to use a particular deterministic implementation of the scheduler, whose specification is *non-deterministic*. Consequently, we may fail to discover bugs that would have appeared using another valid implementation of the scheduler. Current methods for testings SoCs concentrate on the generation of the inputs, and do not address this problem at all. We assume that the selection of relevant data is already done, and we generate several schedulings allowed by the scheduler specification. We use dynamic partial-order reduction techniques to avoid the generation of two schedulings that have the same effect on the system's behavior. Exploring alternative schedulings during testing is a way of guaranteeing that the SoC description, and in particular the embedded software, is scheduler-independent, hence more robust. The technique extends to the exploration of other non-fully specified aspects of SoC descriptions, like timing.**

I. 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)* [1], 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. This new abstraction level comes with new synchronization mechanisms which often make existing methods for RTL validation inapplicable. In particular, recent TLM models do not have clock anymore.

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. Here, 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 [2] 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 that will generate random values of `addr`. 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 [3] is a commercial alternative of SCV which uses the *e* language for describing the constraints.

Contributions and Structure of the paper: 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*, necessarily 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.

We present 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 [4]. This method allows to explore efficiently the states of a system made of parallel processes (given as object code) that execute on a preemptive OS and synchronize with a lock mechanism. We show here that it can be applied to SystemC too. Adaptations are needed because: the SystemC scheduler is not preemptive; SystemC programs use non-persistent event notifications instead of locks; evaluation phases alternate with update phases; an eligible process cannot be disabled by another one.

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.

The paper is structured as follows: section II presents an overview of SystemC. Section III is the formal setting; Section IV explains the algorithms and section V proves the properties of the method. We present our implementation and evaluate it in section VI, related work in section VII, and we conclude with section VIII.

II. SystemC and the Scheduling Problems

A TLM model written in SystemC is based on an *architecture*, i.e. a set of components and connections between them. Components behave *in parallel*. 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 the set of concurrent processes that appear in the components, 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, user-defined mechanisms based on abstract communication channels.

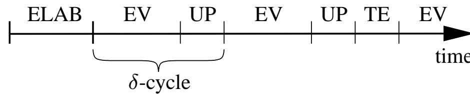

Fig. 1. Diagram of an execution

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 2. 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).

A. The SystemC Scheduler

According to the SystemC Language Reference Manual [5], the scheduler must behave as follows. 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. There are two kinds of *wait* instructions: a process may wait for some time to elapse, or for an event to occur.

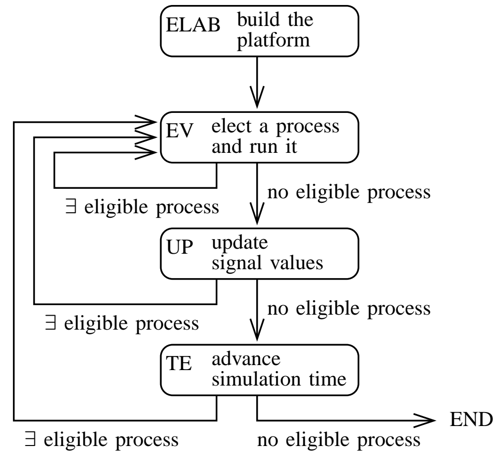

Fig. 2. Automaton of the SystemC Scheduler

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 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 be become eligible at any of the three steps EV, UP or TE.

B. Examples

```
void top::A() {                  | void top::B() {
 wait(e);                        |  e.notify();
 wait(20,SC_NS);                 |  x = 0;
 if (x) cout << "Ok\n";          |  wait(20,SC_NS);
 else cout << "Ko\n";}           |  x = 1;}
```

Fig. 3. The `foo` example

To illustrate possible consequences of scheduling choices, let us introduce two small examples of SystemC programs. Figure 3 shows the example `foo` made of two processes `A` and `B`. It has three possible executions according to the chosen scheduling, leading to very different results:

- A;B;A;[TE];B;A: This scheduling leads to the printing of the string "Ok".
- A;B;A;[TE];A;B: The string "Ko" is printed. It is a typical case of *data-race*: `x` is tested before it has been set to 1.
- B;A;[TE];B: 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 `A` has not been woken up. It is a particular form of *deadlock*.

```
void top::A()          | void top::C() {
   as in example foo   |    sc_time T(20,SC_NS);
void top::B()          |    wait(T);
   as in example foo   | }
```

Fig. 4. The `foobar` example

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 4. `foobar` has 30 possible executions, but only 3 different final states. 12 executions are equivalent to "C;A;B;A;[TE];C;B;A", 12 to "C;A;B;A;[TE];C;A;B" and 6 to "C;B;A;[TE];C;B". The method we present generates only 3 executions, one for each final state (or equivalence class).

In general testing techniques, the idea of generating one representative in each class of an equivalence relation is called *partition-based testing* [6]. It is not always formally defined.

C. Communication Actions

We call *communication actions* all actions that affect or use a shared object. We consider only two kinds of shared objects: events and variables. All other synchronization structures can be modeled using these two primitives.

There are two operations on events: *wait* and *notify*; and two operations on variables: *read* and *write*. In the sequel we will distinguish *caught* notifications (those that have woken up a process) from *missed* notifications, and *writes* that have modified the current value from non-modifying ones. Of course, theses distinctions can only be done dynamically in the general case.

III. Formal Setting

We will now explain how we generate schedulings for multi-threaded models written in SystemC. 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[1]. We have to generate a relevant set of schedulings for this data.

Most of the definitions in this section are quite standard in the literature on partial order reduction techniques.

A. Representation of the SUTD

When data is fixed, a SUT execution is entirely defined by its scheduling; a scheduling is entirely defined by an element of $(P \cup \{\delta, \chi\})^*$ where P is a process identifier and δ, χ are special symbols used to mark the δ-cycle changes and time elapses respectively. We consider 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. It is partial: not all the elements of $(P \cup \{\delta, \chi\})^*$ represent possible schedulings of the SUTD (because of the synchronization constraints between processes).

[1]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.

Definition 1 (Schedulings): Let M be a SUTD. P_M is the set of its processes; S_M is the set of its reachable full states; $F_M : (P_M \cup \{\delta, \chi\})^* \longrightarrow S_M$ is its associated *function*. F_M is partial. A *scheduling* is an element of $(P_M \cup \{\delta, \chi\})^*$; a *valid scheduling* is an element of the definition domain of F_M: $D_{F_M} \subset (P_M \cup \{\delta, \chi\})^*$.
For the programs of Section II-B, we have: $D_{F_{\mathtt{foo}}} = \{ABA\chi BA, ABA\chi AB, BA\chi B\}$ and $F_{\mathtt{foobar}}(ABC) = F_{\mathtt{foobar}}(ACB) = F_{\mathtt{foobar}}(CAB)$.

Definition 2 (Transitions): A *transition* is 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.

Definition 3 (Permutations): Let $u = vp_iwq_j$ be a valid scheduling where the transition p_i (resp. q_j) corresponds to the i-th (resp. j-th) execution of process p (resp. q). *Permuting the transitions* p_i *and* q_j means generating a new valid scheduling u' such that u' begins by v and the j-th transition of q in u' is before the i-th transition of p: there exists x, y, z such that $u' = vxq_jyp_iz$. u' is called a *permutation of* p_i *and* q_j for u.

We will use letters p, q, r to denote processes, $a, b, c, \dots$ to denote transitions and $u, v, \dots$ to denote sub-sequences of schedulings. Indexes will be omitted when obvious by context. An equivalence on the set of schedulings is needed to determine whether two schedulings lead to the same final state. We first define the relation $\sim$:

$$\forall uabv \in D_{F_M},\ uabv \sim ubav \Leftrightarrow (ubav \in D_{F_M} \wedge F_M(uabv) = F_M(ubav))$$

Definition 4 (Equivalence of Schedulings): The equivalence of schedulings is the reflexive and transitive closure of the relation $\sim$. It is noted $\equiv$.
This definition complies with the property: $\forall u, v \in D_{F_M}, u \equiv v \Rightarrow F(u) = F(v)$. Therefore, if we generate one element of each equivalence class of $\equiv$, we will have all possible final states. It allows to detect all property violation as soon as the corresponding output checker has been included into the SUT and drives it to a special final state when it detects an error.

B. Transition Dependency and Permutation Choice

We produce alternative schedulings by permuting some transitions of a given scheduling, but only when this can lead to a non-equivalent scheduling. For example, suppose that we are executing a SUTD and we have just executed the process p and then the process q ($u = u_1p_iq_j$). If there is no causal reason why the transition q_j was after the transition p_i (process q was not waiting for an event notified in p_i), then we can permute these two transitions. In that case, executing q instead of p in the state $F_M(u_1)$ can be a divergent path as illustrated on figure 5. The question we have to answer is: "Do these two schedulings lead to the same state?" or formally: "$F_M(u_1pq) = F_M(u_1qp)$?". Note that we may not be able to prove that $F_M(u_1pq) = F_M(u_1qp)$ because we want to answer this question *without* executing u_1qp entirely. Hence we rely on the common objects accessed by the transitions to

guess whether a permutation has some effect on the final state. This is incomplete. If we cannot prove that the final states are equal, we generate the new scheduling.

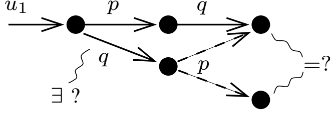

Fig. 5. A Potential Divergent Path, black circles represent global states of the model

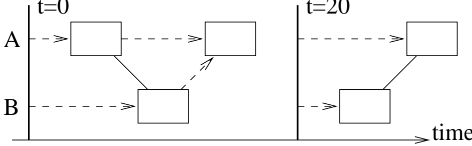

Fig. 6. Dynamic Dependency Graph

We now study the two questions: which transitions *can* we permute? which transition permutations are *useful*? The answer to the first question is given by the *permutability relationship*; the answer to the second question is given by the *commutativity* relationship (it is useless to permute commutative transitions).

The *Dynamic Dependency Graph (DDG)* represents the synchronizations that occur for a particular scheduling. Figure 6 represents the scheduling $aba\chi ba$ of the `foo` program of figure 3. Each horizontal line is a process. New cycles (δ or χ) are represented by vertical lines. Each box is a process transition. Dashed arrows (resp. plain lines) between boxes indicate that the two transitions are dependent but not permutable (resp. non commutative). 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 or line.

Definition 5 (Permutability): The transitions a and b are *causally permutable* in the valid scheduling $u_1au_2bu_3$, noted $(a,b) \in P$, if and only if: $\{u_1v_1ba \in D_{F_M} | \exists v_2, u_1v_1abv_2 \equiv u_1au_2bu_3\} = \emptyset$.
In other words, two transitions are not permutable if:

1) there is an equivalent scheduling in which they are consecutive;
2) the second transition b can be elected in place of the first transition a in this equivalent scheduling.

Definition 6 (Commutativity of Transitions): The non-causally ordered transitions a and b are *commutative* in the valid scheduling $u_1au_2bu_3$ if and only if:
$\forall u_1v_1abv_2 \equiv u_1au_2bu_3, u_1v_1abv_2 \equiv u_1v_1bav_2$
Commutativity is not defined for causally ordered transitions.

The theory of partial order reduction relies on the definition of *dependent* transitions [7]. In our work, we define the dependency relationship D as follows:

Definition 7 (Dependency of Transitions): The transitions a and b are *dependent* if and only if they are not permutable, or permutable but not commutative.

The *causal order* specifies which transitions can be permuted in a particular scheduling without permuting dependent transitions, including themselves. All schedulings of the same equivalence class have the same causal order. Unlike the permutability relationship, the causal order is a partial order.

Definition 8 (Causal Order): The transitions a and b are *causally ordered* in the valid scheduling $u = u_1au_2bu_3$, noted $a \prec_u b$, if and only if $(a,b) \in$ transitive closure of $\{(x,y) \in D | x <_u y\}$.

IV. Algorithms

A. Computation of the Commutativity Relationship

The first step is to detect pairs of transitions which are not commutative. We compute here a relationship C for all pairs of transitions. This computed relationship is correct for permutable transitions, which is sufficient for our problem. Two transitions may be non-commutative ($(a,b) \notin C$) only if they contain non-commutative *communication actions* on the same shared object (see section II-C). Note that the order of these actions within a transition is irrelevant. We examine all cases below.
For shared variables there are three cases of non-commutative actions (since operations on variables have no effect on process eligibility, we just need to check whether the equality of resulting states is still verified after permutation):

1) a *read* followed by a *modifying write*
2) a *modifying write* followed by a *read*
3) a *write* followed by a *modifying write*

In all other cases, the transitions are commutative, as in example 2. Note that the nature of a *write* depends on the scheduling we consider. A *modifying write* can become a *non-modifying write* for another scheduling, and reciprocally.

Example 1: Variable `x` initially set to 0. The first transition executes the action `x=x+2`. The second executes `x=4-x`. It is a *modifying write* followed by a *read* so we consider that the two transitions are not commutative (point 2 above).

Example 2: Variable `x` initially set to 2. The first transition executes the action `x=4`. The second transition also executes this instruction. It is a *modifying write* followed by a *non-modifying write*.

Note that C is symmetric, which may not be obvious from point 3 above. But permutating a *modifying write* with a *non-modifying write* is still a *modifying write* followed by a *non-modifying write*, except if there is another pair of dependent actions. Example 2 also illustrates this remark.
For events, there are three cases of non-commutative actions:

1) a *notification* followed by a *wait*
2) a *wait* followed by a *notification*
3) a *caught notification* followed by a *notification*

The dependency between a *wait* and a *notify* is quite obvious: if the *wait* comes first, then the corresponding process is woken up by the *notify*, otherwise it remains sleeping. Example 3 illustrates the third case.

Example 3: Suppose one runs this three-process model:

- Initial state: process `A` waiting for `e`, `B` and `C` eligible.
- Process `A`: `cout <<'a'; x = 1;`

- Process B: `cout <<'b'; x = 2; e.notify();`
- Process C: `cout <<'c'; e.notify();`

There is exactly one transition per process, noted a, b and c. Four schedulings are valid: bac, bca, cba and cab. In bac and bca, b is dependent with a (2 *modifying writes*) but they are causally ordered (process A was enabled by the transition b). However if we permute b and c, b is no longer causally ordered with a since A was enabled by c instead of b.

Permuting two notifications of an event does not modify the resulting state of the SUTD, but modifies the computed causal order. That's why they are considered as non-commutative.

B. Computation of the Causal Partial Order

In order to compute the permutability, we need to compute the causal order $\prec$. We denote prec(u) the set $\{a, b \in u | a \prec b\}$ obtained after the execution of the scheduling u.

We compute the causal order step by step. Obviously, for the empty scheduling we have prec$(\epsilon) = \emptyset$. Let a and b be two transitions, we have $a \prec b$ and so $(a, b) \in D$ at least in the three following cases:

- a or b indicate a new δ-cycle or time-elapsed.
- a and b belong to the same process (by definition)
- the process of transition b has been woken up by a.

In these cases, we note: $a \prec_\beta b$. The rest of the paragraph below is adapted from [4]. Having prec(u), we compute prec(ub) as follows:

$$\begin{aligned}\text{prec}_1(ub) &= \text{prec}(u) \cup \{a \prec_\beta b | a \in u\} \\ \text{prec}_2(ub) &= \text{prec}_1(ub) \cup \{(a, b) \notin C | a \in u\} \\ \text{prec}(ub) &= \text{transitive closure of prec}_2(ub)\end{aligned}$$

Finally, we have $(a, b) \in P$ in $u_1 a u_2 b u_3$ if and only if: $(a, b) \in$ transitive closure of prec$_1(u_1 a u_2 b)$.

The following property is useful to optimize the implementation: Let $u_1 a u_2 b u_3 c u_4$ be a scheduling. Then process$(a) =$ process$(b) \wedge b \prec c \Rightarrow a \prec c$. Owing to this property, we can represents the causal order with an array T of size $p \times s$ where s is the number of steps and p is the number of processes. The element $T[a, q]$ is the last transition of process q which is causally before a; i.e.: $a \prec b \Leftrightarrow \text{num}(a) \leq T[b, \text{process(a)}]$. Some other optimizations are well explained in [4].

C. Generation of one alternative scheduling

We are now able to determine if two transitions are not commutative (hence should be permuted). Now we explain how we treat such a pair of transitions. Let $uavb$ be a scheduling such that $(a, b) \in D \cap P$. Let $v = v_1 \ldots v_n$ where $v_1, \ldots, v_n$ are transitions. The goal is to generate a new valid scheduling with b before a. We proceed as follows:

- The first part u is unmodified.
- We execute all v_i such that $a \not\prec v_i$.
- We execute b and then a (unlike some other concurrent languages, b cannot disable a in SystemC).
- Then, since two dependent transitions have been permuted, we do not know whether the non-executed transitions v_i such that $a \prec v_i$ are still defined. We are then free to choose the rest of the scheduling.

D. Generation of a full schedulings suite

We start by executing the SUTD with a random scheduling. In parallel with the SUTD execution, we run a checker:

- the checker computes the causal partial order "$\prec$" and builds the Dynamic Dependency Graph.
- if it discovers two non-commutative transitions p_i and q_j, with p_i before q_j:
  - it generates a new scheduling such that q_j before p_i by permuting the transitions with the algorithm described above; the constraint "q_j before p_i" is saved with the new scheduling to prevent further permutations of the same transitions.
  - it continues the current execution, adding the opposite constraint "p_i before q_j" to all of its further children.

Then we replay the SUTD with each generated scheduling u. When we reach the end of u, we continue the SUTD execution with a random scheduling. In parallel, we compute the causal order and generate new schedulings for each non-commutative pair of transitions, as for the previous schedulings. Thanks to the constraints saved with the generated schedulings, each new generated scheduling is more constrained than its father scheduling and so 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.

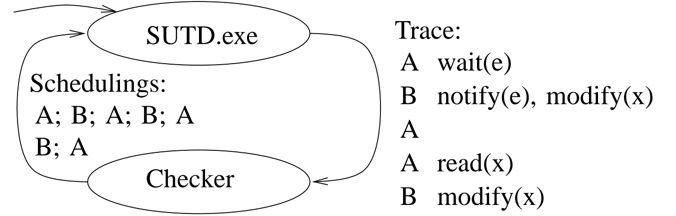

Fig. 7. First iteration of the analysis for the `foo` example. The first execution activates processes A and B in the order $ABAAB$. The checker generates two new schedulings. One to permute A_1 and B_1 (unordered accesses to event e) and the other to permute A_3 and B_2 (unordered accesses to shared variable x).

V. PROPERTIES

The algorithm guarantees that we generate at least one element of each equivalence class (for the equivalence of definition 4).

Theorem 1: Let G_M be the set of all generated schedulings of a model M. For any scheduling $u \in D_{F_M}$, there exists a scheduling $v \in G_M$ such that $u \equiv v$.

There are two useful and direct corollaries. First, if a local process state is present in a scheduling of D_{F_M}, it is also present in a scheduling of G_M. Furthermore, we generate all the final states, including all deadlocks.

To prove the property, we need the definition of $\equiv-$prefix and $\equiv-$dominant for schedulings, directly adapted from *prefix* and *dominant* properties of Mazurkiewicz traces [7].

Definition 9: Let $p, d \in D_{F_M}$ be two schedulings, p is an $\equiv-$prefix of d and d an $\equiv-$dominant of p if and only if there exists a scheduling $u \in D_{F_M}$ such that $u \equiv d$ and p is a string-prefix of u.

Proof: We proceed by contradiction, and assume that there exists a scheduling $u \in D_{F_M}$ which breaks the property. We can write u in the form $u = u_1 a u_2$ where u_1 is the longest prefix of u such that:

$$\exists u_1 u'_2 \in D_{F_M} \text{ and } v \in G \text{ such that } u_1 u'_2 \equiv v$$

This decomposition is unique so we just have to prove that $u_1 a$ has an $\equiv$−dominant in G to get the wanted contradiction.

Let $v \in G$ be a generated completed scheduling such that u_1 is a $\equiv$−prefix of v. As a consequence, there exists a valid scheduling $u_1 u'_2$ such that $u_1 u'_2 \equiv v$ If there is no non-determinism when we are in the state $F_M(u_1)$, then we must have $u'_2 = a u'_3$ and so v would be a $\equiv$−dominant of $u_1 a$.

Consequently a is neither δ nor χ and the process of a is defined and eligible in $F_M(u)$. Since an eligible process cannot become "sleeping" without running, a is present in u'_2 so $u'_2 = w_1 a w_2$. Since a is eligible in $F_M(u)$, it is not causally after any element of w_1. There are three cases:

- if w_1 is empty then we get the needed contradiction
- if $w_1 = xb$ with b I a then there exists another possible scheduling $u_1 u''_2 \equiv v$ such that $u''_2 = w'_1 a b w_2$ with w'_1 shorter than w_1.
- if $w_1 = xb$ with $(b, a) \in D$ then:
  - Transition b is before a in v but they are permutable.
  - So we have generated a scheduling v' with a before b, using the algorithm described in section IV-C.
  - There exists a possible scheduling $u_1 u''_2 \equiv v'$ such as $u''_2 = w'_1 a b w'_2$ with w'_1 shorter than w_1.

Consequently, by induction on the length of w_1, we get the needed contradiction. ■

VI. Prototype Implementation and Evaluation

A. The prototype

Figure 8 is an overview of the tool. The **checker** implements the checking algorithm of section IV-D. It has to be aware of all communication actions. Some of them can be detected by **instrumenting** the SystemC kernel, some other cannot (like accesses to a shared variable, that are invisible from the SystemC kernel). We choose to instrument the C++/SystemC source code. For each communication action in the code of a SystemC process, we add an instruction that notifies the operation to a global recorder. For example, consider the instruction `x=y` where `x` and `y` are shared variables. The two following instructions are added close to the assignment: `recorder->read(&y);recorder->write(&x)`. Instrumentation is based on the open-source SystemC front-end Pinapa [8], and is compositional.

Another solution would have been to interpret or instrument the binaries. However, using a SystemC front-end has some benefits: it allows to generate a *static dependency graph (SDG)* which represents a superset of the communications that can occur between processes (see Figure 9). Moreover, it is easier to link the observed behavior to the source code.

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;

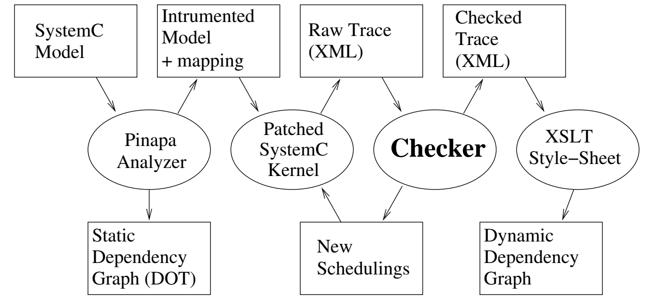

Fig. 8. The Prototype's Architecture

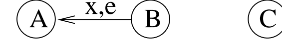

Fig. 9. Static Dependency Graph for the `foobar` example. Nodes represent processes. Arrows represent possible communications between processes. An arrow goes from the master (i.e. the notifier for a SystemC event, the writer for a shared variable) to the slave.

2) adding code to record the communication actions that cannot be detected in 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 checker afterwards. In both cases, we get a list of new schedulings to be executed, and a record of the computed dependencies, usable as input for other checkers or visualization tools, like the production of the dynamic dependency graph (DDG).

B. Evaluation

In order to validate our tool and to evaluate the quality of the test suites produced, we studied several industrial SoC models. Assume that running one test-case takes some time T. In order to cover the scheduling choices, we have to run more than one test-case. Let us denote V the number of *valid* schedulings, and G the number of schedulings *generated* by our tool. It is interesting to compare $V \times T$ with $G \times T + O$, where O is the overhead due to the computation of new schedulings.

With a real application, it is often difficult to evaluate V. We chose to evaluate our method on three examples. First, we considered a SystemC encoding of the indexer problem presented in [4][2], because it is easy to evaluate V. However, the indexer is not representative of the typical SystemC code found in industry. We then looked at two industrial case-studies: the first one has about 50 000 lines of code but only 4 processes, and it does not model a full SoC; the second one has about 250 000 lines of code and 57 processes, and it represents a full SoC.

1) The Indexer Example: There are n components and one global 128-element array used as a hash table. Each component is composed of 2 threads which communicate using a shared variable and a SystemC event. Each component writes 4 messages in the global hash table. This corresponds to schedulings of length $11 \times n$. For $n \leq 11$, there is no collision

[2]For the SystemC version see: `http://www-verimag.imag.fr/~helmstet/indexer.cpp`.

in the hash table and all schedulings lead to the same final state. For $n \geq 12$ there are collisions hence non-equivalent schedulings. Our prototype generates valid schedulings leading to distinct states of the hash table. 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 I. Time is given only to help estimating the curve, not as an absolute measure.

components	generated schedulings	time
$1 \ldots 11$	1	≤ 11 ms
12	8	60 ms
13	64	4 s
14	512	35 s
15	4096	5 mn

TABLE I
RESULTS FOR THE INDEXER EXAMPLE

2) The MPEG Decoder System: 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 of sequential code, and very few synchronizations. We added 340 instrumentation lines to detect communication actions.

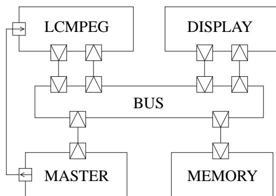

Fig. 10. Architecture of the MPEG decoder system

The test is stopped after the third decoded image, which corresponds to 150 transitions. One simulation takes 0.39 s. Our tool 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. Running the model 128 times takes more time than generating the schedulings (we have $G \times T = 128 \times 0.39$ s ≈ 50 s and $O \approx 1$ mn $08 - 50$ s $\approx$ 18 s). Thus the overhead O remains acceptable.

On this example, we noticed that the number of generated schedulings could be improved. This MPEG decoder, as many other TLM 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 `P; Q` and `Q; P; Q` 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 relationship as computed in section IV. Detecting this kind of structures in the source code and taking them into account for the computation of the dependency relationship would allow to generate less schedulings.

3) A Complete SoC: Complete models of Socs are typically 3 to 6 times bigger than the MPEG decoder. We are currently evaluating our tool on a model —let us call it XX— corresponding to a full SoC: it has about 250 000 lines of code and 57 processes. At the moment we are limited by the code instrumentation tool which still requires some manual work, so we looked at only one case study of this type, but the instrumentation tool will soon be fully automatic. For tests of length around 200 transitions, we expect the tool to behave well on XX: the ability to cope with this number of processes has been tested with the indexer example, and the ability to cope with the complexity of a large and realistic SystemC description has been tested with the MPEG example.

The interesting point with XX is the *granularity* of the transactions. With the MPEG decoder, the granularity corresponds to an algorithm that takes one line of the image at a time. Something interesting can be observed by a test oracle after 150 transitions only (three images have already been decoded). XX corresponds to an algorithm that takes one pixel of the image at a time. It may be the case that the test oracle has to observe thousands of transitions. XX is a very good case-study for observing the combined influence of the test length and the granularity on the performances of our technique. One phenomenon we can expect, and that we have to validate with the case-study, is the following: very abstract TLM descriptions have large-grain transactions, but loose synchronisations; while the more detailed TLM descriptions have finer-grain transactions, but stronger synchronizations. If the number of alternative schedulings decreases (because of stronger synchronizations) when the granularity of a description increases (and thus the length of the interesting test-cases), the method may still be applicable. We also comment on this point in the conclusion.

VII. RELATED WORK

Existing work (see, for instance [9]) addresses formal verification for TLM models. The idea is to extract a formal model from the SystemC code, and to translate it into the input format of some model-checker. In such an approach, the complete model that is model-checked has to include a representation of the scheduler. It is sufficient to use a non-deterministic representation that reflects the specification of SystemC, and then a property that is proved with this non-deterministic scheduler is indeed true for any deterministic implementation. Model-checking is likely to face the state-explosion problem, so testing methods are still useful. But we need the same guarantee on the results of the test being valid for any implementation of the simulation engine.

Partial order reduction techniques are quite old, but their *dynamic* extension is quite recent. As far as we know, it is not included in VERISOFT [10] yet. Partial order reduction is used in many model checkers for asynchronous concurrent programs such as Spin [11] or JAVA PATHFINDER [12]. However, since we use testing, our work is more related with tools which work directly on the program without abstractions,

such as VERISOFT or CMC [13]. The main difference is that our tool is adapted to the TLM SystemC constructs.

To get a complete validation environment, one need to include a test case generator and an output checker. For the latter, *assertion-based verification* [14] proposes to derive monitors from assertion languages. However, these languages are often based on the notion of clocks which are absent in TLM. If ABV is extended to TLM, it will become useful in our framework.

VIII. CONCLUSION AND FURTHER WORK

We presented a method to explore the set of valid schedulings of a SystemC program, for a given data input. This is necessary because the scheduling is a phenomenon due to the simulation engine only, and is unlikely to represent anything concrete on the final SoC. Exploring alternative schedulings during testing is a way of guaranteeing that the SoC description, and in particular the embedded software, is scheduler-independent, hence more robust. By using dynamic partial order reduction, we maximize the coverage and keep the number of tests as low as possible. Our tool also produces several graphical views that help in debugging SoCs. With the prototype tool, we have highlighted unwanted non-determinism in a bus arbiter for a transaction-accurate protocol. Also, some SoC descriptions are scheduler-dependent because they exploit the initial state of the most used implementation. In this case, covering the valid schedulings reveals deadlocks. Our tool is already mature enough to be used for industrial SystemC descriptions of SoCs.

There are at least two ways of improving the prototype performances. The first is to reduce the number of branches explored. A promising solution is to use partial state memorization. It is unrealistic to save all the states and compare the new state at each step due to the size and complexity of a SystemC model state. However, we can save some states and compare only particular new states. We plan to compare each forked execution every new delta-cycle. The second way is to reduce the time overhead needed for runtime checking. Some check results are predictable. Consequently doing static analysis before simulation can avoid runtime computation.

Further work on testing SoCs is threefold. First, the algorithm that fully explores alternative schedulings can be used on large platforms only if the length of the test is reasonable. A promising idea for very long tests is to use the method *locally* on the TLM description: a first execution of the whole platform P is used to record the output transactions of some sub-system S of P. Then, our method is applied on a platform P' obtained by substituting S' with S in P. S' is a sequential algorithm that plays the recorded transactions. It does not introduce scheduling choices. The idea is that the method then concentrates on the schedulings due to P−S, forgetting the schedulings due to S.

Second, the whole approach and the SystemC prototype is being adapted to the exploration of non-fully specified timings in the TLM models. Indeed, TLM models are not cycle-accurate, but people use to label them by approximate timing properties of the components, in order to estimate the timing properties of the SoC early. In this case, the timings should not be taken as fixed values. The embedded software will be more robust if it works correctly for slightly distinct timings. In the testing process, it is useful to explore alternative timings, with the same idea of generating only those timings that are likely to change the global behavior of the SoC. An overview of the method can be found in [15].

We also started working on efficient implementations of the SystemC simulation engine, by exploiting multi-processor machines. Here, the difficulty is to guarantee that a multi-processor simulation does not exhibit behaviors that are not allowed by the non-deterministic reference definition of the scheduler. The formal setting we described here is appropriate for defining the set of behaviors that the multi-processor simulation may produce, without changing the behavior of the embedded software.

REFERENCES

[1] F. Ghenassia, Ed., *Transaction-Level Modeling with SystemC. TLM Concepts and Applications for Embedded Systems*. Springer, June 2005, iSBN 0-387-26232-6.

[2] J. Rose and S. Swan, "SCV Randomization," Cadence Design Systems, Inc., 2003, www.testbuilder.net/reports/scv_randomization.pdf.

[3] T. Kuhn, T. Oppold, M. Winterholer, W. Rosenstiel, M. Edwards, and Y. Kashai, "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, pp. 413–418.

[4] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software," in *Symposium on Principles of programming languages (POPL)*. New York, NY, USA: ACM Press, 2005, pp. 110–121.

[5] *SystemC v2.0.1 Language Reference Manual*, Open SystemC Initiative, 2003, http://www.systemc.org/.

[6] J. B. Goodenough and S. L. Gerhart, "Toward a theory of test data selection," in *Proceedings of the international conference on Reliable software*, 1975, pp. 493–510.

[7] A. Mazurkiewicz, "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, pp. 279–324.

[8] M. Moy, F. Maraninchi, and L. Maillet-Contoz, "Pinapa," 2005, http://greensocs.sourceforge.net/pinapa/.

[9] M. Moy, F. Maraninchi, and L. Maillet-Contoz, "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*, June 2005.

[10] P. Godefroid, "Model checking for programming languages using VeriSoft," in *Symposium on Principles of Programming Languages (POPL)*, ACM, Ed. New York, NY, USA: ACM Press, 1997, pp. 174–186.

[11] G. J. Holzmann, "The model checker SPIN," *Software Engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[12] W. Visser, K. Havelund, G. Brat, and S.-J. Park, "Model checking programs," in *Proc. of the 15th IEEE International Conference on Automated Software Engineering*, 2000.

[13] M. Musuvathi, D. Park, A. Chou, D. R. Engler, and D. L. Dill, "CMC: A Pragmatic Approach to Model Checking Real Code," in *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, Dec. 2002.

[14] "Assertion-based verification," Synopsis, 2003, http://www.synopsys.com/products/simulation/assertion_based_wp.html.

[15] C. Helmstetter, F. Maraninchi, and L. Maillet-Contoz, "Test coverage for loose timing annotations," in *11th International Workshop on Formal Methods for Industrial Critical Systems*, August 2006.