

# Traversal techniques for concurrent systems <sup>\*</sup>

Marc Solé and Enric Pastor

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

**Abstract.** Symbolic model checking based on Binary Decision Diagrams (BDDs) is a verification tool that has received an increasing attention by the research community. Conventional breadth-first approach to state generation results is often responsible for inefficiencies due to the growth of the BDD sizes. This is specially true for concurrent systems in which existing research (mostly oriented to synchronous designs) is ineffective. In this paper we show that it is possible to improve BFS symbolic traverse for concurrent systems by scheduling the application of the transition relation. The scheduling scheme is devised analyzing the causality relations between the events that occur in the system. We apply the scheduled symbolic traverse to *invariant checking*. We present a number of schedule schemes and analyze its implementation and effectiveness in a prototype verification tool.

## 1 Introduction

A lot of effort has been made by the verification community to develop efficient traversal methods [1, 2]. Unfortunately most of them are designed to improve the traversal process of synchronous systems and are not suitable or relevant for concurrent systems (concurrent systems may include asynchronous circuits [3, 4], distributed systems [5, 6], etc.). In synchronous systems, transition relations (TRs) are usually partitioned and the sequence of application of each part must be decided in order to reduce the BDD sizes for intermediate results. The application order in this case is important because the way the variables are quantified depends on it, affecting the size of the intermediate representation. This is usually referred as the quantification schedule problem.

Algorithms developed to solve the quantification schedule problem have no practical application for concurrent systems. In this latter case we usually have a disjunctive collection of small TRs, each one describing the behavior of some component. Each individual TR is applied assuming interleaved semantics and the result is immediately added to the reachable set of states, so the order in which these TR are fired has a strong influence on the overall performance.

---

<sup>\*</sup> This work has been partially funded by the Ministry of Science and Technology of Spain under contract TIC 2001-2476-C03-02 and grant AP2001-2819.

Some authors have studied the influence of ordering the application of the TR to avoid the BDD explosion problem. Their goal is to schedule the exploration of the state space by taking only selected portions of the TR, or by delaying the exploration of certain states. In [7] Ravi and Somenzi proposed a “high density” traverse, which does not use the set of newly reached states as the from set for the next iteration. Instead it uses a subset of the newly reached states that has a more compact representation. This is a partial traverse, so afterwards must be completed. In [8] Cabodi *et al.* use “activity profiles” for each BDD node in the TRs and prune the BDDs to perform a partial traversal, completed again, in the end. The “activity profiles” are obtained in a preliminary reachability learning phase. In [9] Hett *et al.* propose a sequence of partial traverses that combine subsets of the newly reached states and dynamic TR pruning. Both manipulations are applied using the Hamming distance as the main part of the heuristic function. In [10] Ravi and Cabodi allow the user to provide hints to guide symbolic search. User-defined hints are used to simplify the TR, but require the user to understand the design and also predict the BDD behavior.

Our objective is to minimize the CPU time of the traversal process. Usually the problems appear in its intermediate steps, as big BDDs start to be generated. In this cases the faster you can discover the remaining states, the better the performance is, due to BDD recombination. The speed of new states generation is highly related to the number of TRs applications needed to end up the process. Hence an algorithm for determining a good TR application order is crucial.

This paper proposes a method that intends to complete symbolic traversal with the minimum number of TR applications. The number of intermediate steps is reduced, thus reducing the probability of generating an intermediate BDD that is much too big to cope with. We present four symbolic traverse algorithms that schedule the application of the TRs. The TR application schemas are named: token traverse (TOK), weighted token traverse (WTOK), dynamic event-clustered traverse (DEC), and TR cluster-closure traverse (TRCC).

TOK and WTOK require an static analysis of the system to build the TR application schema. The analysis is basically an *a priori* causality analysis between TRs (see Section 3). Once we have derived a TR application schema we use it to decide the order in which the TRs would be applied. The schema does not imply a *static* TR application order because it uses feedback from the traversal to adapt the order dynamically. TOK and WTOK differ in the kind of feedback that receive from the traversal analysis.

DEC tries to be more accurate in its TR application schema, so it is completely adaptable and has no initial precomputation phase. DEC keeps constantly updated information on how many states each TR may be applied for the first time. Hence we can decide each time which TR has the biggest probabilities to generate new states at a fastest ratio.

Finally, TRCC is an adaptation of partial iterative squaring to the scope of concurrent systems. We combine some TRs to (1) reduce the number of TRs while keeping their size small, thus reducing the number of intermediate results,

and (2) due to squaring reduce the number of iterations needed by the schema to complete the analysis.

The paper is organized as follows. Section 2 is devoted to basic models for formal verification of concurrent systems. Section 3 reviews some of the known peculiarities of symbolic traverse for concurrent systems and their impact on performance. Sections 4, 5, 6 and 7 are the core of this paper as they explain the four traversal proposals: TOK, WTOK, DEC, and TRCC respectively. Section 8 presents some preliminary results on the performance of the different methods on some benchmarks. Finally Section 9 concludes the paper.

## 2 Background

A finite transition system (TS) [11] is a 4-tuple  $\mathcal{A} = \langle S, \Sigma, T, S_i \rangle$ , where  $S$  is a finite set of *states*,  $\Sigma$  is a non-empty alphabet of events,  $T$  is a transition relation such that  $T \subseteq S \times \Sigma \times S$ , and  $S_i \subseteq S$ . Transitions are denoted by  $s \xrightarrow{e} s'$ . An event  $e$  is enabled at state  $s$  if  $\exists s \xrightarrow{e} s' \in T$ . Given an event  $e$  its firing region  $\text{Fr}(e)$  is defined as  $\text{Fr} : \Sigma \rightarrow 2^S$  such that  $\text{Fr}(e) = \{s \in S \mid \exists s \xrightarrow{e} s' \in T\}$ . Event  $e$  is said to be *fireable* at state  $s$  if  $s \in \text{Fr}(e)$ .

The concurrent execution of events is described by means of *interleaving*; that is, weaving the execution of events into sequences. Given the significance of individual events, the transition relation of a TS can be naturally partitioned into a disjoint set of relations, one for each event  $e_i \in \Sigma$ :  $T_e = \{s \xrightarrow{e} s' \in T\}$ .

To represent events symbolically we use a set of Boolean variables that encode the states in the TS and a Boolean relation to encode the TR. The application of a TR  $T_e$  on some set of states  $R$  results in a set of states  $R'$  that contains all the states reachable from  $R$  through a transition of event  $e$ .

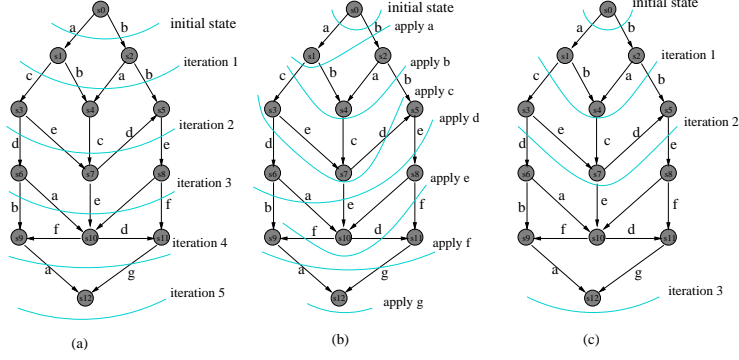
Although a TS is a powerful formalism, it is not usually used directly to specify concurrent systems. Instead, other high-level formalisms like Petri nets [12] or circuit structural descriptions are used, that later on are translated to transitions systems for analysis.

A Petri net (PN) is a 4-tuple  $N = \{\mathcal{P}, \mathcal{T}, \mathcal{W}, M_0\}$ , where  $\mathcal{P} = \{p_1, p_2, \dots, p_n\}$  and  $\mathcal{T} = \{t_1, t_2, \dots, t_m\}$  are finite sets of places and transitions satisfying  $\mathcal{P} \cap \mathcal{T} = \emptyset$  and  $\mathcal{P} \cup \mathcal{T} \neq \emptyset$ ;  $\mathcal{W} : (\mathcal{P} \times \mathcal{T}) \cup (\mathcal{T} \times \mathcal{P}) \rightarrow \mathbb{Z}$  defines the *weighted flow relation*, and  $M_0$  is the initial marking. The function  $M : \mathcal{P} \rightarrow \mathbb{N}$  is called a *marking*; that is, an assignment of a nonnegative integer to each place. If  $k$  is assigned to place  $p$ , we will say that  $p$  is marked with  $k$  tokens. If  $\mathcal{W}(u, v) > 0$  then there is an arc from  $u$  to  $v$  with *weight* (or multiplicity)  $\mathcal{W}(u, v)$ .

PNs are graphically represented by drawing places as circles, transitions as boxes or bars, the flow relation as directed arcs, and tokens as dots circumscribed into the places (see the example in Fig. 5).

## 3 Causality and chaining traversal

To speed up the generation of new states we combine two kinds of techniques: causality analysis and chaining. In traditional breadth first search (BFS) the TR



**Fig. 1.** Example of exploration process of a 13 state concurrent system using (a) BFS (b) BFS with *chaining* in lexicographical order of TR application (c) BFS with *chaining* in inverse lexicographical order.

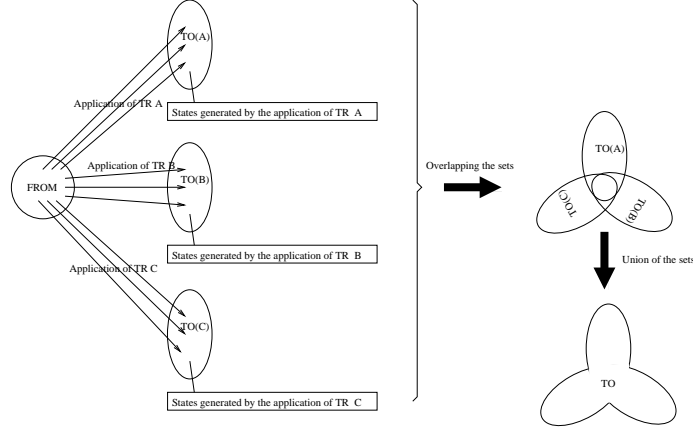
is applied to the same *from* set to generate a new *to* set. Using chaining after each TR application the *from* set is updated with the states recently generated. Thus, a domino effect is produced and more states are discovered in only one TR application. Figures 2 and 3 show the difference between BFS traversal and BFS traversal with chaining.

When chaining is used the order in which TRs are applied plays a crucial role. As an example, Fig. 1 shows a TS in which the behavior of symbolic traverse depends to a great extent on the selected TR application order. Each one of the subfigures in Fig. 1, shows the performance of different approaches on the same system. Subfigure (a) corresponds to a traditional BFS traversal. The progress of the reachability set is indicated by means of labeled arcs of type *iteration n*, indicating that all the states over that arc were discovered in BFS step *n*. Subfigures (b) and (c) show also a BFS approach, but using *chaining*. The difference between these two subfigures is the order in which the chaining of events is applied. In (b) we used lexicographical order (so in each step we applied the events as follow:  $\{a, b, c, d, e, f, g\}$ ), and in (c) we used inverse lexicographical order ( $\{g, f, e, d, c, b, a\}$ ). In this case the length of the traverse process was: (b) 1 iteration (c) 3 iterations. In (b) we show a detailed behavior of chaining and we draw the reachability set after each event is fired. As we can see all the system is traversed in only one step, while in (c) three steps are needed, although chaining is also used.

The state generation ratio of this technique may be limited if a TR application order is established that does not pay attention to causality between TRs.

The causality between pairs of TRs can be approximated by the following heuristic that tries to numerically indicate the *a priori* causal relationship between events. Let  $T_{e_i}$  and  $T_{e_j}$  be the TRs of two events  $e_i$  and  $e_j$ . We define  $XTo_{e_i \rightarrow e_j}(V)$  as

$$XTo_{e_i \rightarrow e_j}(V) = [\exists_{v \in V} [T_{e_i}(V, V') \cap Fr(e_i)(V) \cap \overline{Fr(e_j)(V)}]]_{V \leftarrow V'}$$



**Fig. 2.** State generation using BFS traversal.

(see Fig. 4). From now on we will avoid the overhead of explicitly stating the present set of Boolean variables  $V$  and next state set  $V'$  in the formulas. Therefore the previous formula will be rewritten as

$$XTo_{e_i \rightarrow e_j} = [\exists_{v \in V} [Te_i \cap Fr(e_i) \cap \overline{Fr(e_j)}]]_{V \leftarrow V'}$$

The  $XTo_{e_i \rightarrow e_j}$  operator simply gives us the set of states reached after the firing of event  $e_i$  from the states in which event  $e_j$  was not fireable. The heuristic *causality*( $e_i \rightarrow e_j$ ) is defined as

$$causality(e_i \rightarrow e_j) = \frac{|XTo_{e_i \rightarrow e_j} \cap Fr(e_j)|}{|Fr(e_j)|}$$

and indicates the proportion between the set found with the  $XTo$  operator and  $Fr(e_j)$ . Graphically, see Fig. 4(c), it is the proportion of the dashed area with respect to the whole  $Fr(e_j)$  set.

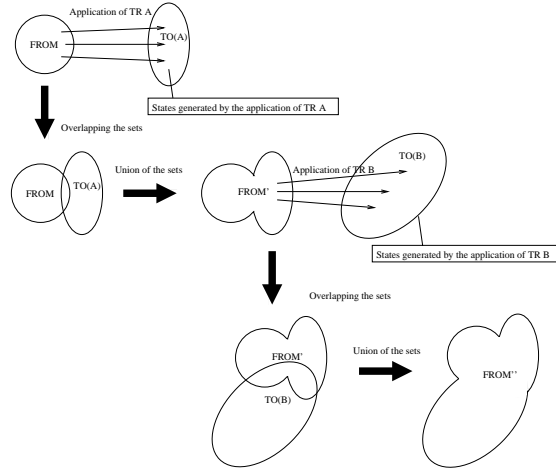
Intuitively, big values of *causality*( $e_i \rightarrow e_j$ ) show that the activation of TR  $Te_i$  will tend to produce states in which the application of TR  $Te_j$  is possible.

It must be noted that it is possible to define the symmetric heuristic of *causality*( $e_i \rightarrow e_j$ ), noted *negative\_causality*( $e_i \rightarrow e_j$ ) by defining the operator  $CTo_{e_i \rightarrow e_j}$  as

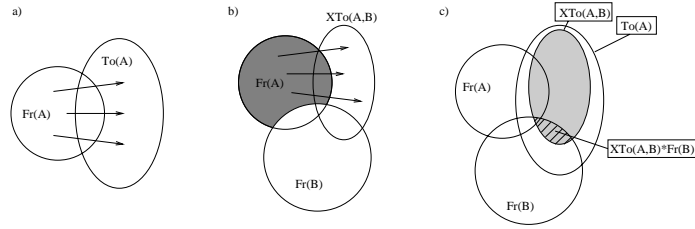
$$CTo_{e_i \rightarrow e_j} = [\exists_{v \in V} [Te_i \cap Fr(e_i) \cap Fr(e_j)]]_{V \leftarrow V'}$$

This function returns the set of states reached after firing event  $e_i$  from the states in which event  $e_j$  was fireable. *negative\_causality*( $e_i \rightarrow e_j$ ) is defined as:

$$negative\_causality(e_i \rightarrow e_j) = \frac{|CTo_{e_i \rightarrow e_j} \cap \overline{Fr(e_j)}|}{|CTo_{e_i \rightarrow e_j}|}$$



**Fig. 3.** State generation using chained traversal.



**Fig. 4.** The  $XT_{O_{A \rightarrow B}}$  operator: (a) shows the  $To$  operator, (b) depicts  $XT_{O_{A \rightarrow B}}$ , and (c) shows their relationship.

**Definition 1.** Two TRs  $A$  and  $B$  are said to be independent iff each one of the transitions of  $A$  falls into one of the following categories:

1. it goes from a state where TR  $B$  is fireable to a state where TR  $B$  is still fireable, or
2. it goes from a state where TR  $B$  is not fireable to a state where TR  $B$  is still not fireable.

and the same must hold if TR  $B$  is applied with respect the fireability of TR  $A$ .

**Theorem 1.** If two TRs  $A$  and  $B$  are independent  $\Leftrightarrow$   $causality(A \rightarrow B) = 0$  **and**  $negative\_causality(A \rightarrow B) = 0$  **and**  $causality(B \rightarrow A) = 0$  **and**  $negative\_causality(B \rightarrow A) = 0$ .

*Proof.* If  $A$  and  $B$  are independent the application of one of them to some set  $S$  of states cannot change the enableness/disableness of the other. Suppose in set  $S$  we have states from which  $B$  can be fired (set  $S_B$ ) and states in which it cannot (set  $S_{\overline{B}}$ ). If we apply  $A$  to  $S_B$  or  $S_{\overline{B}}$  there may be states that will not change. This

ones immediately satisfy the property abovementioned. The states that have changed must satisfy the following: if they were states of  $S_B$  the application of  $A$  must produce only states in which  $B$  is fireable ( $negative\_causality(A \rightarrow B)$  is then 0). If they were part of  $S_{\overline{B}}$  then the states generated cannot be in  $Fr(B)$  (so  $causality(A \rightarrow B)$  must be 0). The same holds if we exchange  $A$  and  $B$ .

It must be noted that this concept of independence may be viewed as a *strong independence* or *structural independence*, as it can happen that two dependent TRs behave, in fact, as independent given some particular initial states.

**Definition 2.** *The set of variables which constitute a formula  $\varphi$  is called support of  $\varphi$ , written as  $Sup(\varphi)$ .*

To specify the formula for a TR we use two sets of variables, one to represent *present state* states and another to represent *next state* states.

**Definition 3.** *Let  $V$  be the set of variables used to represent the present state, and  $V'$  the set of variables used to represent the next state. We define the function  $related(v)$  as a bijective function between  $V'$  and  $V$ . Given a variable  $v'$  in  $V'$ ,  $related(v')$  returns the corresponding variable  $v$  in  $V$ .*

*We extend function  $related(v)$  to sets of variables; i.e.  $related(V_a)$  returns the set of variables related to  $V_a$ . Formally  $related(V_a) = \{related(v) | v \in V_a\}$ .*

For instance, assuming  $V = \{p_1, p_2, p_3\}$  and  $V' = \{q_1, q_2, q_3\}$ , then  $related(q_3) = p_3$  and  $related(\{q_1, q_3\}) = \{p_1, p_3\}$ .

**Definition 4.** *An event  $e_i$  is said to have independent causal support from event  $e_j$  iff  $related(Sup(T_{e_j})) \cap Sup(T_{e_i}) = \emptyset$ .*

**Theorem 2.** *Events that have mutual independent causal support one from each other, are independent.*

*Proof.* if  $Related(Sup(e_j)) \cap Sup(e_i) = \emptyset$  is true, then event  $e_j$  is not able to *write* on the variables on which the enableness of  $e_i$  depends. Then, any state obtained from the activation of  $e_j$  will preserve the enableness of  $e_i$ . Thus,  $e_i$  is independent from  $e_j$ . The same can be stated by interchanging  $e_i$  and  $e_j$ , so  $e_i$  and  $e_j$  are independent events.

Theorem 2 can be used to simplify the computation of the causal matrix (see Section 4), as this independence check only involves variable set manipulation, which is usually very fast. Only for those events that do not satisfy this check we need to compute the *causality* heuristic to determine its final causality value.

## 4 Token traversal

Given a concurrent system, it is possible to compute  $causality(e_i \rightarrow e_j)$  for each pair of events (TRs), resulting in a *causality matrix*. This matrix can be analyzed in such a way that we produce a PN model of the event firing. This transformation is done as follows: suppose that each event is a place, then, for

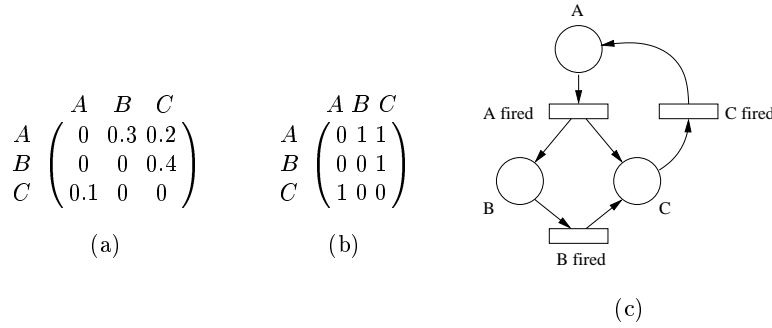


Fig. 5. Event PN inferred from the *causality matrix*.

every position of the matrix different than 0, we establish a relation between the places of the two events that are related to that matrix position. For instance, imagine the *causality matrix* for a three event system shown in Fig. 5(a).

All matrix positions that have a value greater than 0 are changed to 1, otherwise their value remains 0 (see Fig. 5(b)). The corresponding PN is depicted in Fig. 5(c). Although we use the same graphical representation as a PN, it does not behave as a normal PN as defined in Section 2. Instead, the traverse scheme always fires the place with more tokens in it. Obviously we must define some initial tokens. In order to do this we put a token in all places corresponding to events that are initially fireable; more precisely for each event  $e \in \Sigma : New \cap Fr(e) \neq \emptyset$ . Initially, the set of new states is equal to the initial set of states ( $New = From$ ). A brief outline of the algorithm is given below:

1. select place with the highest number of tokens
2. fire the event associated to this place
3. **if** the event has generated new states
4. **then** put one token on all successors
5. **else** absorb tokens

Figure 6 shows an example of the algorithm execution over the system represented by Fig. 5. We assume two initial tokens on events  $A$  and  $B$ . When there is more than one place with maximum number of tokens, one of them is chosen randomly. In our case event  $A$  was selected, although event  $B$  was also a possible election. Let us assume that event  $A$  generates new states (states not already visited), then one token is placed on its successors, that is place  $B$  and  $C$ . Next, event  $B$  is selected (the only possible choice this time) and is fired, successfully generating new states. As a result two tokens are placed on event  $C$  (the initial plus the token from  $B$ ), which is our next choice corresponding to the last state shown in the figure. Now consider what happens if event  $C$  is not successfully fired. All tokens on the net are absorbed, so no possible event can be selected afterwards. In this case, the algorithm starts up again, first by recalculating the

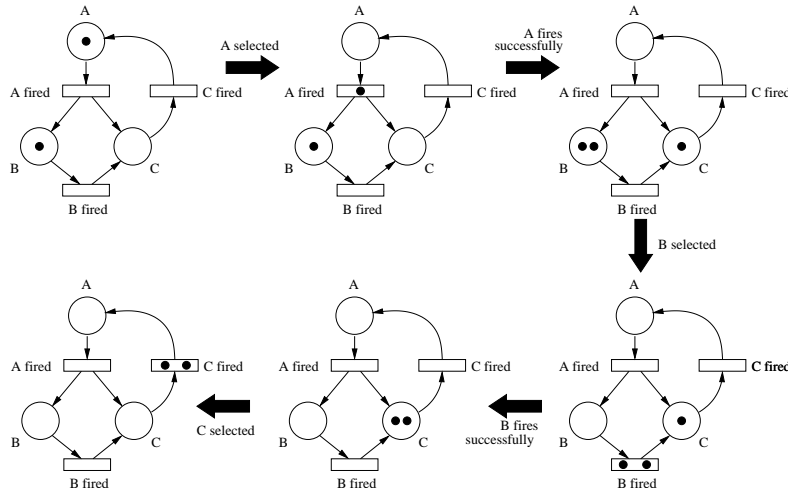


Fig. 6. Token firing scheme (TOK).

*New* set as the set of new states generated since last setup and then placing tokens in the events fireable given this present new set.

Proceeding so, the number of steps can be considered as the number of setups, and inside one step all firings use chaining to take advantage of the causality relation. The algorithm for TOK is shown next. The external loop is repeated until traversal is finished (no new states generated in the last step). The inner loop represents one step, we select events until all tokens are absorbed.

```

1. repeat
2.   oldFrom = from
3.   initial_tokens( net, new )
4.   stop = FALSE
5.   while (  $\neg$ stop )
6.     event = select_event_max_token( net )
7.     to = fire_event( event, from )
8.     if ( to  $\subseteq$  reached )
9.       absorb_token( net, event )
10.    else
11.      propagate_token( net, event )
12.      from = from  $\cup$  to
13.      reached = reached  $\cup$  to
14.      if ( no_more_tokens( net ) )
16.        stop = TRUE
17.      new = reached  $\setminus$  oldFrom
18.      from = new
19. until ( new =  $\emptyset$  )

```

We provide a brief definition of all functions called in this pseudo-code:

- *initial\_tokens* scans all events and adds a token to the corresponding place if the event is fireable in some state contained in *New* i.e.  $New \cap Fr(e) \neq \emptyset$ .
- *select\_event\_max\_token* selects the event that has more tokens in its corresponding place.
- *absorb\_token* just removes the tokens from the place assigned to the event passed as argument.
- *propagate\_token* removes the tokens from the place assigned to the event passed as argument and adds one token on all the successors of that event.
- *no\_more\_tokens* returns *true* if there is no token left in the net.

## 5 Weighed token traversal

It is possible to expand the preceding idea and consider that when an event is successfully fired we do not add only one token to its successors. Instead we can add a number of tokens related to the number of new states generated in which this particular successor is fireable. This would solve one of the problems of our previous proposal, the *ineffective activation*. The problem arises when a token is placed on one event because its predecessor generated new states, but there are no real new states in which this particular event can be fired. As a result its activation is superfluous. We will illustrate this problem with an example.

Suppose we have a TS with three variables  $V = \{p_1, p_2, p_3\}$  and three events  $A, B$  and  $C$ . To specify the TRs we also use an extra set of variables  $V' = \{q_1, q_2, q_3\}$  on the next state. The TR for the events is given below:

- **TR A:**  $\overline{p_1} \cdot q_1$
- **TR B:**  $\overline{p_2} \cdot q_2$
- **TR C:**  $p_1 \cdot p_2 \cdot \overline{q_1} \cdot \overline{q_2} \cdot q_3$

The initial state  $s_0$  is  $p_1 = 0, p_2 = 0, p_3 = 0$  that we write as 000. This system has the reachable set of states  $S$  that we depict in Figure 7. The causality matrix of this system (once all values greater than 0 are converted to 1) is:

$$\begin{array}{c} \\ A \\ B \\ C \end{array} \begin{array}{ccc} A & B & C \\ \left( \begin{array}{ccc} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{array} \right) \end{array}$$

which translates into the net of Fig. 7.

Applying the TOK scheme (see Section 4), Fig. 8 depicts the execution of the traversal on the example. We start at state 000, where events  $A$  and  $B$  can be activated. This is shown in the first net of Fig. 8 by the two tokens placed on places  $A$  and  $B$ . The algorithm may select  $A$  to fire. A token is placed on  $C$  as in the causality matrix  $A$  is related to  $C$ . However, the activation of  $A$  has only produced state 100, from which  $C$  cannot fire although a token has been placed on its place. Now the algorithm may select  $C$  to fire (3rd net on figure), that is a superfluous activation because no new state can be produced.



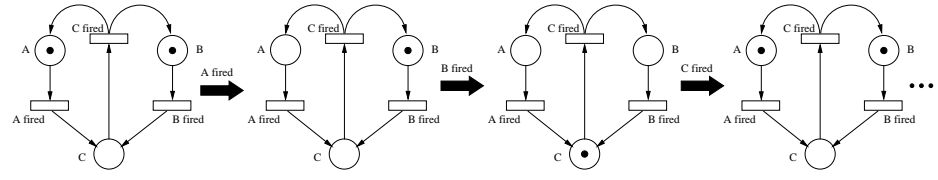
These “untracked” states are always states in which an event  $e_i$  was already fireable and then the activation of  $e_j$  added new fireable states to  $e_i$  (they were independent). Although they are not considered by the number of tokens, the algorithm indirectly keeps track of them because initial tokens are placed on all possible fireable events. In this example, although no additional token is added to the place of event  $B$ , there is *already* a token there and eventually  $B$  will be fired. Next we present the main WTOK traverse schema, which resembles the TOK algorithm:

```

1. repeat
2.    $oldFrom = from$ 
3.    $initial\_tokens( net, new )$ 
4.    $stop = FALSE$ 
5.   while ( $\neg stop$ )
6.      $event = select\_event\_max\_token( net )$ 
7.      $to = fire\_event( event, from )$ 
8.      $inNew = to \setminus reached$ 
9.      $distribute\_tokens( net, event, inNew )$ 
10.     $from = from \cup to$ 
11.     $reached = reached \cup to$ 
12.    if (  $no\_more\_tokens( net )$  )
13.       $stop = TRUE$ 
14.     $new = reached \setminus oldFrom$ 
15.     $from = new$ 
16. until ( $new = \emptyset$ )

```

Using this schema, the sequences of firings for the TS in Fig. 7 is shown in Fig. 9. Note that with respect to Fig. 8 the ineffective activation problem has been eliminated. Compared with TOK, WTOK allows a greater level of accuracy, but is computationally slightly more expensive, because for every possible successor a BDD AND operation is performed.



**Fig. 9.** Solution to the ineffective activation provided by the weighed token traverse.

## 6 Dynamic event-clustered traversal

We have seen that WTOK does not guarantee an exact equivalence between number of tokens and new states to fire from. The main problem are the *untracked* states produced by independent events firings. This is a side-effect of

using only causality to determine the successors events of an event, as we have already stated on the previous section.

Using causality was motivated to produce sequences of firings favorable to chaining. However, if we fire not only causal related events but also independent events, then the use of chaining is unadvisable. A generalized use of chaining usually implies larger execution times as all events are fired in each iteration.

To avoid the ineffective application of the TRs we propose to keep track of all states in which each particular event is enabled (DEC). Hence, we will store a From set for each event in the system (denoted  $From(e)$ ). This set should hold all states up to the current state of the reachability analysis from which the event has not been fired yet; that is, all new states for the event. When an event is fired from the set of states assigned to it, implicitly uses chaining.

The firing scheme is as follows. Given a set of new states, they are distributed over the events in the TS. Those states in which a certain event is enabled are associated to it and accumulated with other states that have been previously assigned. The set is updated as:  $From(e) = From(e) \cup (New \cap Fr(e))$ . The number of tokens “assigned” to each event is computed as the cardinality of the set  $From(e)$ . The event with greater number of fireable states is selected, the event fired, its  $From(e)$  set emptied, and the new states generated distributed again. The scheme ends when all events have an empty from set. The main algorithm is given below:

```

1. stop = FALSE
2. while ( $\neg stop$ )
3.   event = select_event_max_from( event_list )
4.   if (event = NULL)
5.     stop = TRUE
6.   else
7.     to = fire_event( event, event  $\rightarrow$  from )
8.     event  $\rightarrow$  from =  $\emptyset$ 
9.     new = to \ reached
10.    reached = reached  $\cup$  to
11.    distribute_tokens( new, event_list )

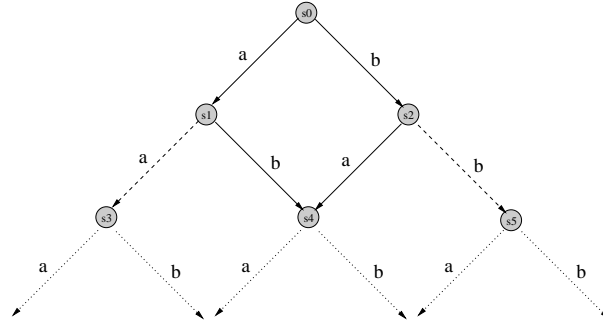
```

The price to pay for the exact knowledge this scheme provides, is an increased computational complexity. For every event activation, the state distribution process implies  $n$  BDD operations, being  $n$  the number of events. Compared to WTOK in which only  $k$  BDD operations were performed, being  $k$  the number of successors for that particular event. Another drawback is the BDD blowup problem, when the from sets tend to grow due to poor BDD recombination. To mitigate this problem the from sets are minimized using the reachability set.

## 7 Transition relation cluster-closure traversal

One of the main bottlenecks of symbolic verification is the size of the TR as a result of its monolithic structure. After partitioned TRs were introduced the bottleneck moved to the representation of the intermediate set of reachable states.

Event "a OR b" closure

**Fig. 10.** Closure of an ORed event.

In concurrent systems partitioned TRs is even more natural due to their inherent structure. However, the additional number of intermediate sets and BDD operations increases the probability of a BDD blowup.

We propose a firing scheme that reduces the number of TR applications by clustering subsets of events (TRCC). A monolithic closed TR is created for each cluster. Events are added to clusters incrementally. Without loss of generality, two events are clustered together by ORing their TRs. ORing produces a single TR whose activation has the same effect than the activation of both TR independently. Note that TR size is increased as the support variables in each TR is increased. Hence, clustering stops when a certain BDD size is reached. As a result, we perform less TR applications but normally more expensive.

In concurrent systems it is common to have *concurrency diamonds* due to events that are independent. In order to generate this diamond in only one firing we also *concatenate* TRs. This process is a particular case of *iterative squaring*. *Iterative squaring* is a powerful technique because when used with a monolithic TR it may exponentially reduce the number of steps required to complete the reachability analysis. Unfortunately, it is often the case that this is computationally too expensive. However, when transitive closure is used with smaller TRs it may be effective and computationally suitable. If we take a two-event TR and compute its closure, we obtain a TR that can compute at least the full *concurrency diamond* in one step. In fact more states can be discovered depending on whether the events can be iteratively activated or not (see Fig. 10).

In practice we add events to the events clusters iteratively. First we OR the TR of the new event and then compute the transitive closure of this new TR (usually we obtain smaller BDD sizes). Our approach does not assume any hierarchical structure in the system. To avoid an uncontrolled BDD growth we cluster the events that share as many variables as possible. In the results presented in Section 8, each event was clustered with some other event that had most variables in common. Doing so the number of events can be reduced at most to half of the original number.

## 8 Experimental results

All the results are from executions on a Pentium III 833 Mhz. On the following tables several concurrent systems are analyzed using the schemes described in this article. Due to space constraints we use abbreviations on the table. The correspondence between the abbreviations and the methods is:

**Seq** *BFS* traversal.

**GChain** *Greedy chaining* traversal.

**TOK** *Token* traverse (see Section 4).

**WTOK** *Weighed token* traversal (see Section 5).

**DEC** *Dynamic event-clustered* traversal (see Section 6).

**TRCC** *Transition relation cluster-closure* traversal (see Section 7).

For TRCC in some examples there is an additional entry. The default method is **TRCC**, but when appears written as **TRCC\*** indicates that the execution used manual clustering. As it is not always easy to define good partitioning schemes, we only report results when this was possible.

The *Greedy chaining* traversal is equal to the BFS algorithm (*i.e.* same firing order) with the only exception that makes use of the chaining technique (see Section 3).

Column (**Events**) shows the total number of event firings used to traverse the system. Note that some results are not directly comparable, *i.e.* TRCC reduces the number of events in the system. Column (**Peak**) shows the peak number of BDD nodes reached during traversal (in thousands). Finally, the last column specifies in seconds the wall-clock time needed to finish the analysis or *timeout* if the algorithm failed to finish within an hour (3600s). Sometimes, when the total time has been obtained, it is specified in brackets. Also when the time at which the algorithm was stopped has been bigger than an hour it is indicated with a “>” sign.

We analyzed different types of systems. Their characteristics are described on Table 1. Basic information on the size is given: number of Boolean variables, number of events and reachable states. The second column shows the original formalism of the system (before generating the equivalent **TS**): **C** for circuits and **P** for Petri nets. We give a brief description of the most relevant systems:

**RGD-arbiter** asP\*, RGD arbiter presented on [13] at transistor level.

**STARI (16)** A self-timed pipeline.

**Slotted ring (n)** *Slotted ring* protocol for LANs (*n* number of nodes).

**dme /DME (n)** Various DME implementations/specifications.

**Muller (n)** Muller’s C-element pipeline of *n* elements.

In all examples the TR application count is largely reduced (the original goal of this work). We can also see that the way in which TRs are applied also provide benefits in terms of CPU execution times and BDD-sizes. The RGD-arbiter, the slotted ring, the DME specification and Muller pipeline are examples in which almost any traversal scheme will provide improvements. On the contrary, the

STARI pipeline does not respond to any of the schemes, except TRCC when a set of clusters was manually provided. The motivation for this behavior is the structure of the pipeline itself: it is a deep structure with lots of concurrency at each level. Clustering the events in each step reduced the depth of the traversal and a BDD reduction due to the complete diamond generation in one step. More experiments are necessary in order to correlate the efficiency of each scheme with the topology of the system under analysis.

Name	Type	Variables	Events	Size
RGD-arbiter	C	63	47	5.49046e+13
STARI (16)	C	100	100	4.21776e+22
Slotted ring (10)	P	100	100	8.49079e+12
Slotted ring (15)	P	150	150	4.79344e+19
Slotted ring (20)	P	200	200	2.86471e+26
dme (3)	C	295	492	6579
dme (5)	C	491	820	859996
DME (8)	P	134	128	311296
DME (9)	P	152	144	3.2768e+06
parallelizer (16)	P	130	100	2.82111e+12
Muller (30)	P	120	60	6.009e+07
Muller (40)	P	160	80	4.64139e+10
Muller (50)	P	200	100	3.61071e+13
Muller (60)	P	240	120	8.38369e+15
buf (100)	P	200	101	1.26765e+30
sdLarq_deadlock	P	154	92	3954

**Table 1.** Concurrent systems under test.

Method	— RGD-arbiter —				— STARI (16) —			
	Steps	Events	Peak	Time (s)	Steps	Events	Peak	Time (s)
Seq	>38	>1786	>1755	[>14400]	>329	>33000	-	[>8800]
GChain	24	1175	1755	1476	127	12800	440	2435
TOK	8	1430	20	30	>34	N/A	[>1590]	[>10800]
WTOK	10	1280	20	26	67	10555	698	[8890]
DEC	N/A	1334	50	82	N/A	8135	572	[7997]
TRCC*	10	55	63	45	48	833	106	138
TRCC	17	468	1335	1501	110	5550	852	[4318]

Method	— slotted ring (10) —				— slotted ring (15) —			
	Steps	Events	Peak	Time (s)	Steps	Events	Peak	Time (s)
Seq	189	19000	445	4195	-	-	-	timeout
GChain	17	1800	68	65	24	3750	391	781
TOK	1	1486	17	16	1	3206	71	248
WTOK	1	1296	20	20	1	2690	220	414
DEC	N/A	2500	307	802	N/A	5621	893	[7196]
TRCC	12	780	32	15	18	1710	77	87

Method	— slotted ring (20) —				— parallelizer (16) —			
	Steps	Events	Peak	Time (s)	Steps	Events	Peak	Time (s)
Seq	-	-	-	timeout	99	10000	70	189
GChain	32	6600	1562	[5296]	5	600	20	26
TOK	1	5463	191	966	1	314	20	18
WTOK	1	4474	118	545	1	342	20	30
DEC	-	-	-	timeout	N/A	194	20	39
TRCC	22	2760	311	531	3	272	20	18

— dme (3) —				— dme (5) —				
Method	Steps	Events	Peak	Time (s)	Steps	Events	Peak	Time (s)
Seq	114	56580	150	289	>83	>68060	>1297	[>9900]
GChain	46	23124	70	105	86	71340	977	[4166]
TOK	1	2938	87	305	1	9453	1055	[9865]
WTOK	1	3235	78	156	1	10328	857	[11989]
DEC	N/A	544	45	103	N/A	2708	373	2321
TRCC	46	11562	77	145	86	35670	756	3089

— DME (8) —				— DME (9) —				
Method	Steps	Events	Peak	Time (s)	Steps	Events	Peak	Time (s)
Seq	40	5248	36	26	51	7488	51	82
GChain	12	1664	20	9	15	2304	20	18
TOK	1	545	26	20	1	690	20	22
WTOK	1	528	26	19	1	697	25	34
DEC	N/A	250	20	7	N/A	392	20	10
TRCC	12	936	20	11	15	1216	20	30

— Muller (30) —				— Muller (40) —				
Method	Steps	Events	Peak	Time (s)	Steps	Events	Peak	Time (s)
Seq	140	8460	258	1386	248	19920	1026	[15361]
GChain	23	1440	43	32	29	2400	103	151
TOK	1	901	20	16	1	1536	30	52
WTOK	1	774	20	16	1	1305	47	59
DEC	N/A	666	113	98	N/A	>211	-	timeout
TRCC	23	720	41	17	29	1200	103	79

— Muller (50) —				— Muller (60) —				
Method	Steps	Events	Peak	Time (s)	Steps	Events	Peak	Time (s)
Seq	-	-	-	timeout	-	-	-	timeout
GChain	35	3600	219	456	43	5280	431	907
TOK	1	2336	57	155	1	3185	80	244
WTOK	1	1965	57	111	1	2763	155	320
DEC	-	-	-	timeout	-	-	-	timeout
TRCC	35	1800	213	246	43	2640	429	582

— buf100 —				— sdl_arq_deadlock —				
Method	Steps	Events	Peak	Time (s)	Steps	Events	Peak	Time (s)
Seq	>352	>35552	-	[>8100]	120	11132	42	35
GChain	100	10201	13	7	40	3772	20	7
TOK	1	10202	51	690	1	1354	15	3
WTOK	1	6200	21	155	1	1242	15	3
DEC	N/A	7864	-	[13407]	N/A	448	20	8
TRCC	100	5151	334	595	35	1800	22	22

## 9 Conclusions

This paper proposes four different schemes to speed up reachability analysis on concurrent systems. Their main contribution is to establish different heuristic orderings for the application of the TRs that can reduce substantially the time required to generate the full state space. Although firing order has been studied on state reduction techniques (i.e partial order [14]), to our knowledge this is the first time this issue is addressed to generate all the reachable states for concurrent systems.

Experimental evidence has been given that the methods proposed are most times faster than a classical BFS approach or even a BFS with chaining. For all benchmarks, the use of the simple *greedy chaining* (BFS) scheme has proved to be very useful. However it is important to note that at least one of the proposed schemes always performed better than the latter. It remains as an open problem to decide a priori which method is more suitable for a given system. If this could not be decided on a reasonable amount of time there is always the possibility to try all the schemes sequentially or in parallel.

## References

1. R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Computers*, vol. C-35, pp. 677–691, Aug. 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. O. Roig, J. Cortadella, and E. Pastor, "Verification of asynchronous circuits by bdd-based model checking of petri nets," in *16th International Conference on Application and Theory of Petri Nets*, pp. 374–391, June 1995.
4. J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, "Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers," *IEICE Transactions on Information and Systems*, vol. E80-D, no. 3, pp. 315–325, March 1997.
5. A. S. Miner and G. Ciardo, "Efficient reachability set generation and storage using decision diagrams," in *ICATPN*, pp. 6–25, 1999.
6. J. C. E. Pastor and O. Roig, "Symbolic analysis of bounded petri nets," *IEEE Transactions on Computers*, vol. 50, no. 5, pp. pp. 432–448, May 2001.
7. K. Ravi and F. Somenzi, "High-density reachability analysis," in *Proc. of the IEEE/ACM International Conference on Computer Aided Design*, pp. 154–158, 1995.
8. G. Cabodi, P. Camurati, and S. Quer, "Improving symbolic traversals by means of activity profiles," in *Design Automation Conference*, pp. 306–311, 1999.
9. A. Hett, C. Scholl, and B. Becker, "State traversal guided by hamming distance profiles."
10. K. Ravi and F. Somenzi, "Hints to accelerate symbolic traversal," in *Conference on Correct Hardware Design and Verification Methods*, pp. 250–264, 1999.
11. A. Arnold, *Finite Transition Systems*. Prentice Hall, 1994.
12. C. Petri, *Kommunikation mit Automaten*. PhD thesis, Schriften des Institutes für Instrumentelle Mathematik, Bonn, 1962.
13. M. R. Greenstreet and T. Ono-Tesfaye, "A fast, ASP\*, RGD arbiter," in *Proceedings of the Fifth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, (Barcelona, Spain), pp. 173–185, IEEE, Apr. 1999.
14. P. Godefroid, *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, vol. 1032. New York, NY, USA: Springer-Verlag Inc., 1996.