Formal Methods in System Design, 21, 251–280, 2002

Combining Software and Hardware Verification Techniques

ROBERT P. KURSHAN k@research.bell-labs.com
VLADIMIR LEVIN levin@research.bell-labs.com
Lucent Technologies, Bell Laboratories, Murray Hill, NJ 07974, USA

MARIUS MINEA marius+@cs.cmu.edu
Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA

DORON PELED doron@research.bell-labs.com
HÜSNÜ YENIGÜN husnu@research.bell-labs.com
Lucent Technologies, Bell Laboratories, Murray Hill, NJ 07974, USA

Received June 6, 2000; Accepted December 3, 2001

Abstract. Combining verification methods developed separately for software and hardware is motivated by the industry's need for a technology that would make formal verification of realistic software/hardware co-designs practical. We focus on techniques that have proved successful in each of the two domains: BDD-based symbolic model checking for hardware verification and partial order reduction for the verification of concurrent software programs. In this paper, we first suggest a modification of partial order reduction, allowing its combination with any BDD-based verification tool, and then describe a co-verification methodology developed using these techniques jointly. Our experimental results demonstrate the efficiency of this combined verification technique, and suggest that for moderate–size systems the method is ready for industrial application.

Keywords: formal verification, model checking, hardware/software co-design, partial order reduction

1. Introduction

Software and hardware verification, although having a lot in common, have developed along different paths. Even in the specific context of model checking, in which the system is represented as a graph or an automaton, several differences become apparent. Software systems typically use an asynchronous model of execution, in which concurrent actions of component modules are interleaved. In verification, the asynchrony is exploited using *partial order reduction* [8, 21, 26], which explores during verification only a subset of the available actions from each state. The remaining actions are delayed to a subsequent step, as long as this does not result in any change visible to the specification being checked.

On the other hand, hardware is typically designed for synchronous execution. All component modules perform an action at each execution step. Hardware verification usually exploits the regularity of digital circuits, often built from many identical units, by representing the state space using *binary decision diagrams* (BDDs) [20]. Another technique which

makes hardware verification manageable is *localization reduction* [14] which abstracts away the hardware design parts which are irrelevant to the verified property.

Thus, traditionally, formal verification of hardware and software is done through different techniques, using tools which are based on different algorithms, representations and principles.

However, there are important and growing classes of mixed (combined) hardware-software systems, co-designs, in which hardware and software are tightly coupled. The tight coupling precludes testing the hardware and software separately. On the other hand, there may be 100 hardware steps for one software step. The difference renders conventional simulation test exceedingly inefficient, and results in co-design systems that cannot be effectively tested by conventional means. New testing methods and commercial tools to support them have emerged to address this problem. For example, we refer the reader to websites of major EDA vendors,[1] where co-verification (as a matter of fact, co-simulation) tools are more and more heavely promoted: there are too many of them to mention here. They generally involve ad hoc abstraction of the hardware (such as removing clock dependencies), in order to speed up the simulation of the hardware relative to the software. These methods may result both in missed design errors, and false error indications that reflect errors in the abstraction, not the design. Nonetheless, the design community is forced into these new methods as the only available alternative.

Yet, there is another alternative, based on model checking. Formal verification in this context has all the well-known advantages over simulation test: better coverage, and it may be applied sooner in the design cycle. It also is able to deal in a sound fashion with the interface between hardware and software, in particular, with different speed rates on the two sides. This motivates our efforts to introduce formal verification into the area of co-design systems. For this, an efficient verification technique is needed that is able to address co-design systems containing both kinds of components, hardware and software.

In this paper, we attempt to combine the benefits of both methodologies: we suggest a verification technique that combines partial order reduction with a BDD representation and, in general, hardware verification techniques. The partial order reduction principle of selecting a subset of the enabled actions from each state poses no problem when combining it with BDDs or localization reduction. It only means that the transition relation needs to be restricted so that it takes advantage of the potential commutativity between concurrent actions. The idea that this can be done statically, at compile time, was suggested by [12], but their implementation required some changes in the depth-first search algorithm (in order to control the backtracking mechanism) of the Spin model checker [11].

The subtle point that has so far made explicit state enumeration seem more appropriate than BDD-based symbolic exploration for implementing the partial order reduction algorithm is the *cycle closing* problem. Since partial order reduction may defer an action in favor of another one that can be executed concurrently, one needs to ensure that no action is ignored indefinitely along a cycle in the state space. One solution to this problem was proposed by [3] and elaborated by [1] in the first attempt at combining partial order reduction with BDDs. Their solution is based on a conservative approximation of when a cycle may be closed during a breadth-first search. Essentially, when an edge connects a node to

another node that is at the same or a lower level in the breadth-first search, it is assumed (conservatively) to close a cycle.

We propose an alternative solution [15] that computes at compile time the conditions which guarantee that no action is ignored. The method is based on the observation that any cycle in the global state space projects to a local cycle in each participating process. These local cycles can be detected at compile time. An action from each cycle is selected, such that at run time, the execution of each selected action forces a complete exploration of all actions that have been deferred so far in favor of other actions. The number of these special actions (the more of which there are, the less the achieved reduction) can be minimized by analyzing the effects of transitions.

Our implementation of the algorithm has the unique feature that *all* the information needed for performing the partial order reduction is obtained during a compilation of the software system model. There is no change at all in the verification tool, in this case the model checker COSPAN [9, 10]. Thus, with the new algorithm, partial order reduction is implemented as a compilation or preprocessing phase for model checking, rather than as a modified model checking algorithm. It is precisely this feature that allows a combination of the partial order reduction with BDD-based algorithms and, in general, with any optimization technique applied by the model checker.

Can we gain by combining software- and hardware-oriented verification techniques, partial order reduction and BDDs? We answer this question affirmatively: the combination of partial order reduction and hardware-oriented verification techniques makes possible hardware/software co-verification, i.e., the integral verification of a hardware/software co-design. In particular, there are examples in this area for which the use of a single method (be it BDDs or partial order reduction) terminates in lack of memory due to state space explosion, whereas the combination of the two methods makes verification possible (see Section 5).

The remainder of the paper is organized as follows. The next section explains the modification of partial order reduction aimed at its combination with any existing model checker, in particular, with one based on BDDs. Section 3 presents a co-verification methodology that makes use of the combination of partial order reduction with hardware-oriented verification techniques. Section 4 describes our current implementation of this co-verification technique with an emphasis on modifying the transition relation according to the partial order reduction constraints. Section 5 presents experimental results and Section 6 the conclusion.

2. Partial order reduction

In Sections 2.1 and 2.2, we present the basics of the temporal logic LTL [7] and of the partial order reduction technique. Sections 2.3 and 2.4 describe the modification to partial order reduction required to fit our needs.

2.1. Preliminaries

The system to be analyzed is viewed as a *state graph*. If S is the set of states, a transition is a relation $\alpha \subseteq S \times S$. A state graph is defined as a tuple $M = (S, S_0, T, L)$, where $S_0 \subseteq S$

is the set of initial states and T is the set of transitions. The labeling function $L : S \to 2^{AP}$ associates each state of M with a set of atomic propositions that hold at s.

A transition $\alpha \in T$ is *enabled* at state $s \in S$ if there exists a state $s' \in S$ such that $(s, s') \in \alpha$. Otherwise α is said to be *disabled* at s. For a state s, *enabled*(s) is the set of all transitions α such that α is enabled at s. A transition α is called *deterministic* if for any state $s \in S$ in which α is enabled there is a unique s' such that $(s, s') \in \alpha$. In this case α can be viewed as a partial function on S, and the notation $s' = \alpha(s)$ can be used instead of $(s, s') \in \alpha$. In this paper, we restrict ourselves to state graphs with only deterministic transitions. Yet, nondeterminism may appear as a nondeterministic selection among several enabled transitions.

In order to simplify the picture, we avoid states from which no transition is possible, and therefore, for such (i.e. deadlocked) states, force T to have the self-looping transition $\delta = \{(s, s) \mid s \text{ has no successors except } s\}$.

An *execution sequence* σ of a state graph M is an infinite alternating sequence of states s_i and transitions $\alpha_i : \sigma = s_0 \stackrel{\alpha_0}{\to} s_1 \stackrel{\alpha_1}{\to} \cdots$ such that $s_{i+1} = \alpha_i(s_i)$ for all i. If $s_0 \in S_0$ then σ is referred to as a *full execution sequence*. We denote by σ_i the suffix of σ that starts from its ith element-state, i.e., $\sigma_i = s_i \stackrel{\alpha_i}{\to} s_{i+1} \stackrel{\alpha_{i+1}}{\to} s_{i+2} \stackrel{\alpha_{i+2}}{\to} \cdots$.

For assertions about the behavior of a program, we use the temporal logic LTL. Given a set AP of atomic propositions, LTL formulas are defined as follows:

- for all $p \in AP$, p is a formula
- if ϕ and φ are formulas, then so are $\neg\phi$, $\phi \wedge \varphi$, $\bigcirc\phi$, and $\phi \,\mathcal{U}\, \varphi$.

An execution sequence $\sigma = s_0 \stackrel{\alpha_0}{\to} s_1 \stackrel{\alpha_1}{\to} \cdots$ is said to satisfy an LTL formula ϕ (denoted by $\sigma \models \phi$) under the following conditions:

- if $\phi = p$ for some $p \in AP$, and $p \in L(s_0)$
- if $\phi = \neg\varphi$, and not $\sigma \models \varphi$
- if $\phi = \varphi \wedge \psi$, and $(\sigma \models \varphi) \wedge (\sigma \models \psi)$.
- if $\phi = \bigcirc\varphi$, and $\sigma_1 \models \varphi$
- if $\phi = \varphi \,\mathcal{U}\, \psi$, and there exists an $i \geq 0$ such that for all $0 \leq j < i$, $\sigma_j \models \varphi$ and $\sigma_i \models \psi$.

We also use the following abbreviations: $\mathit{false} = \phi \wedge \neg\phi$, $\mathit{true} = \neg\mathit{false}$, $\phi \vee \varphi = \neg((\neg\phi) \wedge (\neg\varphi))$, $\Diamond\phi = \mathit{true}\,\mathcal{U}\,\phi$, and $\Box\phi = \neg\Diamond\neg\phi$.

A state graph M satisfies an LTL formula ϕ (denoted by $M \models \phi$), iff for each full execution sequence σ in M, $\sigma \models \phi$.

In this paper, the verification problem is considered to be "given a state graph M and a specification expressed by an LTL formula ϕ, check if $M \models \phi$". Thus, from now on we consider the specification formula ϕ to be fixed. In order to simplify the following definitions we assume that the entire set of atomic propositions AP is in fact formed only by the atomic propositions used in ϕ. In other words, ϕ makes use of all the atomic propositions in AP. This condition immediately affects the following definitions of stuttering equivalence and visibility, as well as partial order reduction based on those notions (see Section 2.2), making them relative to the specification ϕ. It also reflects the implementation (see Sections 2.4 and 4).

One of the main notions underlying the partial order reduction technique is the stuttering equivalence relation between execution sequences. In an execution sequence, some

consecutive states may have the same labeling, and specifications that refer to the atomic propositions without counting the succession of states at which they are true cannot distinguish between such states. The execution sequence can then be divided into segments, each consisting of consecutive identically labeled states. Two execution sequences $\sigma = s_0 \stackrel{\alpha_0}{\rightarrow} s_1 \stackrel{\alpha_1}{\rightarrow} \cdots$ and $\rho = r_0 \stackrel{\beta_0}{\rightarrow} r_1 \stackrel{\beta_1}{\rightarrow} \cdots$, are called *stuttering equivalent* (denoted by $\sigma \sim_{\text{st}} \rho$) if there exist two infinite sequences of indices $0 = i_0 < i_1 < \cdots$ and $0 = j_0 < j_1 < \cdots$ such that $\forall k \geq 0$, $L(s_{i_k}) = L(s_{i_k+1}) = \cdots = L(s_{i_{k+1}-1}) = L(r_{j_k}) = L(r_{j_k+1}) = \cdots = L(r_{j_{k+1}-1})$. Intuitively, two execution sequences are stuttering equivalent if they have identical state labelings after in each of them, any finite sequence of identically labeled states is collapsed into a single state. Two state graphs M and M' are said to be stuttering equivalent, denoted by $M \sim_{\text{st}} M'$, if for each full execution sequence σ of M, there exists a full execution sequence ρ of M' such that $\sigma \sim_{\text{st}} \rho$, and vice versa (for each full execution sequence ρ of M', there exists a full execution sequence σ of M such that $\rho \sim_{\text{st}} \sigma$).

The importance of stuttering equivalence is the following. We call an LTL formula ϕ *stuttering invariant* if for any two execution sequences σ and ρ such that $\sigma \sim_{\text{st}} \rho$, it holds that $\sigma \models \phi$ iff $\rho \models \phi$. This definition together with the definition of stuttering equivalence between state graphs easily implies that if ϕ is stuttering invariant and $M \sim_{\text{st}} M'$, then $M \models \phi$ iff $M' \models \phi$. This is the basic idea behind partial order reduction: generate a model M' with a smaller number of states than M and use M' to model check a stuttering invariant property ϕ. Lamport [17] showed that any LTL property that does not use the next-time operator is stuttering invariant. Conversely, in [22], it is shown that any stuttering invariant LTL formula can be written without the use of the next-time operator $\bigcirc$. From now on, we restrict ourselves only to stuttering invariant LTL formulas.

We conclude this section by introducing two basic concepts used in partial order reduction. A transition α is said to be *visible* (*with respect to* ϕ) if there exist two states s and s' such that $s' = \alpha(s)$ and $L(s) \neq L(s')$.

The other key concept is the independence relation between the transitions. Two transitions $\alpha, \beta \in T$ are said to be *independent* if for all states $s \in S$, if $\alpha, \beta \in \mathit{enabled}(s)$, then: (i) $\alpha \in \mathit{enabled}(\beta(s))$; and (ii) $\beta \in \mathit{enabled}(\alpha(s))$; and (iii) $\alpha(\beta(s)) = \beta(\alpha(s))$. Intuitively, if both transitions are enabled at a state, then the execution of one of them must not disable the other (i and ii), and executing these transitions in either order must lead to the same state (iii). If two transitions are not independent, then they are called *dependent* transitions.

2.2. *Basic partial order reduction*

As explained in Section 2.1, the purpose of partial order reduction is to generate a reduced state graph M' with a smaller number of states than the original state graph M and with the property that $M' \sim_{\text{st}} M$, and then perform the model checking of a stuttering invariant LTL formula ϕ on M' rather than on M.

No matter what search technique is used (depth-first, breadth-first, explicit or symbolic), with a traditional model checker one has to generate the successors of a state s for the enabled transitions $\alpha \in \mathit{enabled}(s)$. However, a partial order search technique attempts to explore the successors of a state only for *a subset* of the enabled transitions of s. Let's call such a set of transitions $\mathit{ample}(s) \subseteq \mathit{enabled}(s)$. Following [21], we define below this

subset of transitions by the conditions **C0** through **C3** that it must satisfy. Exploring only the ample transitions results in the reduced state graph M'.

C0 (Emptiness). $ample(s) = \emptyset$ iff $enabled(s) = \emptyset$.

Since we are trying to generate for each execution of M a corresponding, stuttering equivalent execution sequence in M', we must explore at least one successor of s in M', if there are any successors of s in M. In our case, we have assumed for convenience that state s has at least one transition enabled in M, cf. Section 2.1. Therefore, **C0** implies that $ample(s) \neq \emptyset$.

C1 (Faithful decomposition). Along every execution sequence of transitions in M that starts at s, a transition that is dependent on any transition in $ample(s)$ cannot be executed without a transition from $ample(s)$ occurring first.

This constraint is introduced to ensure that any execution sequence of the full state graph M may be represented by a stuttering equivalent execution sequence in the reduced graph M'. For this purpose, the transitions of the original execution sequence may have to be re-ordered. Condition **C1** ensures that all transitions before the first ample transition α in the original execution sequence are independent of α and, hence, can be commuted with α.

In order to implement condition **C1**, we can use further information about the semantics of the modeled system. For example, given a collection of concurrent processes, with the program counter of one process allowing the execution of only one local transition, choosing this transition as a singleton ample set will not violate condition **C1**. On the other hand, consider the case where the program counter of the process is at a point where there is a selection between two input messages, of which one is enabled at the current state and the other is not (until another process progresses to send such a message). In this case, selecting the enabled input transition α as a singleton ample set may violate **C1**, since some transitions independent of α may execute in the other process enabling the alternative input transition, which is interdependent with α. Implementing condition **C1** typically involves identifying such cases, see, for example, [12]. Each such case needs to be checked against the definition of condition **C1**.

C2 (Visibility). If there exists a visible transition $\alpha \in ample(s)$, then $ample(s) = enabled(s)$.

In other words, if $ample(s) \subset enabled(s)$ then no transition in $ample(s)$ is visible. In practice, one tries to avoid including visible transitions in the ample set at state s, since otherwise the entire set of enabled transitions $enabled(s)$ has to be explored. Indeed, let two independent transitions, α and β, be enabled at state s_i in graph M and let M allow these two executions:

$$\sigma = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \cdots s_i \xrightarrow{\alpha} s_{i+1}^1 \xrightarrow{\beta} s_{i+2} \cdots$$

$$\rho = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \cdots s_i \xrightarrow{\beta} s_{i+1}^2 \xrightarrow{\alpha} s_{i+2} \cdots$$

Condition **C1** by itself suggests that we do not necessarily need both executions σ and ρ in the reduced graph M'. That is: if only **C1** is applied then σ may not be generated in M' assuming that ρ will represent it in M' (or vice versa). Now, consider the case that the propositional labeling is different at states s^1_{i+1} and s^2_{i+1} where the two executions σ and ρ differ from each other, i.e. $L(s^1_{i+1}) \neq L(s^2_{i+1})$. If both transitions α and β are visible then it cannot be guaranteed that executions σ and ρ are stuttering equivalent: they are not if, for instance, $L(s_i) \neq L(s^1_{i+1})$ and $L(s_i) \neq L(s^2_{i+1})$. Therefore, $ample(s) = enabled(s) = \{\alpha, \beta\}$ must hold in this case to force exploration of both transitions α and β. However, if one of the two transitions, let, α, is invisible, then $\sigma \sim_{\text{st}} \rho$ is guaranteed. This is because in this case $L(s_i) = L(s^1_{i+1})$ and $L(s^2_{i+1}) = L(s_{i+2})$. Then, we don't have to generate both execution orders in M' and may select $ample(s) = \{\alpha\}$ to factor out the execution sequence ρ.

C3 (Cycle closing). Given an execution sequence $\sigma = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \cdots$ of M, if $s_k = s_0$ for some $k > 0$, then there exists $0 \leq i < k$ such that $ample(s_i) = enabled(s_i)$.

In other words, any cycle in the full state graph of M must have at least one state s on the cycle such that $ample(s) = enabled(s)$. The intuitive reason for this condition is to avoid postponing an enabled transition indefinitely while generating the reduced graph M'. A good reduction algorithm will aim at selecting ample sets that satisfy **C1** and **C2** yet contain few transitions, thus postponing the execution of visible and dependent transitions as long as possible. Conditions **C1** and **C2** guarantee that any postponed transition remains enabled. However, if the search on the reduced graph M' closes a cycle and terminates, the postponed transitions will not be executed at all. This may cause a relevant execution sequence of M to be lost, not being represented by a stuttering equivalent execution of M'. Condition **C3** prevents this.

These four conditions complete the definition of the partial order reduction. It has been shown [21] that if a search generates M' instead of M while satisfying **C0** through **C3**, then $M' \sim_{\text{st}} M$.

2.3. Static partial order reduction

One of the goals for our modification to the basic form of partial order reduction is to separate this technique from the model checking algorithm. We achieve this by a supplementary compilation phase which statically analyzes and preprocesses the model, as explained below.

In general, the system to be analyzed is not given directly as a state graph M (cf. Section 2.1), but rather as a set of component processes $\{P_1, P_2, \ldots, P_n\}$ and a set of variables $\{V_1, V_2, \ldots, V_m\}$. The states of M are then tuples (vectors) of the form $(c_1, \ldots, c_n, v_1, \ldots, v_m)$, where c_i is the control point (state) of process P_i and v_j is the value of the variable V_j. Thus, the state vector is composed of a *control part* and a *data part*.

The component processes can perform local actions and communicate with each other via certain mechanisms such as shared variables or message passing (blocking or non-blocking). The transitions of the underlying state graph M are then the local transitions of a single process, accompanied perhaps by assignments to variables, and the shared transitions of two or more processes (e.g., a communication transition).

Given a state graph M, we define the *control flow graph* of a component process P_i by projecting the state vectors of M onto the ith component. More exactly, if $s[i]$ is the ith field of the state vector $s \in S$, where S is the set of states of M, the set of *control points* of process P_i is defined as $C_{P_i} = \{s[i] \mid s \in S\}$.

A transition $\alpha \in T$ of the state graph M is actually performed by one or more component processes (in the case of local and shared transitions, respectively). With each transition α we associate therefore a set of processes $act(\alpha) \subseteq \{P_1, P_2, \ldots, P_n\}$ that are *active* for the transition α. A process is active if it updates its control point and, possibly, some variables, while executing transition α. For example, in a system with rendezvous synchronization, $act(\alpha)$ would include all the processes that participate in the synchronization on α. However, in a message passing system, the active set of a send transition only includes the sending process. The receiving process does not actually participate in this transition, since the send transition only updates an input buffer, which is a variable in the system.

The control flow graph G_{P_i} of a process P_i is defined as a labeled directed graph with the set of vertices C_{P_i}. For two control points $c_1, c_2 \in C_{P_i}$, there is an edge from c_1 to c_2 in G_{P_i}, iff there exist two states $s_1, s_2 \in S$ and a transition $\alpha \in T$ such that $s_1[i] = c_1$, $s_2[i] = c_2$, $P_i \in act(\alpha)$, and $s_2 = \alpha(s_1)$. Thus, transition α is projected onto edges in the control flow graphs of the component processes in $act(\alpha)$. Edge (c_1, c_2) in G_{P_i} is labeled by (the set of) all the transitions $\{\alpha_1, \ldots, \alpha_k\} \subseteq T$ which are projected onto this edge. If any of those transitions is executed, we say that edge (c_1, c_2) is executed. Note that the set of transitions labeling an edge is never empty. One may also observe that a system M may often be formalized in such a way that each edge in a control flow graph will be labeled with exactly one global transition.

In the following, the terms *local transition* and *local state* are used to refer to an edge and a vertex in a control flow graph G_{P_i}, whereas the terms *global transition* and *global state* are used as synonyms for a transition and a state of the underlying state graph M, respectively. The local transition γ of G_{P_i} that corresponds to a global transition α is referred to as the *local image* of α in G_{P_i}.

The generic approach to partial order reduction described in Section 2.2 operates by defining a reduced state transition graph M' which is equivalent to M. Past approaches have incorporated this technique directly into model checkers, by modifying their algorithms to explore at each state s the transitions in $ample(s)$ rather than all the transitions in $enabled(s)$. Our key idea is different. Suppose that the component processes of the model can be modified to make a transition α enabled at s precisely if $\alpha \in ample(s)$ in the original model. Then the original model checking algorithm can be applied without modifications, resulting in the verification of the reduced model M'. Besides simplicity, this method allows partial order reduction to be combined with all features of the existing model checker. Essentially, verification of a system is split into two phases: a syntactical transformation of the system to a reduced system that generates the partial order reduction in the course of the state space search of model checking, followed by the actual application of model checking.

To illustrate the basic idea by an example, suppose that we are given the two-process system depicted in the upper left box of figure 1. Each process has only one transition, α and β respectively, both with enabling condition *true*. Thus, α is always enabled when process P is at state $s1$, and β is always enabled when process Q is at state $r1$. Suppose that P and

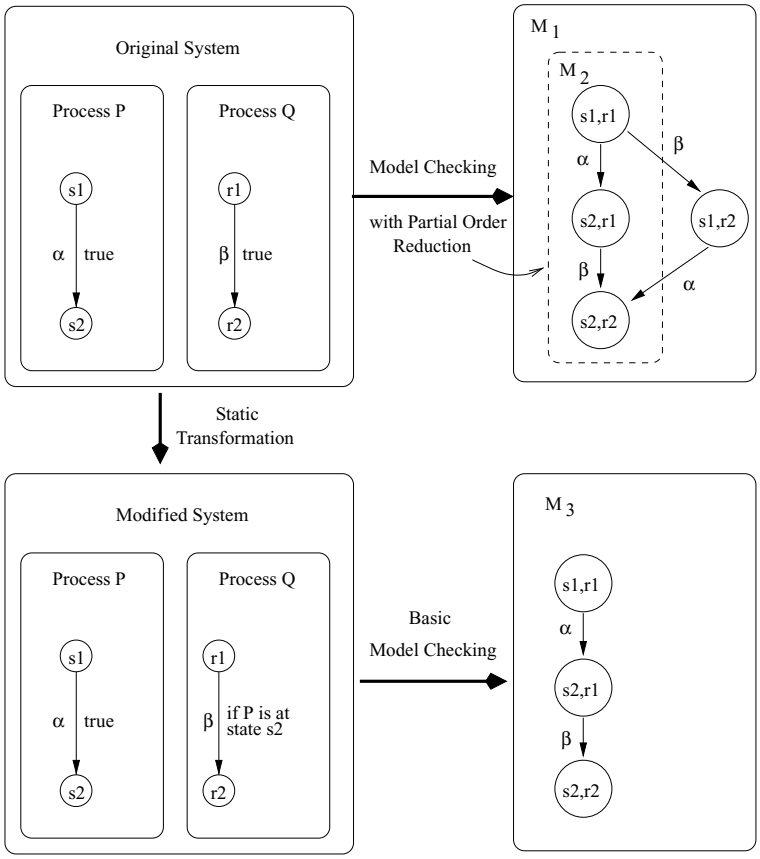

Figure 1. An example static transformation.

Q run concurrently, and that concurrency is modeled by interleaving. Then the upper right box in figure 1 shows the state transitions of the underlying state graph M_1 of the original system, as generated by a model checker without partial order reduction. A model checker with partial order reduction capabilities might produce the state transition graph M_2, shown in the dashed box. Assuming that α and β are invisible, we have $M_1 \sim_{st} M_2$.

Our method transforms the original system statically into the system shown in the lower left box of figure 1, by a compilation prior to the model checking step. The modified system is described in the same language as the original, and can therefore be analyzed by the same tool. However, its structure already incorporates the partial order reduction. The modified enabling condition of transition β guarantees that the state graph M_3 generated by the model checker from the modified system is the same as M_2, and therefore $M_3 \sim_{st} M_1$.

The above example shows that our transformation changes the enabling conditions of the transitions in the component processes. Generally, in order to guarantee the conditions **C0–C3** introduced in Section 2.2, the compilation step needs to determine for any global

state which transitions can form an ample set, and change the enabling conditions of the transitions accordingly. This may include introducing conditions on the current state of other processes. Computing these changes is quite easy for conditions **C0–C2**, and Section 4 explains how to check **C0** and **C1**. However, condition **C3** refers to the state transition graph M', which is only unfolded in the verification stage. Thus, it seems difficult to determine statically, at compilation time, the transition cycles that will appear in M'. Condition **C3** can be checked naturally during a depth-first search, as implemented in most previous partial order model checkers, which use explicit state enumeration. However, that is the opposite of our goal, since we aim for BDD-based symbolic verification, which operates in a breadth-first fashion.

To make it easier to implement condition **C3** through a syntactic transformation of the system, below we conservatively modify this condition and combine it with **C2**, observing that both **C2** and **C3** give conditions under which $ample(s) = enabled(s)$ must hold, i.e. all enabled transitions are to be explored.

A subset of transitions $\hat{T} \subseteq T$ is called *a set of sticky transitions* if it satisfies these two conditions:

- $\hat{T}$ includes all visible transitions.
- Given an execution sequence $\sigma = s_0 \stackrel{\alpha_0}{\rightarrow} s_1 \stackrel{\alpha_1}{\rightarrow} \cdots$ of M, if $s_k = s_0$ for some $k > 0$, then there exists i, $0 \leq i < k$ such that $\alpha_i \in \hat{T}$. In other words, any cycle in the full state graph M includes at least one sticky transition.

Now, we state the combined condition **C2′** as follows:

C2′ (Visibility and cycle closing). There exists a set of sticky transitions $\hat{T}$ such that for any state s, if $ample(s)$ contains a transition from $\hat{T}$ then $ample(s) = enabled(s)$.

Thus, even when being ample, transitions in $\hat{T}$ "stick" to all other transitions enabled at state s and force their exploration. From the definition of $\hat{T}$, it follows that any cycle in the reduced graph M' will execute at least one sticky transition. Since M' is only explored through ample transitions, **C3** is implied. **C2** is also implied, since $\hat{T}$ also includes all visible transitions. Thus, the partial order reduction can be performed under the three conditions **C0**, **C1** and **C2′**.

A set of sticky transitions $\hat{T}$ may be calculated at compile time. One efficient method of doing this is described in the next section.

2.4. Calculating sticky transitions

Recall that a transition is visible iff it changes the value of an atomic proposition used in a specification ϕ. We have seen that the set $\hat{T}$ must contain all visible transitions. In practice, a concurrent system consists of communicating processes, and atomic propositions appear as boolean predicates over the (data) variables and control points of the processes. We illustrate this using an example. Consider the concurrent system in figure 2, which is composed of two processes P and Q, and three variables x, y and z. Process P is a loop that repeatedly

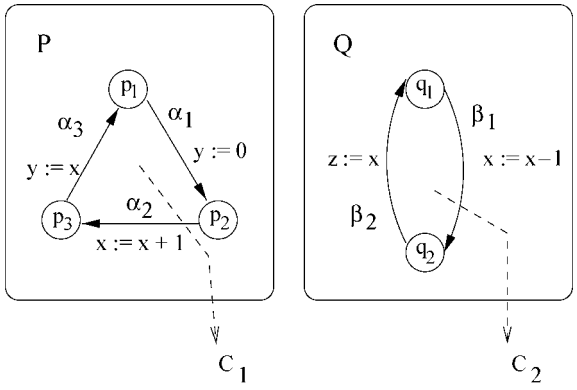

Figure 2. Control flow graphs of the example.

executes the three sequential assignments "$y := 0$", "$x := x + 1$" and "$y := x$". Similarly, process Q is a loop that executes the assignments "$x := x - 1$" and "$z := x$". In figure 2, the local transitions in the control flow graphs of P and Q are annotated with the corresponding assignments. The underlying state graph of this system has global state vectors of the form (c_1, c_2, x, y, z) where c_1 and c_2 are the control points of processes P and Q, respectively, and x, y and z are the values of the corresponding variables. The global transitions, denoted by α_1, α_2, α_3, β_1 and β_2, are in fact the five assignments, which are bound to and executed at the corresponding local transitions.

Suppose that a specification of the system in figure 2 is the LTL formula $\diamond p$, with p the atomic proposition $y > 3$. The only transitions that can directly affect the value of this predicate are α_1 and α_3. Therefore, these are the only visible transitions with respect to the given specification.

A transition may leave the value of an atomic proposition unchanged even though it assigns a variable referenced in it. This is hard to analyze at compile time. We conservatively mark as visible all local transitions that assign (data) variables used in the atomic propositions of a specification. An atomic proposition may also refer to specific control points, for example, be of the form "process P stays at point p_1". In such a case, it can be statically analyzed whether the execution of a local transition, for example, (p_2, p_3), changes the value of the atomic proposition. If it does, it must be marked as a visible transition.

Below, we assume that static analysis conservatively calculates the set of local visible transitions E_v, such that all local images of each visible global transition belong to E_v.

Besides visible transitions, the set of sticky transitions $\hat{T}$ includes for each cycle in the full state space of M at least one transition that is executed on that cycle. Consider figure 3 that presents a *global cycle* in the state graph from figure 2. It is easy to see that the local

$$(p_1, q_1, 0, 0, 0) \xrightarrow{\alpha_1} (p_2, q_1, 0, 0, 0) \xrightarrow{\alpha_2} (p_3, q_1, 1, 0, 0) \xrightarrow{\beta_1}$$
$$(p_3, q_2, 0, 0, 0) \xrightarrow{\beta_2} (p_3, q_1, 0, 0, 0) \xrightarrow{\alpha_3} (p_1, q_1, 0, 0, 0)$$

Figure 3. A global cycle in the underlying state graph.

images of the global transitions that appear in this global cycle form a *local cycle* in the control flow graph G_P (and likewise for G_Q). This is natural, since transition α_1 moves the control point of process P from p_1 to p_2. In order to complete the global cycle, the control point of P has to be restored back to p_1. This is only possible by executing a sequence of global transitions whose local images form a local cycle in the graph G_P. In general, one can observe the following:

Lemma 1. *If the global transitions* $\{\alpha_1, \ldots, \alpha_k\} \subseteq T$ *are executed on a global cycle, and if process* $P \in act(\alpha_i)$ *for some* α_i*, then the local images of the transitions* $\{\alpha_1, \ldots, \alpha_k\}$ *form a (local) cycle in the control flow graph* G_P.

Let E be the set of edges (i.e. local transitions) in all control flow graphs $G_{P_1}, \ldots G_{P_n}$, and $E_v \subseteq E$ the set of local visible transitions. Assume that a subset $\hat{E} \subseteq E$ is chosen such that $E_v \subseteq \hat{E}$ and, for each graph G_{P_i}, removing the edges in $\hat{E}$ from G_{P_i} results in an acyclic sub-graph. Then the following holds.

Lemma 2. *The set* $\hat{T}$ *of all global transitions whose local transitions are in* $\hat{E}$*, forms a set of sticky transitions.*

Proof: Since any global cycle in system M must be projected onto a local cycle in each control flow graph active along this global cycle (see Lemma 1), and these local cycles are broken by some local transitions in $\hat{E}$, the global cycle must contain one or more global transitions with those local transitions. However, all such global transitions have been included into $\hat{T}$. Since $\hat{T}$ also includes all the global visible transitions, as all their local images belong to E_v, it follows that $\hat{T}$ forms a set of sticky transitions. □

For example, in the two control flow graphs given in figure 2 there are two local cycles C_1 and C_2. Then the set $\hat{T} = \{\alpha_1, \alpha_2, \alpha_3, \beta_1\}$ can be used as a sticky transition set that guarantees condition **C2′**. Here, α_1 and α_3 are included into $\hat{T}$ since their local images, the edges (p_1, p_2) and (p_3, p_1), must be conservatively marked as visible and, hence, included into E_v. Global transitions α_2 and β_1 also appear in $\hat{T}$, if their local images (p_2, p_3) and (q_1, q_2) are choosen to break the cycles C_1 and C_2, respectively.

Lemma 2 effectively suggests to select sticky transitions from the set of local transitions E. This can be done advantageously at compile time, during the phase of syntactic transformation of system M. Next, we explain a method to calculate the set $\hat{E}$ of local images of sticky transitions.

Since executing a sticky transition from some state forces all enabled transitions at that state to be explored, the number of sticky transitions should be kept small in order not to diminish partial order reduction. In the example above, $\hat{T}$ must contain α_1 and α_3 as visible transitions. Since transitions α_1 and α_3 already break the cycle C_1, transition α_2 is no longer needed in $\hat{T}$. Thus, $\{\alpha_1, \alpha_3, \beta_1\}$ is a smaller set of sticky transitions. However, we can do better still. If we can conclude that any global cycle that includes α_2 must also include β_1 (say), then we can further reduce $\hat{T}$ to $\{\alpha_1, \alpha_3\}$ which corresponds to $\hat{E} = \{(p_1, p_2), (p_2, p_3)\}$. A method to calculate $\hat{E}$ with this type of reduction is explained next.

Finding a minimal set of edges breaking the cycles in a graph is known as *the feedback arc set problem*, which is shown to be NP-hard in [13]. Our algorithm uses known heuristic methods to compute such sets, as well as a static analysis of the data effects of transitions, as explained below.

In order to calculate the set of local sticky transitions $\hat{E}$, we start by including in it all the local visible transitions E_v, and then remove those transitions from the local control flow graphs $G_{P_1}, G_{P_2}, \ldots, G_{P_n}$. Then, we analyze the resulting local graphs $G'_{P_1}, G'_{P_2}, \ldots, G'_{P_n}$ to find local transitions that break the remaining local cycles: the found transitions are also included into $\hat{E}$. For this we can use heuristics, such as in [2, 6], to minimize the number of transitions we select. Alternatively, we can simply perform a depth-first search on the remaining transitions in the local graphs $G'_{P_1}, G'_{P_2}, \ldots, G'_{P_n}$, and select the backward edges (edges that go back to a node that is in the DFS stack) found in the course of DFS. This may not produce an optimal result, but is linear in the number of edges in the local graph, and performs well in practice. This algorithm can be improved as explained next.

The computation of $\hat{E}$ as described above conservatively considers only the effect of transitions on the control part of the state vector. However, a sequence of global transitions only closes a global cycle if the data variables also revert to the same value. A smaller set of local sticky transitions may be obtained by taking into account the effect of transitions on the data variables. To preserve the static character of our method, we restrict ourselves to effects that can be analyzed at compilation time. To illustrate this, observe that in the global cycle given in figure 3, transition α_2 changes the value of x to 1, whereas transition β_1 restores it to 0. Examining the assignment syntax of the local transitions in figure 2, we deduce that any global cycle that includes the global transition α_2 must also include the execution of the opposite transition β_1 (and vice versa), since it is not possible to have a global cycle along which x is only incremented or only decremented. We have already seen that if α_2 appears in a global cycle, then α_1 and α_3 must also be executed along the same cycle. Thus, we can further reduce the set of sticky transitions for this system to $\hat{T} = \{\alpha_1, \alpha_3\}$.

In general, assume that we are given a subset of variables $\check{V} \subseteq \{V_1, \ldots, V_m\}$, heuristically viewed as important for partial order reduction, such that for each variable $x \in \check{V}$, a partial order $\prec$ is defined on the possible values of x. This order also induces an equivalence relation $\simeq$: namely, $x_1 \simeq x_2$ iff $x_1 \not\prec x_2 \wedge x_2 \not\prec x_1$. We will commonly use the terms "greater", "less" and "equivalent" when referring to $\prec$ and $\simeq$.

For example, in a system M based on message passing, a good choice for a variable $x \in \check{V}$ is an input buffer queue of a process, and an order $\prec$ is introduced by the function $\#(x)$ that counts the number of messages currently in the queue. For two values x_1 and x_2 of the queue variable x, we have $x_1 \prec x_2$ iff $\#(x_1) < \#(x_2)$.

The *effect* of a local transition γ on variable $x \in \check{V}$ with respect to order $\prec$ is said to be

- *incrementing (decrementing)*, if after γ is executed, any new value of x is always greater (less) than the previous value of x, with respect to $\prec$,
- *null (no effect)*, if γ never changes equivalence class of value of x, with respect to $\prec$,
- *complex*, otherwise.

A local transition is conservatively classified as having complex effect on a variable x if its effect on x is impossible or difficult to determine statically. This only means that

our static analysis extracts no information for reduction in this case, but does not affect our subsequent reasoning. For example, in figure 2, the local transitions that correspond to assignments β_1, α_2 and β_2 have, respectively, decrementing effect, incrementing effect and no effect on x with respect to the natural total order on integers, $<$. The local transition (that corresponds to) α_3 has complex effect on y with respect to $<$ since α_3 assigns to the variable y but no more information about its effect can be immediately deduced.

We assume that static analysis can generate a mapping Φ that captures effects of local transitions E on variables in $\check{V}$:

$$\Phi : E \times \check{V} \to \{\text{“incrementing”, “decrementing”, “null”, “complex”}\}$$

The mapping Φ, the set of visible local transitions E_v and the local control flow graphs $G'_{P_1}, \ldots, G'_{P_n}$ (with visible transitions removed) together constitute the input to the algorithm *ComputeStickySet* explained below.

A local transition is called a *monotonic* transition if it has incrementing or decrementing effect on at least one variable in $\check{V}$.

Two local transitions are called *opposite* to each other iff there exists a variable $x \in \check{V}$ such that one transition has incrementing or complex effect on x, and the other has decrementing or complex effect on x. A non-monotonic transition with complex effect on at least one variable in $\check{V}$ is considered opposite to itself.

When selecting sticky transitions that will break all cycles in the global state space, one can observe that a transition α in the global state graph M that changes the value of some variable x can only be in a global cycle with a transition that compensates the effect of α on x. From this follows that every global cycle that executes a monotonic local transition γ must also execute a local transition opposite to γ, in order to return to the same global state. Hence, some monotonic transitions may be removed and yet enough local transitions may remain to break all global cycles in M. This observation is problematic to use, since the number of cycles in a graph can be exponentially bigger than the size of the graph. Instead, in our algorithm *ComputeStickySet* we perform a depth-first search in which backward edges are selected to break the local cycles that remain after removing some monotonic transitions. The following lemma supports this idea:

Lemma 3. *For any reachable cycle C in a directed graph G, one of the edges of C will appear as a backward edge in the course of the depth-first search in G, for any order in which the successors of a vertex are searched.*

Since visible transitions are sticky, cycles that contain a visible transition are automatically handled. Therefore, all visible transitions E_v can be eliminated from the local control flow graphs as a first step.

Let E' be the set of all edges in the resulting control flow graphs $G'_{P_1}, \ldots, G'_{P_n}$, i.e. $E' = E \setminus E_v$. Define the function

$$opptr : E' \to 2^{E'}$$

such that $opptr(\gamma)$ is the set of all local transitions opposite to γ in E'; γ is also included into this set if it is opposite to itself.

Let $\Gamma \subseteq E'$ and G a sub-graph of G_{P_i}. We denote by

- *edges*(G) the set of edges in G,
- *monotonic*(Γ) the set of monotonic transitions in Γ,
- *rem*(G, Γ) the sub-graph remaining after removing the edges Γ from graph G,
- *backedges*(G) the set of backward edges found in G in the course of DFS (we assume that the vertices of G are arbitrarily indexed to direct the DFS).

Algorithm *ComputeStickySet* given next calculates the set $\hat{E}$ of local sticky transitions. The loop in lines 1 to 4 iterates with parameter i taking values in the sequence $1, 2, \ldots n$. Thus, for each graph G'_{P_i} we calculate the sub-graph g_i remaining after removing every monotonic transition γ such that γ may have opposite transitions only in graphs $G'_{P_{i+1}}, \ldots, G'_{P_n}$. In line 5, DFS is applied to each graph g_i to find the backward edges, which are then included into $\hat{E}$ together with local visible transitions.

1. for $i \in (1, \ldots n)$ do
2. $\quad \Gamma := \{\gamma \in \mathit{monotonic}(\mathit{edges}(G'_{P_i})) \mid \mathit{opptr}(\gamma) \subseteq \bigcup_{j>i} \mathit{edges}(G'_{P_j})\}$;
3. $\quad g_i := \mathit{rem}(G'_{P_i}, \Gamma)$;
4. od;
5. $\hat{E} := E_v \cup \bigcup_i \mathit{backedges}(g_i)$;
6. return $\hat{E}$;

Algorithm 1. *ComputeStickySet.*

Theorem 1. *The set $\hat{T}$ of all global transitions that execute the local transitions in $\hat{E}$ which is returned by algorithm* ComputeStickySet *forms a set of sticky transitions.*

Proof: Since every visible transition executes a (local) transition in E_v (see above) and $E_v \subseteq \hat{E}$, $\hat{T}$ includes all visible transitions. We prove next that global transitions in $\hat{T}$ also break all global cycles.

Consider a global cycle C, and let *loc*(C) be the set of local cycles that C is projected onto (cf. Lemma 1). Note that execution of each global transition in C will involve executing local transitions in one or more cycles in *loc*(C), and every local transition γ in each cycle in *loc*(C) will execute along C. Therefore, if γ is monotonic, an opposite transition must also execute along some cycle in *loc*(C).

If some cycle in *loc*(C) includes a (visible) transition from E_v then C is broken, as $E_v \subseteq \hat{E}$.

Now, consider a cycle C such that each cycle in *loc*(C) includes only transitions from the set $E' = E \setminus E_v$, i.e. belonging to graphs $G'_{P_1}, \ldots, G'_{P_n}$. In this case, we prove that C will include a global transition from $\hat{T}$ that executes some local transition in *backedges*(g_i), for some i. Contrarily, assume that C does not execute any local transition in $\bigcup_i \mathit{backedges}(g_i)$, and, hence, none of those local transitions belongs to any cycle in *loc*(C). This means by Lemma 3 that no cycle in *loc*(C) remains completely in $g_1, \ldots, g_n$, which may be the case only if each such cycle has been already broken by the algorithm's action that removes monotonic transitions in line 3.

Consider then a local cycle in $loc(C)$ such that it belongs to graph G'_{P_k} with the largest process index k, and a monotonic transition γ removed from this cycle by the action in line 3 and, hence, included into set Γ in line 2. Now note that for transition γ to be included into set Γ, all transitions opposite to γ (if any) must belong to graphs $G'_{P_{k+1}}, \ldots, G'_{P_n}$. On the other hand, as explained above in the proof, γ must have an opposite transition $\bar{\gamma}$ in (at least) one of the cycles $loc(C)$, hence, in G'_{P_j}, where $j \leq k$. The contradiction finalizes the proof. □

Note that the contents and size of set $\hat{E}$ returned by algorithm *ComputeStickySet* may depend, first, on the order in which control flow graphs $G'_{P_1}, \ldots, G'_{P_n}$ are processed by the loop in lines 1 through 4 (i.e. on the order in which system's processes are indexed) and, second, on the order in which the vertices of the remaining sub-graph g_i are indexed for DFS. Thus, heuristics for both process ordering and vertex indexing may be useful to optimize the algorithm. This issue is left open for now.

3. Hardware/software co-verification methodology

In this section we present a co-verification methodology and a tool that combine static partial order reduction with hardware-oriented verification techniques.

We have developed a co-verification tool based on existing programming environments for VHDL [28], Verilog [27], and SDL [23], and an existing model checking engine, COSPAN [9, 10]. However, the methodology described below is not strongly tied to a particular representation of hardware or software, except with respect to synchrony assumptions about the design. For hardware, it is assumed that the design is clock cycle dependent and thus we use a synchronous model of coordination. In contrast, software is presumed to operate under synchronization conditions guaranteed by its host, and thus coordination among software design components is considered to be asynchronous, and modeled using interleaving. Since different abstraction levels can be used for hardware and software, we do not make an assumption about their relative speeds.

The choice of a hardware description language is strongly influenced by the computer industry, where VHDL and Verilog are in widest use and have been standardized by the IEEE. Our platform supports both, by translating them into S/R, the native language of COSPAN, using FormalCheck^{TM2} translators. For the software part, we use SDL, a standard language of the International Telecommunications Union (ITU-T), an overview of which is presented in the next section. Here, two of the co-authors have developed a model checking tool SDLCheck [19] that implements static partial order reduction, generates a (reduced) S/R model of the original SDL system and then, on the verification phase, makes use of COSPAN. To support co-verification, SDLCheck also implements interface constructs introduced in [18] as an extension to SDL to allow describing an interface between software and hardware components.

The general structure of the tool support for our co-verification technology is shown in figure 4. We assume that the hardware part of a co-design is described either in VHDL or Verilog and the software part in SDL. The interface between hardware and software is described in an extended SDL, SDL+, using interface constructs. In essence, interface constructs allow an SDL+ process to read and write combinational signals from the hardware

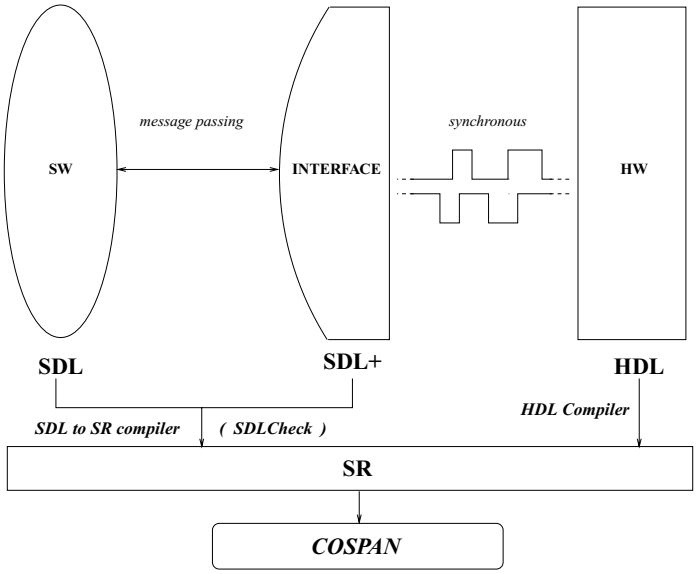

Figure 4. The general structure of the tool support.

part. Below, we refer to such a process as an *interface process*. The coordination of an interface process with software processes and with other interface processes is handled completely by the software communication mechanism, which in the case of SDL is message exchange through buffers. Therefore, an interface process looks like another software process from the software point of view. On the other hand, the interaction of the interface process with the hardware is performed through shared variables called *interface variables*, using the synchronous model of the hardware. Therefore, an interface process appears as another hardware module to the hardware part of the system.

In a co-design system we distinguish between two types of global transitions. All transitions of a single software process are *software transitions*. As explained in Sections 2.3 and 2.4, a software transition changes the control point of the process and may also execute either a local action or a shared communication action. All global transitions of a hardware component are *hardware transitions*. Since hardware has a synchronous nature, a hardware transition corresponds to simultaneous transitions of all hardware modules in the synchronous hardware component. An interface process can have both types of transitions. A transition of an interface process in which an interface variable is referenced counts as a hardware transition, otherwise it counts as a software transition.

Since the static partial order reduction only restricts the software transition relation, as explained in Section 2.3, its implementation is only influenced by the selection of a particular software description language. The details of our SDL-based implementation of static partial order reduction are described in Section 4 below. For now, we only present the general picture.

First, we describe the basic methodology. The hardware, software and interface parts of a co-design system, and the property ϕ to be checked are all translated into a formalism

based on synchronous automata, used as an input interface by a model checker (in our current implementation, the model checker COSPAN and the S/R language). This enables us to treat the entire co-design system, with parts of different nature, as a single, formally synchronous model, that is to be verified with regard to an automaton expressing property ϕ. Software asynchrony is modeled by self-looping, using nondeterminism as proposed in [16]. The synchronous model generated for the software and interface processes is augmented with an additional constraint automaton, which implements static partial order reduction by restricting the software transition relation. This model is then composed with the synchronous hardware model, finalizing the compilation stage of co-verification. Next, the model checking stage starts out by applying localization reduction [14] to the overall combined synchronous model, reducing it relative to the checked property ϕ. For example, in software-centric verification (figure 5, discussed below), which is aimed at properties ensured mainly through control structures in the software part, one may expect much of the hardware to be abstracted by localization reduction. Finally, the remaining reduced model is analyzed using a BDD-based search.

A more advanced methodology can be applied, if a co-design system is so complicated that the simple combination of three reduction techniques, described above, is not sufficient, i.e. may result in state space explosion. In this case, first, the entire co-design system is translated into a synchronous model in a conventional way, i.e. without static partial order reduction, and then localization reduction is applied alone to reduce this synchronous model relative to the checked property ϕ: it conservatively removes variables. As a by-product, the set of removed variables is reported. Using it, the localization reduction is automatically projected onto the original co-design system, which is correspondingly simplified by removing those variables. Localization reduction also reduces the range of values of a surviving variable, by compressing it to the subset really used in the reduced model, and this effect is

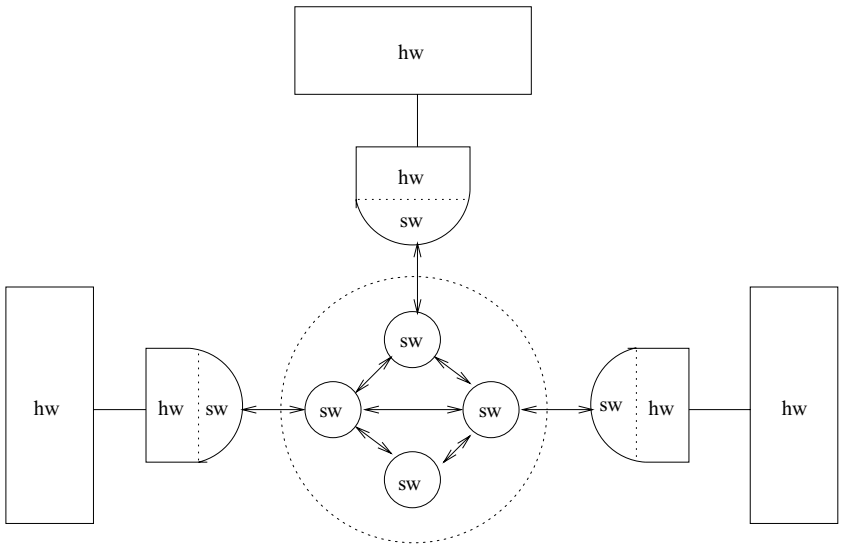

Figure 5. Software-centric view.

also projected onto the original co-design system. At last, the basic methodology is applied, as described above, to this reduced co-design system. Since the system is now smaller, i.e. more abstract, the number of sticky transitions generated by the static partial order reduction will, in some cases, be smaller too, thus reducing the following model checking search.

Combining partial order reduction with BDDs, or, in general, with hardware-oriented reductions has been motivated by the necessity to deal with the complexity problem known as state space explosion. However, this is not the only problem faced when verifying a real system with a model checker. A non-trivial task also arises from the technological side, namely constraining the system to be verified. In practice, a system is always designed under certain assumptions with respect to the behavior of the system environment. The environment provides inputs to the system and, in general, interacts with it, ensuring the external conditions for correct functionality.

Current model checking technology typically suggests capturing the designer's environmental assumptions by constraints expressed in temporal logic [14, 20]. Although this approach works when adequate abstractions are obvious enough, there are many other cases in which it fails. In particular, using temporal logic constraints appears difficult for a subsystem of a larger system. A subsystem is not intended to be delivered to a customer as a product, and the external conditions necessary for its correct operation are therefore often not documented by the designer. Moreover, those conditions typically reflect the internal behavior of the remainder of the system, and are thus often complicated and difficult to express in temporal logic.

Yet, since checking an entire system is often prohibitively complex, such subsystems make up a large fraction of the target domain for model checking. We argue that our co-verification methodology can be used to solve this problem by making it feasible to use as an environment the surrounding subsystems themselves, after replacing some of them either by abstractions in the same design language or by appropriate constraints. This environment is then translated, together with the central subsystem subject to verification, into a combined synchronous model and subject to model checking. This approach can be successful under two conditions: first, the description of the central subsystem and the surrounding subsystems should be easy to extract and modify, and second, the combined model should not lead to irreducible state space overhead. For mixed hardware/software designs described in high-level languages such as VHDL/Verilog and SDL, respectively, we can identify two extremal co-verification cases which generally satisfy the first condition, while our approach seems promising in ensuring the second.

The extremal co-verification cases mentioned above are software-centric and hardware-centric co-verifications, in which the properties to be verified refer entirely or mainly either to the software part or to the hardware part, respectively. The two cases are illustrated in figures 5 and 6, which represent software and hardware components by circles and rectangles, respectively, and interface processes by hybrid shapes (half circle, half rectangle). In the software-centric view, the central subsystem comprises several software processes which communicate directly to each other and/or indirectly, via interface processes, to hardware components. The hardware components are generally also connected to each other. In a software-centric view, they are encapsulated by the layer of interface processes and form together with this layer the environment of the central software subsystem. Symmetrically,

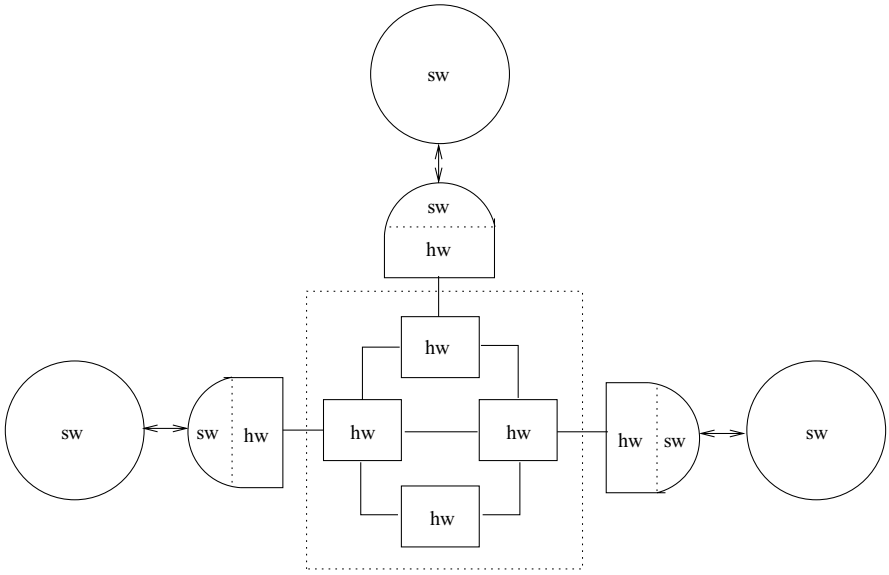

Figure 6. Hardware-centric view.

in the hardware-centric view, the hardware design occupies the central place, whereas the software components together with the interface processes form its environment. One can expect that in either of the two extremal co-verification cases localization reduction combined with static partial order reduction may abstract away a significant part of the state space of the environment.

4. Implementation

Before describing the implementation of our co-verification methodology (realized in SDLCheck [19]), we first briefly present the semantics of SDL, which is used as part of its language platform.

4.1. Overview of SDL

An SDL system is given as a set of processes which are assumed to run concurrently and may have local variables. The communication between processes is handled via message passing through infinite buffers. For the purposes of practical verification we however restrict these buffers to a finite size. Each process has a single FIFO buffer which queues all incoming messages. A message consists of a signal name, and optionally some parameter values, whose number and types are specified in the signal declaration. Below, some commonly used SDL constructs are explained by means of the example[3] given in figure 7. In the example the annotations "*control point i*" reflect the control points of the SDL process. SDL actions between control points correspond to the local images of the global transitions, in the control flow graph.

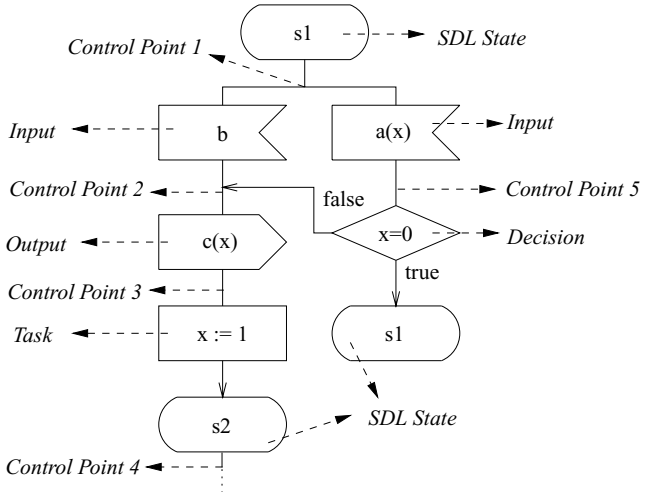

Figure 7. An example SDL process.

An *SDL state* construct is the point where an SDL process can check its incoming message buffer. Therefore, an *input* action, which is used to consume a message from the buffer, is always used together with an *SDL state*. An SDL process can only consume the first message in the buffer. There are some constructs (e.g. *save*, *priority input*) that override this restriction. An *input* construct specifies a signal name, and possibly local variables to be assigned, if the signal is defined to carry parameters. In our example, a is such a signal, which carries one parameter. When the process consumes signal a at the *SDL state* $s1$, the parameter value carried by a is assigned to local variable x of the process, the message is removed from the buffer head and the process moves from *control point 1* to *control point 5*.

An *output* action is used by a process to send a message to another process. It specifies the name of the signal, and also the parameter values if the signal being sent is capable of carrying parameters. In our example, the message sent at *control point 2* will have the signal name c and will carry as a parameter the current value of the local variable x at the time the *output* action is executed.

A *task* action is used to assign to local variables of a process. A *decision* action is similar to an if-then-else statement of usual programming languages. In our example, if the process stays at *control point 5*, and the current value of x is 0, then the process moves back to *SDL state* $s1$, or, in other words, makes a transition to *control point 1*. If the value of x is nonzero, the process moves to *control point 2*.

There are a number of other constructs (e.g., *none input* and *any decision* for nondeterminism, *procedure call*, etc.) to describe the behavior of a process, but we don't present them here, although these constructs are handled by our implementation.

An SDL system may contain, beside processes, entities called *channels*. These are used to describe the communication paths between processes. There are two types of channels, delaying and non-delaying. In our implementation, a delaying channel is viewed as another SDL process which receives messages and forwards them to their receivers.

4.2. Static partial order reduction in the SDL to S/R compiler

In this section we explain how we implement our method of changing the enabling conditions of transitions so that the generated code has the partial order reduction incorporated.

As explained in Section 4.1, a transition of an SDL process is always defined to occur between two control points of the process. The ample sets that we define will always contain all enabled transitions of a single process. Therefore, we say that an SDL process P is *ample* at a state if its current enabled transitions form an ample set. Hence, we need to identify the local states of P for which the current enabled transitions satisfy the conditions **C0**, **C1** and **C2′**.

We start with condition **C1** and examine the types of transitions introduced in Section 4.1. We will check if a transition is *independent* of all transitions of other processes in the entire system and if this is the case, tag it as `GoodForC1`. An *input* transition is always independent of all transitions of any other process. Note that an input transition may change the local variables of that process if the consumed signals carries parameters. However, this does not cause a problem since no other process can read[4] or write to a local variable of P. Another effect of an input transition is the removal of the message at the head of the buffer. The buffer of P is also accessed by any process P' that can send a message to P, therefore it may seem that the execution order of these two transitions is important. However, an output transition places the message at the tail of the buffer, whereas an input transition removes the message from the head of the buffer. Therefore, in both execution orders, the resulting buffer content will be the same. There is no other action of another process that can be dependent on an input transition of P. Hence, an input transition can indeed be tagged as `GoodForC1`.

The *task* and *decision* transitions of process P can only access local variables of P. A *task* updates its local variables, and a *decision* changes the current control point of P. Since no other process can directly access the local variables nor the control point of process P, its task and decision transitions are independent of all other transitions. So, we can tag all such transitions as `GoodForC1` provided that they appear in a software SDL process. However, in an interface SDL process, which may have interface variables shared by the hardware part, any transition that accesses (reads or assigns to) an interface variable is never tagged as `GoodForC1`, see Section 3.

An *output* transition of P which sends a message to a process P_0, is dependent on an output transition of any process P' that also sends a message to P_0. The reason is that different execution orders of these transitions will result in a different contents for the input buffer of P_0, since the messages are queued in their arrival order. We therefore tag an output transition from P to P_0 as `GoodForC1` only if there is no other process P' that outputs to the same process P_0.

Note that in some of these cases a transition can *enable* another transition. For example, an output transition can enable an input transition, by providing a message in a buffer that was previously empty. This doesn't affect the reduction technique we use in our implementation, in contrast to some other approaches, because the underlying definition of the independence relation given in Section 2.1 is relaxed: it specifies that transitions may not disable another, but does not prohibit enabling.

The next step is to find the visible transitions. We assume that the atomic propositions of the LTL formula ϕ to be checked are expressed in terms of the control points and local variables of processes (e.g., "Process P is at control point 1" or "Local variable x of process P has value 5"). Hence, we tag a transition as `Visible` if it moves the control point of a process to or from a control point mentioned in an atomic proposition, or it assigns to a variable that is mentioned in an atomic proposition. Note that since visible transitions are determined at compile time, and different properties contain different atomic propositions, each LTL formula requires a different compilation run.

After completing the tagging with `GoodForC1` and `Visible`, we perform a similar tagging for condition **C2′**. This time we tag transitions with `Sticky` such that the set $\hat{T}$ of transitions with this tag guarantees **C2′**. We analyze the types of transition data effects to reduce this set as explained in Section 2.4. With this regard, we only consider input and output transitions as having opposite effects on process input buffer queue, with respect to a given signal A: output of signal A to process P is treated as having incrementing effect, whereas input of signal A in P as having decrementing effect.

This completes the tagging steps. We call a control point of a process an *ample* control point, if all local transitions from this control point are tagged as `GoodForC1` and not tagged as `Sticky`.

Let the system be composed of processes $\{P_1, P_2, \ldots, P_N\}$. We define for each process P_i a new boolean variable $ample_i$. P_i assigns *true* to $ample_i$ iff its current local state is at an ample control point. All other processes can read the current value of $ample_i$. We also impose an artificial priority ordering on the set of processes, for example, as in the list $P_1, P_2, \ldots, P_N$. Finally, we define the new enabling condition of a local transition in a process P_i as follows: if the original enabling condition of the transition is a boolean predicate p, it is changed to $p \wedge (p_{C1C2'} \vee p_{C0})$, where

$$p_{C1C2'} = \neg ample_1 \wedge \neg ample_2 \wedge \cdots \wedge \neg ample_{i-1} \wedge ample_i$$
$$p_{C0} = \neg ample_1 \wedge \neg ample_2 \wedge \cdots \wedge \neg ample_N$$

As the name implies, $p_{C1C2'}$ is used to guarantee conditions **C1** and **C2′**. At a given global state, there may be more than one ample process. Since the enabled transitions of each ample process form an ample set at that state, we need to execute only one of them: it is selected as the first ample process in the list of processes $(P_1, P_2, \ldots, P_N)$ given in the (artificial) priority order. If no process is ample at the current state, all enabled transitions are executed. This is guaranteed by p_{C0}, which is true in this case.

Our compiler generates the code in S/R. To implement the $ample_i$ variables we use the *selection variables* of S/R. These are combinational variables which are not part of the state and thus don't take up space in the state vector during the search. Hence, the introduction of $ample_i$ flags does not introduce any memory overhead.

5. Experimental results

We have evaluated our method with several examples specified in SDL and translated into S/R using the static partial order reduction method described in this paper. Several case

studies have been also conducted by our colleagues at Bell Laboratories and partners at the University of Texas at Austin. Below, four of our examples are described and verification results of two other case studies are summarized.

All examples presented below were run on a VA Linux machine with Intel® Pentium III processor at 800 MHz and 2300 MB of memory available for a process.

The first example is a concurrent sorting algorithm, *SortN*. There is a communication chain comprising $N + 1$ processes that sort N randomly generated numbers. One process simply generates N random numbers and sends them to the next process in the chain. Each process that receives a new number compares it with the current number it has and sends the greater one to the next process. The last process receives only one number which is the largest one generated by the first process.

The second example is a leader election protocol, *LeaderN*, given in [5]. It contains N processes, each with an index number, that form a ring structure. Each process can only send a signal to the process on its right and can receive a signal from the process on its left. The aim of the protocol is to find the largest index number in the ring. The protocol is verified with respect to all possible initial states, i.e. each process initially selects an index number nondeterministically.

Each of these two examples is naturally scalable through the number of processes N. We have attempted to verify each example for various indices N using both explicit search and symbolic search, with and without static partial order reduction. We have omitted checks for N if the same combination of methods did not work for $N - 1$: i.e. ran out of memory or time. Table 1 presents the results of those experiments. The results for the sorting algorithm suggest an evidence that in some cases, verification conducted by symbolic search combined with static partial order reduction may outperform verification conducted by symbolic search alone as well as verification conducted by explicit search combined with static partial order reduction. However, this is not always the case, as suggested by the results for the leader election protocol.

On these two examples, *SortN* and *LeaderN*, static partial order reduction implemented in SDLCheck gives a reduction in the number of states that is as good as explicit search with dynamic partial order reduction as implemented in Spin [11]. This is due to the fact that in these two examples the majority of local loops in the component processes are broken by visible actions and the remaining loops appear inter-dependent through output and input actions in neighboring processes, cf. Section 4.2. Therefore, in these particular cases, dynamic breaking the global state transition cycles (cf. condition **C3** in Section 2.2) does not give rise to a decisive advantage over breaking few local cycles by the sticky transition analysis (cf. Algorithm 1 in Section 2.4). However, a general comparison between dynamic and static partial order reduction has not been done here, and it is not known how the relative performance of these two techniques would compare over a broad set of models.

Our next two examples are combined hardware/software systems. The verification results for them are given by Table 2.

The third example is the alternating bit protocol, *HW-ABP*. The system was architectured hardware-centric in [18]. Its core transmission part is given by a register transfer level gatelist that may be expressed in Verilog or VHDL. Both sending and receiving end of the hardware transmitter are controlled by their own drivers implemented in SDL+ as interface

Table 1. Experimental results for software verification.

Experiments and methods	States	Edges/bdd nodes	Time(s)	Memory(M)
Sort6				
Explicit	Killed after		84922	1300
+SPOR	2.50388e+6	122690120	2436	121
Symbolic	7.33483e+7	13623	4.2	3.3
+SPOR	2.50388e+6	33851	3.2	3.9
Sort8				
Explicit	Not run			
+SPOR	Killed after		59871	976
Symbolic	2.98508e+11	38641	23	7.1
+SPOR	1.12819e+9	80072	38	5.3
Sort10				
Symbolic	2.07538e+15	68436	124	13
+SPOR	8.57609e+11	136978	21	7.5
Sort12				
Symbolic	2.20155e+19	115731	547	19
+SPOR	9.08039e+14	180557	54	9.4
Leader4				
Explicit	9197	13844	1.9	0.8
+SPOR	5571	6708	1.4	0.6
Symbolic	9197	141889	51	20
+SPOR	5571	218087	207	25
Leader5				
Explicit	247276	429360	71	16
+SPOR	112217	143490	30	7.5
Symbolic	247276	1025010	8359	816
+SPOR	112217	1234204	15935	521
Leader6				
Explicit	9212107	18285510	5828	664
+SPOR	2834882	3766265	1080	204
Symbolic	Out of memory		8994	
+SPOR	Out of memory		10903	

processes. Each driver serves one software user process written in SDL that implements, respectively, either the initial sender or the terminal receiver of 1-bit messages. The system has been verified using all four combinations of methods (see Table 2). In this case, static partial order reduction provided no improvement. The reason is that the software part, although comprising four SDL processes, has little concurrency, being split into two (sending

Table 2. Experimental results for co-verification.

Experiments and methods	States	Edges/bdd nodes	Time(s)	Memory (M)
HW-ABP				
Explicit	346318	47294	143	14
+SPOR	346318	47294	145	14
Symbolic	346318	10680	2.8	3.2
+SPOR	346318	10680	2.9	3.2
Elevator				
Explicit	Killed after		83544	47
+SPOR	Killed after		126847	75
Symbolic	Out of memory		9853	
+SPOR	1.00593e+09	5102631	14403	415

and receiving) parts, each tightly coupled with hardware. The major role is conducted by hardware. This explains the better performance of symbolic over explicit search.

The fourth example is a system of two elevators in a four story building. The system is given in a software-centric architecture as follows. Elevator cabins have request buttons for each floor, and on every floor there is an external elevator call button. When an external call button is pressed, one of the cabins is assigned to go to that floor according to the position of the cabins and the current selections of the people in the cabin. The system requirement is that all requests are eventually serviced. The two cabins and the four external call buttons are hardware components. Each cabin is modeled by a synchronous sub-system of finite states machines (FSM's) and each call button by a single FSM. The hardware components are separated from each other. Each of them is conditioned on a common clock and progresses along the clock pulses. On the software side, there is a distinct driver for each hardware component, which is specified in SDL+ as an interface process. The drivers support the communication between hardware components and the central software process that handles requests from control buttons and assigns elevators. For this example, symbolic search combined with partial order reduction completes the verification in four hours consuming 415 MB of memory, whereas the three other (combinations of) methods could not complete at all, see Table 2.

Now, we present the verification results from two case studies performed by our partners from the University of Texas at Austin, Fei Xie and Natasha Sharygina. They developed OO design models of two software systems using the Objectbench^{TM} tool[5] (which supports an executable UML-style graphical interface that follows the Shlaer-Mellor method [25]). Using a tool that translates OO design models into an intermediate form acceptable by SDLCheck, they applied SDLCheck (which implements static partial order reduction) to generate reduced S/R models of their designs and then Cospan to verify them.

The case study conducted by Natasha Sharygina is described in a detail in [24]. It is an OO design model of the robot control system developed by the robotics research group at the University of Texas at Austin. The design describes a simplified version of the system

that controls a robot with one arm. The arm consists of two joints and an end effector that moves around and performs designated functions such as grabbing. Two major robotics algorithms are captured in the design: arm control and fault recovery. In total, the design model includes seven process components (called *active objects*). The active objects interact by sending to each other messages of 31 different types. A typical active object consists of 7 object states, at which messages sent to the object may be read. Fourteen different properties that express functions of the system have been successfully verified. In all the cases, the best performance has been shown by explicit search combined with static partial order reduction. Depending on a property, verification times range from 336 seconds to almost 70 hours, and memory consumption from 0.2 MB to 198 MB.

The case study conducted by Fei Xie is an on-line ticket sale system. The system design includes four classes of active objects: a single dispatcher and ticket server, and multiple agents and customers. When the dispatcher receives a purchase request from a customer, it assigns an agent to serve the customer. The agent guides the customer through a ticket purchase transaction with the ticket server. Agents are also responsible for keeping misbehaving customers from corrupting the system. A simplified version of the system (with only one agent and one customer) has been successfully verified with regard to five properties under a general fairness assumption that guarantees that none of the active objects enabled to execute is ignored. An example of the verified properties is: if the agent receives infinitely many purchase requests then it provides infinitely many transaction services. In this case, symbolic search combined with static partial order reduction performs in 1765 seconds consuming 77 MB of memory, whereas explicit search also combined with static partial order reduction spends 6682 seconds and 17 MB. Neither of the two methods, symbolic or explicit, could finish the verification when applied alone, without partial order reduction.

6. Conclusion

We have presented a formal verification methodology that supports the combination of hardware- and software-oriented model checking techniques. An important consequence is to be able to run together symbolic BDD-based verification and partial order reduction. Both techniques address the same problem of exponential computational complexity of state space exploration, but use search algorithms that are intrinsically different, operating in breadth-first and, respectively, depth-first manner. To merge these two quite different techniques into one methodology, we apply them successively, in two different stages of the verification process. Partial order reduction is handled completely in the compilation phase, which generates a reduced model with the transition relation constrained by the ample set conditions **C0–C2′**. Subsequently, the model checking phase can employ BDD-based algorithms as well as any other reduction techniques with no change to the existing verification engine.

We suggest the use of this method primarily for models with large state spaces that cannot be handled by either reduction technique alone: symbolic evaluation without partial order reduction or partial order reduction with explicit state enumeration. For such models, combining symbolic evaluation with partial order reduction may make verification feasible. Our experiments and case studies show that such models exist, for which the proposed

combined method may succeed, whereas either of its two component techniques applied alone fails (runs out of memory or time) or performs too slowly. However, the experiments also show that this combined method is not always the best. Characterization of the models for which it suits better than other methods is an open problem.

One natural niche for the proposed verification methodology are software/hardware co-design systems. Indeed, partial order reduction reflects (and reduces!) interleaving, commonly used to model software concurrency, and symbolic evaluation is important for hardware verification, where partial order reduction has little or no effect. Our fourth example in the previous section demonstrates not only the efficiency of the joint approach to reduction, but also the fact that even a simple co-design model may fall out of the scope of a stand-alone verification technique. However, we expect that the application area is wider. It may also include some software systems whose key issue is interprocess communication, and complicated hardware designs with major communication components represented by interleaving abstractions.

We have also presented a toolkit that supports the proposed verification methodology in the area of software/hardware co-design. The toolkit comprises Verilog and VHDL compilers for translation of hardware components, a tool that translates software parts and generates a model incorporating partial order reduction, and finally a model checking engine with support for both symbolic evaluation and explicit search. Although the implementation is necessarily language specific, the basic ideas explained above are general enough to allow re-implementations for different languages and/or BDD-based model checkers. In fact, a re-implementation using Objectbench^{TM} prior to the SDL interface has already been done at Bell Laboratories in collaboration with Professor James C. Browne and his students at the University of Texas at Austin.

Acknowledgments

We thank James Browne, Fei Xie and Natasha Sharygina for their valuable contributions that assisted our method to mature, and the reviewers for their useful comments.

Notes

1. *www.cadence.com*, *www.mentor.com*, *www.synopsys.com*, etc.
2. The FormalCheck^{TM} verification system is licensed by Lucent Technologies to Cadence Design Systems, Inc.
3. The example is given in the graphical syntax recommended by [23], but italic labels and dashed arrows are just annotations.
4. SDL has two constructs, *view* and *import*, that enable a process to see a variable owned by another process. The import construct is a shorthand of message passing that requests the owner process to send the current value of the variable. The view construct, which provides direct read access, is an outdated feature of the first version of SDL, not recommended (though allowed) by the standard [23]. Currently, we ignore these constructs, although both can be handled by the method of Section 2.3.
5. Objectbench^{TM} is owned by Scientific and Engineering Software, Inc.

References

1. R. Alur, R.K. Brayton, T.A. Henzinger, S. Qadeer, and S.K. Rajamani, "Partial-order reduction in symbolic state space exploration," in O. Grumberg (ed.), *Computer Aided Verification*, 9th International Conference,

(CAV '97) Proceedings, Vol. 1254 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1997, pp. 340–351.
2. B. Berger and P.W. Shor, "Approximation algorithms for the maximum acyclic subgraph problem," in *First ACM-SIAM Symp. on Discrete Algorithms. Proceedings*, 1990, pp. 236–243.
3. C.-T. Chou and D. Peled, "Formal verification of a partial-order reduction technique for model checking," in T. Margaria and B. Steffen (Eds.), *Tools and Algorithms for the Construction and Analysis of Systems*, Second International Workshop (TACAS '96) Proceedings, Vol. 1055 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1996, pp. 241–257.
4. D.L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. MIT Press, Cambridge, MA, 1989.
5. D. Dolev, M. Klawe, and M. Rodeh, "An $O(n \log n)$ unidirectional distributed algorithm for extrema finding in a circle," *Journal of Algorithms*, Vol. 3, No. 3, pp. 245–260, 1982.
6. P. Eades, X. Lin, and W.M. Smyth, "A fast and effective heuristic for the feedback arc set problem," *Information Processing Letters*, Vol. 47, No. 6, pp. 319–323, 1993.
7. D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi, "On the temporal analysis of fairness," in *Conference Record of the Seventh ACM Symposium on Principles of Programming Languages*, 1980, pp. 163–173.
8. P. Godefroid and D. Pirottin, "Refining dependencies improves partial-order verification methods," in C. Courcoubetis (Ed.), *Computer Aided Verification*, 5th International Conference (CAV '93) Proceedings, Vol. 697 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1993, pp. 438–449.
9. R.H. Hardin, Z. Har'El, and R.P. Kurshan, "COSPAN," in R. Alur and T.A. Henzinger (Eds.), *Computer Aided Verification*, 8th International Conference (CAV '96) Proceedings, Vol. 1102 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1996, pp. 423–427.
10. Z. Har'El and R.P. Kurshan, "Software for analytical development of communication protocols," *AT&T Technical Journal*, Vol. 69, No. 1, pp. 45–59, 1990.
11. G.J. Holzmann, "The model checker Spin," *IEEE Trans. on Software Engineering*, Vol. 23, No. 5, pp. 279–295, 1997.
12. G.J. Holzmann and D. Peled, "An improvement in formal verification," in D. Hogrefe and S. Leue (Eds.), *Formal Description Techniques VII*, Proceedings of the 7th IFIP WG 6.1 International Conference Bern, Switzerland, 1994, pp. 197–211.
13. R.M. Karp, "Reducibility among combinatorial problems," in *Complexity of Computer Computations*, Plenum Press, New York, 1972, pp. 85–103.
14. R.P. Kurshan, *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*, Princeton University Press, Princeton, NJ, 1994.
15. R. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigün, "Static partial order reduction," in B. Steffen (Ed.), *Tools and Algorithms for the Construction and Analysis of Systems*, 4th International Conference (TACAS'98) Proceedings, Vol. 1384 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1998, pp. 345–357.
16. R.P. Kurshan, M. Merritt, A. Orda, and S. Sachs, "Modeling asynchrony with a synchronous model," in P. Wolper (Ed.), *Computer Aided Verification*, 7th International Conference (CAV'95) Proceedings, Vol. 939 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1995, pp. 339–352.
17. L. Lamport, "What good is temporal logic," in R.E.A. Mason (Ed.), *Proceedings of IFIP Congress*, North Holland, 1983, pp. 657–668.
18. V. Levin, E. Bounimova, O. Başbuğoğlu, and K. İnan, "A verifiable software/hardware co-design using SDL and COSPAN," in *Proceedings of the COST 247 International Workshop on Applied Formal Methods in System Design*, Maribor, Slovenia, 1996, pp. 6–16.
19. V. Levin and H. Yenigün, "SDLCheck: A model checking tool," in G. Berry, H. Comon, and A. Finkel (Eds.), *Computer Aided Verification*, 13th International Conference (CAV 2001) Proceedings, Vol. 2102 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2001, pp. 378–381.
20. K.L. McMillan, *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, MA, 1993.
21. D. Peled, "Combining partial order reductions with on-the-fly model-checking," *Formal Methods in System Design*, Vol. 8, pp. 39–64, 1996.
22. D. Peled and T. Wilke, "Stutter–invariant temporal properties are expressible without the next-time operator," *Information Processing Letters*, Vol. 63, No. 5, pp. 243–246, 1997.

23. SDL92, "Functional specification and description language (SDL), ITU-T Recommendation Z.100," 1993, Geneva.
24. N. Sharygina, R.P. Kurshan, and J.C. Browne, "A formal object-oriented analysis for software reliability: Design for verification," in Heinrich Husmann (Ed.), *Fundamental Approaches to Software Engineering*, 4th International Conference (FASE 2001) Proceedings, Vol. 2029 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2001, pp. 318–332.
25. S. Shlaer and S.J. Mellor, *Object Lifecycles Modeling the World in States*, Prentice-Hall, Englewood Cliffs, NJ, 1992.
26. A. Valmari, "A stubborn attack on state explosion," in E.M. Clarke and R.P. Kurshan (Eds.), *Computer-Aided Verification*, 2nd International Conference (CAV '90) Proceedings, Vol. 531 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1990, pp. 156–165.
27. Verilog95, "IEEE standard hardware description language based on the Verilog™ hardware description language, "IEEE Std 1364-1995," 1996, New York.
28. VHDL93, "IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-1993," 1994, New York.