Efficient Hybrid Reachability Analysis for Asynchronous Concurrent Systems*

Enric Pastor and Marco A. Peña

Department of Computer Architecture
Technical University of Catalonia
08860 Castelldefels (Barcelona), Spain
{enric,marcoa}@ac.upc.es

Abstract. Symbolic reachability analysis based on Binary Decision Diagrams (BDDs) is a technique that allows the implementation of efficient state space exploration algorithms. However, in practice it is well known that the BDD blowup problem limits the size of the systems that can be analyzed. Conversely, simulation is a low-cost state generation technique, although its effectiveness is limited due to its inherent sequentiality. We present a hybrid methodology that combines simulation and symbolic traversal in order to improve the state space exploration of large systems. The methodology concentrates on asynchronous concurrent systems, whose peculiarities are not fully exploited by other existing techniques for hybrid verification. Our approach exploits the information obtained from simulations to improve the knowledge of the state space, effectively guiding symbolic traversal. We demonstrate the applicability of this methodology in the verification of complex control-dominated asynchronous circuits.

1 Introduction

State space computation is the main bottleneck for most formal verification techniques. As an example, for invariant verification all reachable states of the system are calculated and the desired invariants are checked to hold in all of them. If the system fails to satisfy the invariants, it is necessary to identify a counter-example that reproduces the sequence of actions that the system performs before failing. The computational complexity of invariant verification is revealed when systems that exhibit high degrees of concurrency with irregular state spaces are analyzed (the well-known state explosion problem). In those cases, even the utilization of BDD-based symbolic techniques [1,2] does not allow the complete analysis of the state space.

In recent years mixed approaches combining simulation and formal verification have been introduced, coining the term *hybrid verification* [3]. Instead of ensuring the complete exploration of the state space, hybrid verification intends to provide efficient mechanisms to identify significantly large portions of

* Funded by the Ministry of Science and Technology of Spain TIC 2001-2476-C03-02 and DURSI of Generalitat de Catalunya 2001SGR-00226.

D. Geist and E. Tronci (Eds.): CHARME 2003, LNCS 2860, pp. 378–393, 2003.
Springer-Verlag Berlin Heidelberg 2003

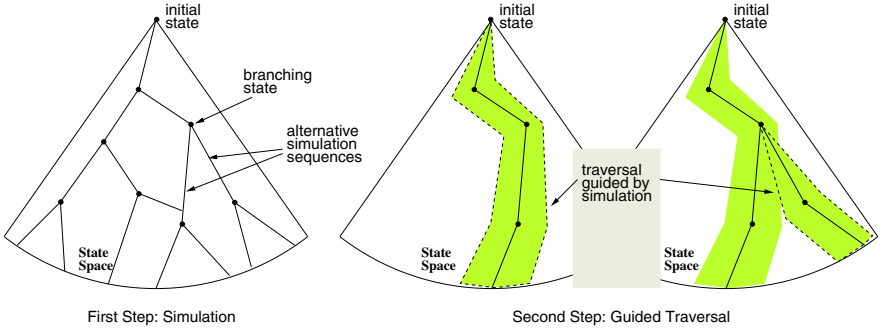

Fig. 1. Two-step scheme: simulation followed by guided-traversal.

the space space with a reduced computational complexity. Hybrid verification has been traditionally useful when the size of the system under analysis is too large to be fully verified by conventional means. In these cases, hybrid verification provides the designer with positive feedback to improve the reliability of the system in terms of failures discovered in a first step of the verification flow. These techniques may also help in the early stages of the design, when failures are not real design errors but holes in the specifications.

This paper presents a hybrid reachability strategy tailored for asynchronous concurrent systems, i.e. to consider the interleaved execution of concurrent events. We propose a two-step mechanism based on a combination of simulation and reachability analysis (see Figure 1). In a first step, simulation provides an initial depth-first view of the states in the system. In order to guarantee a good coverage of the state space, simulation detects those states where the system chooses between alternative execution sequences (*i.e. branching sequences*). Then, each one of the possible alternative sequences will be further explored. The analysis introduced in this work guarantees that interleaving branches due to concurrency will not be exhaustively explored during simulation. Conversely, only one of the sequences is explored, resembling those techniques used in partial order reduction methods [4,5].

In a second step symbolic traversal is applied to improve the state coverage. The information about the ordering in which events are fired, obtained by simulation, is used to guide the way in which traversal is applied. Reachability analysis is performed for each one of the sequences generated by the simulation phase, accumulating the obtained states.

The remainder of the paper is organized as follows. Section 2 introduces existing previous research related to our methodology. Section 3 provides background on the model used for asynchronous concurrent systems, and on the peculiarities of their reachability analysis. The proposed simulation scheme is described in Section 4. The analysis of the dynamic behavior of the system and its application to guided traversal is described in Section 5. Experimental results on the application to invariant checking on control-dominated asynchronous circuits are analyzed in Section 6. Section 7 concludes the paper.

2 Previous Work

State space exploration using guided techniques has become subject of wide interest. These techniques tackle the guidance of reachability analysis toward failure detection rather than to complete state space computation. Guided search typically uses "score-boarding" to find sequences from the initial states to failure states. Various metrics have been proposed to prioritize the state exploration based on the Hamming distance [6], *tracking* [6], *reachability probability* [7], *lighthouses* or *guide-posts* [8,6,9], and *rarity search* [10].

Several techniques have been introduced to guide the search toward uncovered regions of the state space. Ganai et al. [8] introduced a combination of *adaptive simulation* with *retrograde analysis*. Adaptive simulation is based on random simulation with a backtracking mechanism to avoid getting stuck during the search. Retrograde analysis involves a combination of forward analysis with pre-images from the failure states. Bloem et al. [11] use *hints* to guide the symbolic search and to alleviate the BDD explosion problem. Each hint indicates which portion of the transition relation should be used at each step to avoid a BDD blowup. Ganai et al. [8] and Yang et al. [6] suggest the manual insertion of guide-posts. User defined guide-posts are variables inserted in the system, which if activated during the traversal indicate that we are in the right way to find a failure. In [9] an automatic guide-post insertion mechanism is proposed. Kuehlmann et al. [7] suggest using the state reachability probability as a guide for state prioritizing. Again, Ganai et al. [10] propose a rarity-based guide that tracks latch toggle activity to improve state coverage.

Some authors suggest the combination of symbolic reachability analysis with BDD-subsetting. In [12] when the BDD representing the state space grows beyond a certain limit, a subset is taken such that the BDD size is reduced but a large fraction of the state space is kept. [3] attempts to improve the subsetting mechanism by differentiating control and data-path, and keeping subsets that preserve all possible control behaviors.

The work presented in this paper resembles some of the strategies using by partial order reduction techniques [4,5]. However, some key aspects differentiate our approach from these techniques. First of all, the goal of the approach is to generate the largest possible portion of the state space. This goal is radically opposite to partial order reduction. Second, the state successors to be explored are selected taking into account exclusively the causality relations between events in the system. No assumption is made on the type of temporal property being verified. Additionally, the reduced state space is never rebuild, only finite sequences of states are generated.

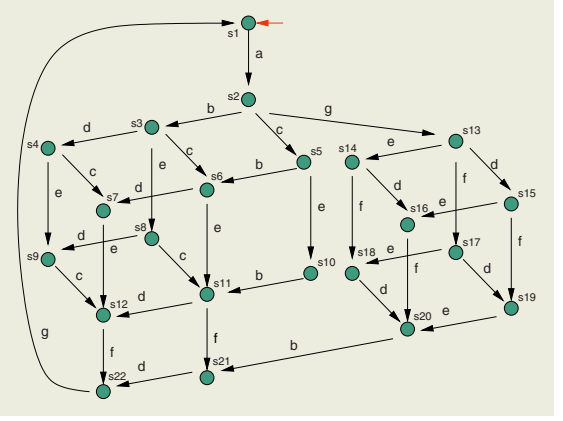

Fig. 2. An example of transition system.

3 Background

3.1 Transition Systems

A TS is a formalism oriented to modeling asynchronous concurrent systems that emphasizes the execution of abstract events rather than how they are encoded. Events may have different semantics depending on the level of detail of the model (signal changes, protocol operations, etc). The concurrent execution of events is described by means of *interleaving*, *i.e.* weaving the execution into sequences.

Formally, a *transition system* (TS) [13] is composed of a non-empty set of states $\mathcal{S}$, a non-empty alphabet of events Σ, a transition relation $T \subseteq \mathcal{S} \times \Sigma \times \mathcal{S}$, and a set of initial states S_{in}. Transitions are denoted by $\mathsf{s} \xrightarrow{\mathsf{e}} \mathsf{s}'$. The *firing region* of an event e is defined as $\mathsf{Fr} : \Sigma \to 2^S$ such that $\mathsf{Fr}(\mathsf{e}) = \{\mathsf{s} \in S \mid \exists\, \mathsf{s} \xrightarrow{\mathsf{e}} \mathsf{s}' \in T\}$. Thus, event e is *firable* at state s if $\exists \mathsf{s} \xrightarrow{\mathsf{e}} \mathsf{s}' \in T$, *i.e.* $\mathsf{s} \in \mathsf{Fr}(\mathsf{e})$. The set of events *firable* at state s is denoted by $\mathcal{E}(\mathsf{s})$. A *run* of a TS is a *firing sequence* $\sigma = \mathsf{s}_1 \xrightarrow{\mathsf{e}_1} \mathsf{s}_2 \xrightarrow{\mathsf{e}_2} \cdots$, such that $\mathsf{s}_1 \in \mathsf{S}_{in}$ and $\forall i \geq 1 : \mathsf{s}_i \xrightarrow{\mathsf{e}_i} \mathsf{s}_{i+1} \in T$. Given the significance of individual events, the transition relation (TR) of a TS can be naturally partitioned into a disjoint set of relations, one for each event $\mathsf{e} \in \Sigma$: $T_{\mathsf{e}} = \{\mathsf{s} \xrightarrow{\mathsf{e}} \mathsf{s}' \in T \mid \exists \mathsf{s}, \mathsf{s}' \in \mathcal{S}\}$.

Figure 2 shows a TS that will be used as a running example. The system contains 22 states and a set of events $\Sigma = \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}, \mathsf{e}, \mathsf{f}, \mathsf{g}\}$. State s_1 is its initial state. Note the existence of multiple interleaving sequences due to concurrency, e.g. $\mathsf{a} \longrightarrow \mathsf{b} \longrightarrow \mathsf{c}$ and $\mathsf{a} \longrightarrow \mathsf{c} \longrightarrow \mathsf{b}$.

3.2 Reachability Analysis

The set of states that is reachable in any number of steps from a set of states C ($Reach(T, C)$) is defined as the least fix-point of the following recurrence:

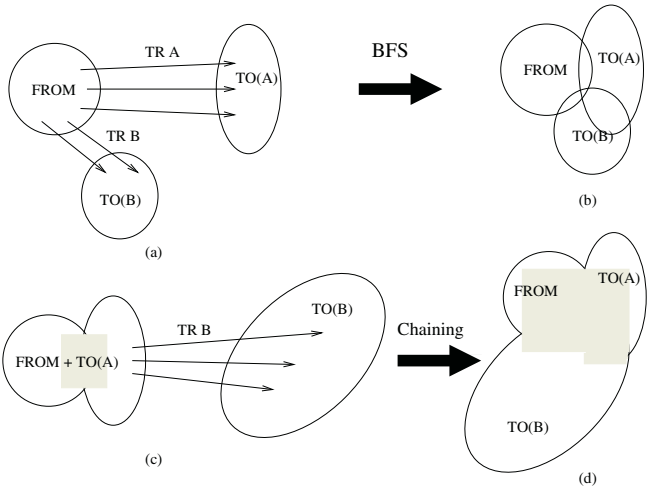

Fig. 3. General example for the chaining technique.

$$\begin{aligned} \mathsf{S}_o &= C \\ \mathsf{S}_{i+1} &= \mathsf{S}_i \cup \mathit{Img}(T, \mathsf{S}_i) \ . \end{aligned} \tag{1}$$

where $\mathit{Img}(T, \mathsf{S}_i)$ is the one-step image computation applying the TR T on a set of states S_i. When, C equals S_{in} for a given TS, this algorithm generates the state space of a system in a Breath First Search (BFS) style. The number of iterations performed by such traversal is determined by the maximum number of steps from the initial state to the first occurrence of each of the reachable states (called the *sequential depth* of the TS). In the example of Figure 2, the application of BFS from the initial state gives state s_2 in a first step, states $\mathsf{s}_3, \mathsf{s}_5, \mathsf{s}_{13}$ in a second step, states $\mathsf{s}_4, \mathsf{s}_8, \mathsf{s}_6, \mathsf{s}_{10}, \mathsf{s}_{14}, \mathsf{s}_{17}, \mathsf{s}_{15}$ in a third, etc.

The classical BFS algorithm can be improved based on two key observations. First, at each iteration of the BFS traversal, most transitions described in the monolithic TR are not applied (*e.g.* at the second BFS step, only events $\mathsf{b}, \mathsf{c}, \mathsf{g}$ are significant). And second, the TR of a TS can be naturally partitioned into disjunctive TRs, one for each event, that can be applied individually.

These observations have suggested alternative traversal algorithms, named *chaining* [14,15]. Chaining applies the individual TRs of events in a predetermined order such that the number of new states generated at each step is maximized. After the application of the transition relation of an event, the newly generated states are immediately used as domain for the next event in order, hence coining the term chaining.

Figure 3 shows the general concept for two TRs A and B. If A and B are applied to the same set FROM in a BFS style, a certain number of states is reached (see Figure 3(a) and (b)). However, chaining would apply A to FROM and generate a new set of states ($\mathsf{FROM} + \mathsf{TO(A)}$ in Figure 3(c)), and afterward

apply B to this set (in Figure 3(d)). The number of reached states increases with almost the same computational effort.

In practice, chaining can significantly reduce the number of iterations of the BFS algorithm [15,16]. The method is specially effective if the appropriate firing order of the events is selected. Chaining outperforms BFS techniques in the verification of asynchronous concurrent systems because states are computed at a much faster ratio and with less effort, thus reducing the number TR applications. Moreover, partitioning the TR provides important memory savings and CPU speed-ups when implemented over BDD structures.

4 Simulating Transition Systems

This section presents a simulation approach for asynchronous concurrent systems that automatically provides a good state space coverage. At each explored state the causality between firable events is analyzed to identify firing conflicts between them. Conflict detection allows to identify execution sequences that exclude each other in a variety of ways, including mutual exclusion for example. This simulation scheme resembles those state exploration techniques used by partial order reduction [4,5].

Simulation chooses a particular firing order among all possible interleaved executions of concurrent events. Thus, simulation alone has limited coverage effectiveness for concurrent systems. We will show in Section 5 that the interleaving of events due to concurrency can be explored more efficiently by symbolic traversal once the information from a particular simulation sequence is available.

4.1 Conflict Detection to Improve Coverage

Conflict detection is the key mechanism that allows to distinguish between sequences of events representing alternative behaviors of a system or interleaving sequences of concurrent events. The first type of sequences are relevant and must be explored in order to guarantee a good coverage of all possible behaviors of the system. Exploring interleaved sequences must be avoided and postponed to the symbolic traversal phase.

An event e_1 *disables* another event e_2 if a pair of states exists $\mathsf{s}_1, \mathsf{s}_2$ such that $\mathsf{s}_1 \xrightarrow{\mathsf{e}_1} \mathsf{s}_2 \in T$ and e_2 is firable in s_1 ($\mathsf{e}_2 \in \mathcal{E}(\mathsf{s}_1)$) but e_2 is not firable in s_2 ($\mathsf{e}_2 \notin \mathcal{E}(\mathsf{s}_2)$). Two events $\mathsf{e}_1, \mathsf{e}_2$ are in *conflict* if e_1 disables e_2 or e_2 disables e_1. A conflict is called *symmetric* if e_1 disables e_2 and e_2 disables e_1. The conflict is called *asymmetric* if e_1 disables e_2 but e_2 does not disable e_1, or vice versa.

Figure 4 depicts a portion of the state space of a concurrent system. The figure illustrates the conflict situations previously described. From the initial state (a) shows three events e_1, e_2, e_3 that are mutually concurrent; (b) shows a symmetric conflict between e_1 and e_2; and (c) shows an asymmetric conflict in which e_2 disables e_1 but not the contrary (event e_3 remains concurrent to e_1 and e_2).

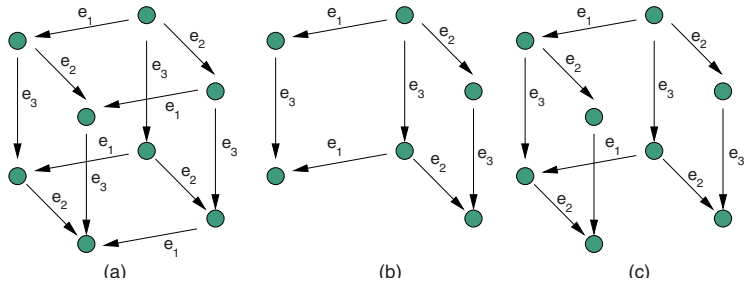

Fig. 4. Concurrent and conflict situations.

A state in which two or more events are in conflict is called a *branching state* in which alternative execution sequences exist (see Figure 1). Each separate sequence can be followed, resulting into different behaviors of the system.

Symmetric conflicts are associated to states in which the system takes a decision. The behavior in each branch may involve completely different sets of events and thus produce distinct/disjoint sets of states. A different simulation sequence is generated for each branch in order to achieve a better coverage of the state space. On the contrary, asymmetric conflicts can be associated to *disablings*, in which the firing of one event (the *disabler*) prevents the firing of a second event (the *disabled*). In this type of conflict two different firing sequences exists. In one of the sequences, both events can fire concurrently and no disabling occurs. In the other sequence, the disabler event fires thus disabling the second event and, in consequence, disabling also some part of the system behavior. As an example, disablings can be associated to races in digital circuits (*e.g.* producing either glitches or dead-locks at the output of some gates).

4.2 Simulation Algorithm

This section presents an improved simulation mechanism based on the analysis of the conflicts found along the simulation sequences. Every time a pair of conflicting events is identified, a new sequence is generated and stored in a list of *pending* sequences. The sequence duplication scheme is detailed in Figure 5. In the example there is a firing sequence (σ_1) in which three events $\mathsf{e}_1, \mathsf{e}_2, \mathsf{e}_3$ are firable in state s_1. Let us assume that events e_1 and e_2 are in symmetric conflict. A copy of the branch state s_1 is generated (s'_1), together with a copy (σ'_1) of the sequence (up to state s_1) being explored. The exploration continues by removing the disabled event (e_2) from the list of firable events at s_1. Then, event e_1 is fired in the *active* sequence σ_1 generating a state where only e_3 remains firable. On the other hand, σ'_1 is *stored* for later exploration. The disabled event (e_1) is removed from the list of firable events at state s'_1. Note that the order in which events have been selected it is not necessarily the order in which our algorithm may proceeed. In that case, concurrent events like e_3 may be given priority.

Simulation sequences are processed following a configurable priority scheme. States can be analyzed following a DFS or BFS style, or a mixture of both.

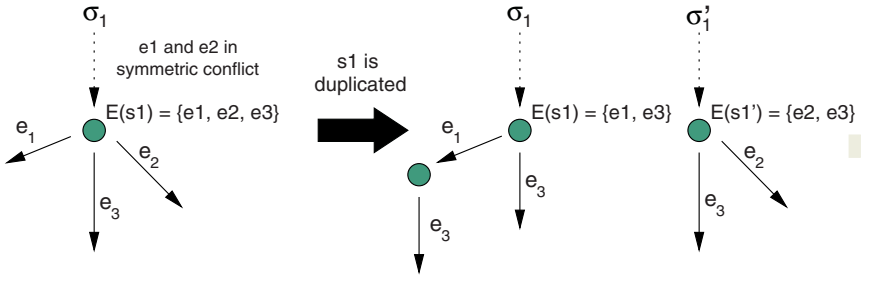

Fig. 5. Branching sequences due to conflicts.

However, other parameters can be taken into account, *e.g.* the number of choices already taken. The firing order of the events can be also decided according to some priority scheme. In our simulator, we keep track of the number of times that each event is fired. To avoid locking the state exploration in some local region of the state space, we give additional priority to those events which have been fired less often.

The algorithm in Figure 6 describes the suggested simulation scheme. The simulation engine stores the set of sequences, together with the events that are ready to fire, in the `active` list. Sequences are stored as linked lists of BDD cubes, each cube representing a state of the system. Terminated sequences due to state repetition, deadlocks or simulation limits are stored in `seq`. All states visited along the simulation are stored in `visit`. Each sequence analyzed along the simulation is stored as a tuple $\tau = (\mathtt{s}, \sigma, E, D, B)$ that consists of: $\mathtt{s}$ the last state in the sequence; σ the firing sequence required to reach $\mathtt{s}$ from $\mathtt{S}_{in}$; a set $E \subset \Sigma$ that indicates the events that remain firable at s; and two integers to indicate the firing depth D with respect to $\mathtt{S}_{in}$, and the number of taken choices B required to reach that depth.

Without loss of generality we will assume that the simulation starts from a single initial state $\mathtt{s}_{in}$. A tuple is created for this state by using the empty sequence $\{\mathtt{s}_{in}\}$ and all possible firable events (retrieved by function $\mathtt{firable}(\mathtt{s}_{in})$). This initial sequence is placed into the list of active sequences pending of being processed.

The simulator takes one sequence from the list of pending sequences. The last state of the sequence is checked to determine if the simulation should proceed from it. If the state has been already visited, or simply the depth/branch limit have been surpassed, the sequence is stored in `seq`. If the last state can be processed, a firable event $\mathtt{e} \in \tau.E$ is selected. Events can be selected giving priority to either: events that are not in conflict, events that are in symmetrical conflict and events that are in asymmetric conflict. If event $\mathtt{e}$ is in conflict ** , then sequence τ is duplicated into an exact copy τ'. Event $\mathtt{e}$ is marked as non-firable in τ' to avoid exploring the same sequence multiple times (see Figure 5). Finally, τ' is inserted back into the list of active sequences for a later exploration of alternative branches.

```
1.  visit := seq := active := ∅;
2.  τ := alloc_sequence (s_in, {s_in}, E(s_in), 0, 0);
3.  active := active ∪ τ
4.  while (active ≠ ∅) do
5.      τ := get_priorized_sequence (active)
6.      visit := visit ∪ τ.s
7.      if (termination_condition (τ)) then
8.          seq := seq ∪ τ.σ
9.          free_sequence(τ)
10.         continue;
11.     e := select_firable_event(τ.E);
12.     if (event_disables(τ, e)) then
13.         τ' := duplicate_sequence (τ)
14.         τ'.E := τ'.E \ e
15.         τ'.B = τ'.B + 1
16.         active := active ∪ τ'
17.     τ.s := Img(T_e, τ.s)
18.     τ.D := τ.D + 1
19.     τ.E := firable(τ.s)
20.     τ.σ := τ.σ --e--> s
21.     active := active ∪ τ
```

Fig. 6. Pseudocode of the simulation algorithm.

The selected event e is fired from state τ.s, generating its successor $Img(T_{\mathsf{e}}, \tau.\mathsf{s})$ that is updated in the sequence. The remaining information is also updated, including the extension of the firing sequence by $\sigma \xrightarrow{\mathsf{e}} \mathsf{s}$. Finally, τ is updated and placed back into the list of active sequences.

Given the example of Figure 2, the simulator generates two firing sequences (assuming an alphabetical firing order of the events) shown in Figures 7(a) and 8(a), respectively. Two sequences are generated because a conflict is detected at state s_2 between events b and g, when b is selected to fire. A third sequence is also generated, although not shown due to lack of space, because events c and g are also in conflict at s_2. Note that sequences are annotated with the events firable at each state.

This simulation scheme allows a fast in-depth analysis of the system, providing a good state coverage since all conflict branches are identified. A number of heuristic termination conditions are included to avoid repeating equivalent execution sequences: stop exploring a sequence whenever an already visited state is reached, and bound the depth of the sequences to a factor of the total number of events in the TS.

5 Guided Traversal

This section shows how the sequences generated by the simulation phase, together with the information about which events are firable at each state, allows analyzing the causality relations between events. Such causality is later exploited to improve the symbolic traversal by using chaining. An efficient traversal algorithm is applied for selected sequences, thus improving the state coverage. Following the ideas in [16], the TR of the system is partitioned, and the application of each part is scheduled by analyzing the causality relations between the events in the sequence, thus maximizing the state generation ratio.

5.1 Extracting Causality Relations

Causal event structures (CES) describe all possible sequential and concurrent executions of a set of events. A CES [17] is a tuple $\langle \Sigma, \prec \rangle$ where $\Sigma = \{\mathsf{e}_1, \ldots, \mathsf{e}_n\}$ is a finite set of *events* and $\prec \subseteq \Sigma \times \Sigma$ is a strict partial order (irreflexive and transitive) over Σ called the *causality relation*.

Given a CES the following relations can be defined where **co** is called the *concurrency relation*:

$$\begin{aligned} \mathbf{id} &\stackrel{def}{=} \{(\mathsf{e},\mathsf{e}) \mid \mathsf{e} \in \Sigma\} \\ \succ &\stackrel{def}{=} \{(\mathsf{e}_1,\mathsf{e}_2) \mid (\mathsf{e}_2,\mathsf{e}_1) \in \prec\} \\ \mathbf{co} &\stackrel{def}{=} \Sigma \times \Sigma - \{\prec \cup \succ \cup \mathbf{id}\} \end{aligned}$$

Provided a firing sequence of events, we can recover a partial order showing the causality relationships among those events, *i.e.* a CES. A partial order $\prec$ over a set of events Σ and the associated relations $\succ$, **id** and **co** completely partition $\Sigma \times \Sigma$. We can use this fact to derive a CES from a sequence σ, such that: **id** is obviously defined; e_1 **co** e_2 if e_1 and e_2 are firable simultaneously in some state visited by σ; and $\mathsf{e}_1 \prec \mathsf{e}_2$ if e_1 precedes e_2 and are not firable simultaneously in σ (similarly for $\succ$).

Figures 7(b) and 8(b) show the CESs derived from the sequences in Figures 7(a) and 8(a), respectively. Arcs denote causality relations between events. For sake of clarity, we also indicate with dotted arcs the conflict relations, although they are not part of the actual CES. Thus, if event b disables g, a dotted arc from b to g is drawn. In Fig. 7, note that event g appears two times. The first time is disabled by b, while the second one is fired after g.

5.2 Reachability Analysis

A *topological order* of the events of a CES is a sequence $\mathsf{e}_1 \cdots \mathsf{e}_n \in \Sigma^*$ $(n = |\Sigma|)$, such that all e_i are distinct and $\forall\, 1 \le i,j \le n : \mathsf{e}_i \prec \mathsf{e}_j \Rightarrow i < j$. Firing the events following such topological order often guarantees that when an event is fired all its causal predecessors have been already fired. Given an event e_i ready to fire, if all the events concurrent to e_i are fired before e_i, most states in $\mathsf{Fr}(\mathsf{e}_i)$ will

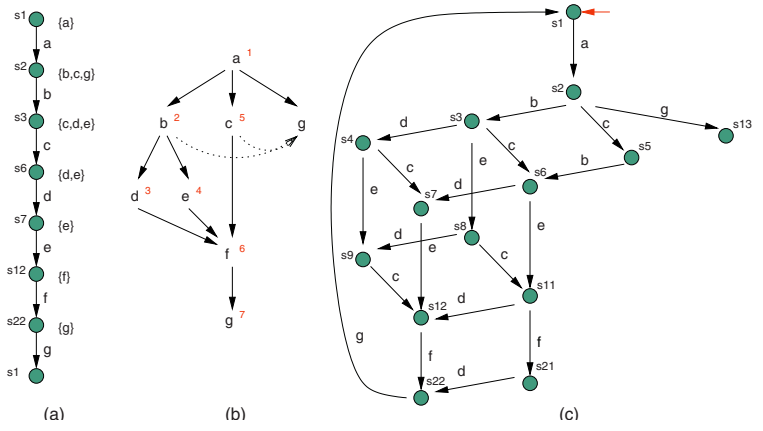

Fig. 7. Sequence 1: (a) Firing sequence, (b) CES capturing causality, and (c) states reached after guided traversal.

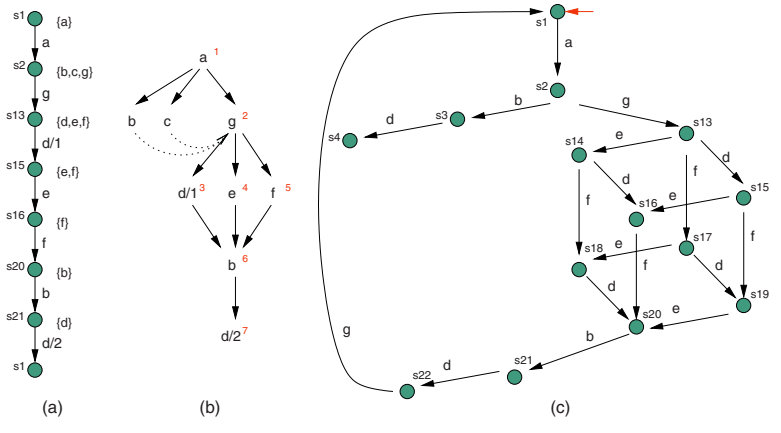

Fig. 8. Sequence 2: (a) Firing sequence, (b) CES capturing causality, and (c) states reached after guided traversal.

be already reached. A traversal algorithm in which events are fired following the topological order guarantees a good effectiveness. Unfortunately, the causality relations of a complex system cannot be described with a single CES. A pair of events may be causally ordered in some part of the state space, whereas they may be concurrent in another. On the contrary, causality relations derived from a single firing sequence provides a quite precise approximation of the behavior at localized areas of the state space. Hence, traversal can successfully exploit that information in those cases.

Given a set of sequences generated from the described simulation process, we propose the following three-step traversal strategy (see Figure 9): Generate the CES for each firing sequence (2). Find a topological order of the events in the

```
1. S_σ := S_in;
2. CS := build_CES(σ);
3. T_order := find_topological_order(CS);
4. foreach e_i ∈ T_order do
5.       S_σ := S_σ ∪ Img(T_{e_i}, S_σ);
```

Fig. 9. Guided traversal algorithm.

CES(3). Execute a symbolic traversal algorithm from the initial state by applying the TR of each event in sequence (4–5). Events will be applied following the topological order extracted from the CES. The states generated after the image computation of one event will be immediately applied as domain for the image computation of the successor event in order, thus *chaining* the effect.

Note that, in practice, not all firing sequences need to be considered for traversal. Some sequences will be almost equivalent to other, with only a small suffix of the simulation being different. In those cases, causality analysis and traversal should be only applied to the suffix. Otherwise large amounts of states will be repeatedly generated from different sequences.

Figures 7(b) and 8(b) show the CES annotated with an index that indicates the position in the topological order selected for traversal. Observe that the disabled events are not annotated since they do not actually belong to the CES. Figures 7(c) and 8(c) show the portions of the state space in the original TS (see Figure 2) generated by the guided traversal for each firing sequence. After both traversals only state s_{10} remains unreached. Note that in Figure 8, event d appears two times along the sequence. In that case, duplicated events are renamed to satisfy the topological order requirements.

5.3 Methodology Implementation

Figure 10 sketches an implementation of the proposed strategy for hybrid exploration of the state space. The process is divided in two parts: a simulation phase followed by a traversal phase.

Simulation phase: From the initial state, the simulation engine generates multiple branching sequences. The number of sequences will be determined by the set of conflicts found during the state exploration, or limited by a user-defined limiting parameter. Causality is extracted and attached to each sequence to be used later in the traversal phase.

Traversal phase: Sequences are iteratively taken to apply symbolic traversal on them. Heuristically, we choose the sequence that contains more states not covered by previous sequences. Note that if all states in a sequence are already contained in the set of reached states so far, the sequence will be discarded for traversal. The events in the selected sequence are fired following a topological order. The order is either extracted from the causality information attached to the sequence, or directly taken from the order in which events are fired along

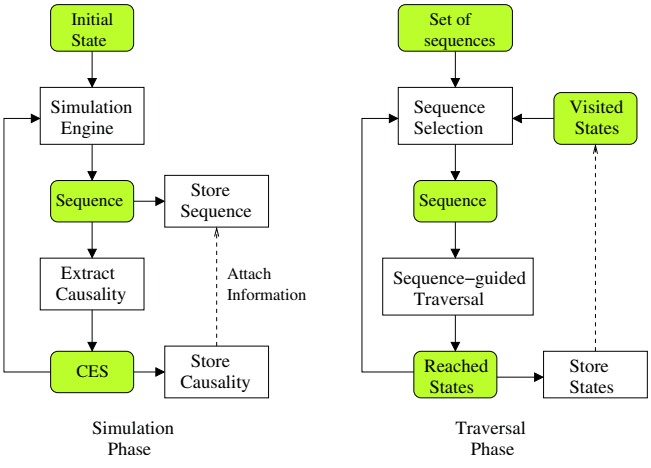

Fig. 10. Structure of the hybrid traversal methodology.

the sequence (also a valid topological order). Events are iterated once if applied from the causality information, or until a fix-point is reached if applied from the simulation order.

6 Experimental Results

In the following tables several asynchronous concurrent systems are analyzed using the hybrid reachability scheme described in this paper. A brief description of each systems follows:

PCC Pausible clock controller for heterogeneous systems in [18].
GALS Globally-Asynchronous Locally-Synchronous design in [19].
RGD-arbiter asP*, RGD arbiter in [20] described at transistor level.
IPCMOS A pulse-based controller for asynchronous pipelines in [21].
STARI A self-timed pipeline in [22].

All these results are from executions on a 2Ghz Pentium IV Linux computer with 512Mb of memory. Note that the behavior of all these systems is delay-dependent. In our experiments we only concentrate on the untimed state space.

Table 1 compares the results of full reachability analysis when using different traversal strategies on the selected benchmarks. Suffix C is used for circuits and A for abstractions. The number in parenthesis indicates the number of stages in case of pipelines. We provide results for BFS traversal (BFS), chained traversal using a greedy ordering strategy (C Greedy) [14], and a token-traverse chained strategy (C Token) [16]. Our goal when presenting these experiments is to demonstrate the significant impact that the chaining methodology has on the efficiency of traversal. Moreover, we will use these results as a reference to evaluate the proposed hybrid methodology.

Table 1. Experimental results: various forms of traversal.

		BFS			C Greedy			C Token		
Name	States	Iter	BDD	CPU	Iter	BDD	CPU	Iter	BDD	CPU
GALS-C	1.232e+3	68	10498	42.7	17	10914	0.2	10	10767	0.2
PCC-C	9.89184e+5	64	80979	42.4	15	18104	2.6	5	12573	2.7
RGD-arbiter-A	3.33813e+9	79	218088	695.7	20	113757	22.6	5	13938	6.1
RGD-arbiter-C	5.46918e+13			Tout	27	823820	1469.5	16	44238	46.0
IPCMOS-C (4 c)	8.15635e+9			Tout	30	80479	51.6	10	121994	44.1
IPCMOS-C (6 c)	1.78657e+14			Tout	41	126707	41.3	13	207124	19.1
IPCMOS-A (4 c)	1.16785e+7	129	201380	96.9	12	153160	28.0	11	223037	48.4
IPCMOS-A (6 c)	9.15592e+9	237	209198	1055.1	16	54978	22.1	8	188061	27.3
STARI-C (8 c)	1.07225e+12			Tout	56	170544	105.5	11	219575	73.0

Table 2. Experimental results: simulation followed by guided-traversal.

	Simulation				Traversal			
Name	Seq	BDD	States	CPU	Seq	BDD	States	CPU
GALS-C	27	13485	381	0.5	1	16208	1.232e+3	0.8
PCC-C	1	9120	306	0.5	1	21185	9.89184e+5	3.7
RGD-arbiter-A	17	10493	142	0.5	1	33355	1.05433e+9	2.7
RGD-arbiter-C	30	17480	221	1.2	1	148711	9.18829e+12	17.4
IPCMOS-C (4 c)	1	8088	179	0.3	1	99799	8.05928e+9	21.6
IPCMOS-C (6 c)	1	15191	263	0.6	1	278575	1.75992e+14	14.9
IPCMOS-A (4 c)	1	13727	133	0.3	1	151493	1.16785e+7	25.6
IPCMOS-A (6 c)	1	28481	241	0.9	1	179577	9.15592e+9	32.9
STARI-C (8 c)	8	141299	5646	16.9	2	283725	9.73548e+11	126.0

The first column in Table 1 shows the total number of states (States). The second set of columns shows the number of iterations (Iter) of BFS traversal, the peak BDD size (BDD) and the computation time (CPU) (in seconds). The third set of columns shows the same parameters but for the chained traversal with greedy ordering. The last set of columns shows the same parameters but for the token traverse chained strategy. Note that in both modified traversals Iter refers to the iterations of the algorithm, not to the sequential depth of the experiment.

Table 2 shows the results of the hybrid traversal strategy. The first set of columns provides data to evaluate the simulation phase. Column Seq indicates the total number of sequences that have been explored; column BDD shows the peak BDD size; column States shows the total number of states visited along the simulation; finally CPU indicates the computation time in seconds. It is important to note that the ratio of visited states versus CPU time is small compared to standard simulations. The reason is that, at each state, conflict relations should be analyzed penalizing the simulation efficiency. However, this initial effort should pay off later in the traversal phase.

The second set of columns provides data to evaluate the traversal phase. Column Seq indicates the subset of sequences that have been traversed; column BDD shows the BDD peak size during traversal, Column States shows the states reached after guided-traversal; and column CPU indicates the computation time in seconds for this phase. Note that some BFS results not shown are due to a CPU time-out set to 1 hour.

The initial set of experiments, BFS traversal versus chained traversal highlights the importance of a good chained strategy. The impact in both number of

iterations and peak BDD size allows reducing the computation times for traversal.

These preliminary experiments show that significant portions of the state space can be reached by our hybrid approach in reduced CPU times. BDD sizes remain reasonable for all examples, as expected, due to the spatial locality obtained by the guided-traversal step. In addition, the portion of the state space generated by the approach is a good starting point to execute symbolic traversal until the full state space is reached.

In the future we intend to improve the strategies used to select which events must be fired first during simulation. These strategies should influence to a great extent the coverage of the state space achieved during simulation, and later on during guided-traversal. We also want to explore in more detail how the firing order for events influences in the BDD sizes and the state coverage.

7 Conclusions

We believe that the incremental analysis of the state space of a system by techniques that exploit state locality is the key for the success of traversal algorithms. Instead, most existing approaches try to exploit the locality available in the transition relations to minimize them, rather than considering the impact in the representation of the state space. Following this line of reasoning, we have proposed a two-step hybrid reachability analysis strategy that combines fast simulation and guided-traversal. Simulation provides information to identify subsets of the state space in which the causality between events can be properly identified. This information can be exploited in a second phase. Causality provides enough information to efficiently generate large portions of the state space. Additionally, information about good chaining order is also extracted, which is used to guide the later traversal. The combination of both strategies should allow the reduction of BDD sizes as well as the execution times.

References

1. R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, 1986.
2. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, "Symbolic model checking: 10^{20} states and beyond," *Information and Computation*, vol. 98, no. 2, pp. 142–170, 1992.
3. J. Yuan, J. Shen, J. Abraham, and A. Aziz, "On combining formal and informal verification," in *Proc. International Conference on Computer Aided Verification*, vol. 1254 of *LNCS*, pp. 376–387, Springer-Verlag, 1997.
4. A. Valmari, "Stubborn sets for reduced state space generation," in *Proc. of International Conference on Application and Theory of Petri Nets*, vol. 483 of *LNCS*, pp. 491–515, Springer-Verlag, 1989.
5. P. Godefroid, "Using partial orders to improve automatic verification methods," in *Proc. Workshop on Computer Aided Verification*, vol. 531 of *LNCS*, pp. 176–185, Springer-Verlag, 1990.

6. C. H. Yang and D. Dill, "Validation with guided search of the state space," in *Proc. Design Automation Conference*, pp. 599–604, 1998.
7. A. Kuehlmann, K. McMillan, and R. Brayton, "Probabilistic state space search," in *Proc. International Conference on Computer Aided Design*, pp. 574–579, 1999.
8. M. Ganai and A. Aziz, "Efficient coverage directed state space search," in *Proc. International Workshop on Logic Synthesis*, 1998.
9. P. Yalagandula, V. Singhal, and A. Aziz, "Automatic lighthouse generation for directed state space search," in *Proc. Design, Automation and Test in Europe*, pp. 237–242, 2000.
10. M. Ganai and A. Aziz, "Rarity based guided state space search," in *Proc. Great Lakes Symposium on VLSI*, pp. 97–102, 2001.
11. R. Bloem, K. Ravi, and F. Somenzi, "Symbolic guided search for CTL model checking," in *Proc. Design Automation Conference*, pp. 29–34, 2000.
12. K. Ravi and F. Somenzi, "High-density reachability analysis," in *Proc. International Conference on Computer Aided Design*, pp. 154–158, 1995.
13. A. Arnold, *Finite Transition Systems*. Prentice-Hall, 1994.
14. O. Roig, J. Cortadella, and E. Pastor, "Verification of asynchronous circuits by BDD-based model checking of petri nets," in *International Conference on Application and Theory of Petri Nets*, vol. 935 of *LNCS*, pp. 374–391, Springer-Verlag, 1995.
15. E. Pastor, J. Cortadella, and O. Roig, "Symbolic analysis of bounded petri nets," *IEEE Transactions on Computers*, vol. 50, no. 5, pp. 432–448, 2001.
16. M. Sole and E. Pastor, "Traversal techniques for concurrent systems," in *Proc. International Conference on Formal Methods in Computer-Aided Design*, vol. 2017 of *LNCS*, pp. 220–237, Springer-Verlag, 2002.
17. M. Nielsen, G. Plotkin, and G. Winskel, "Petri Nets, Event Structures and Domains," *Theoretical Computer Science*, vol. 13, pp. 85–108, 1981.
18. K. Yun and A. Dooply, "Pausible clocking based heterogeneous systems," *IEEE Transactions on VLSI Systems*, vol. 7, no. 4, pp. 482–487, 1999.
19. J. Muttersbach, T. Villigers, and W. Fichtner, "Practical design of globally-asynchronous locally-synchronous systems," in *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 52–59, 2000.
20. M. R. Greenstreet and T. Ono-Tesfaye, "A fast, ASP*, RGD arbiter," in *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 173–185, 1999.
21. S. Schuster, W. Reohr, P. Cook, D. Heidel, M. Immediato, and K. Jenkins, "Asynchronous Interlocked Pipelined CMOS Circuits Operating at 3.3 – 4.5GHz," in *IEEE International Solid-State Circuits Conference*, pp. 292–293, 2000.
22. M. R. Greenstreet, *STARI: A Technique for High-Bandwidth Communication*. PhD thesis, Princeton University, 1993.