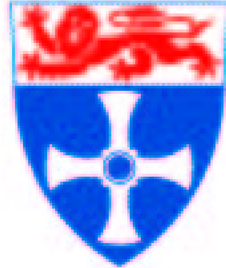
Department of Computing Science,
University of Newcastle upon Tyne



ICU: A tool for Identifying State Coding Conflicts using STG unfoldings

A. Madalinski, A. Bystrov, A. Yakovlev

Technical Report Series

June 2002

Contact:
A.A.Madalinski@ncl.ac.uk
A.Bystrov@ncl.ac.uk
Alex.Yakovlev@ncl.ac.uk

Copyright©2002 University of Newcastle upon Tyne
Published by the University of Newcastle upon Tyne,
Department of Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, UK

ICU: A tool for Identifying State Coding Conflicts using STG unfoldings

A. Madalinski, A. Bystrov and A. Yakovlev

June 2002

Abstract

State coding conflict detection is a fundamental part of the synthesis of asynchronous concurrent systems from their Signal Transition Graph (STG) specifications. This paper presents the extension of the method proposed earlier, the identification of state coding conflicts in STGs which is intended to work within a synthesis framework based on STG unfoldings. This approach has been implemented as a software tool using refined algorithms. A necessary condition detects state coding conflicts by using an approximate state covering approach. Being computationally efficient, this algorithm may generate false alarms. Thus a refinement technique is applied based on partial construction of the state space with extra computational cost. The experimental results demonstrating the efficiency of this approach are presented.

1 Introduction

Methods for the synthesis of asynchronous circuits from Signal Transition Graphs (STGs), which are Petri Nets whose events are interpreted with signal transitions of a modelled circuit, have commonly used State Graphs (SGs) to solve the main steps of the synthesis process, viz. the state assignment and the generation of hazard-free logic. Existing synthesis tools , such as Petrify [1], use this approach for the synthesis of asynchronous circuits by direct construction of the full reachable state space and extraction of subsets of states required for implementation.

An obvious practical limitation of this approach is the potential combinatorial growth of the number of reachable states. To avoid the construction of the full reachable state space, partial order based methods can be used. A compact representation of STG state space is provided by PN unfolding [2][3]. It is known that its finite complete prefix, a truncated unfolding, completely represents the entire reachability graph of the PN. The size of a finite complete prefix is typically larger than the original PN and significantly smaller than the SG.

STG unfoldings are already used to verify STG specifications efficiently but there is little research in using unfoldings in asynchronous circuits synthesis. The previously known method [4] for deriving logic circuits from STG unfoldings offered an important conceptual approach based on approximated boolean covers and has proved to be promising in dealing with large STG models. However, it cannot be applied to STGs having state coding conflicts. Such conflicts happen when a pair of different states in the SG have the same binary encoding. A necessary and sufficient condition for STGs to derive hazard-free circuits is the Complete State Coding (CSC) condition.

The method in [5] addresses the problem of identification of state coding conflicts in STGs and is intended to work within the synthesis framework based on STG unfoldings. This method is the extension of [6]. It has been implemented as a software tool ICU where the detection of coding conflicts is divided into two stages. The first stage exploits the necessary conditions for state coding conflicts such as partial coding information,

represented as place cover approximations, and information from the computation of maximal trees. While these conditions are computationally efficient (quadratic in the size of the unfolding) they may hide non-existing "fake" conflicts. At the second stage sufficient conditions are used based on partial construction of the state space. The state space construction is computationally expensive, though this is done only for small parts of the unfolding identified in the first stage.

2 Basic notions

This section introduces the basic definitions and notations used in this paper. These include models, such as STG and STG unfolding, state coding conflicts and cover approximation.

2.1 Signal Transition Graph and state coding problems

A Petri Net (PN) is a quadruple $PN = \langle P, T, F, m_o \rangle$, with sets of places P, transitions T, flow relation F and initial marking m_o. A marking m is represented with a number of tokens $m(p)$ in each place $p \in P$. A Signal Transition Graph (STG) is a triple $N = \langle PN, A, \lambda \rangle$, where PN is a PN, $A = I \cup O$ is a set of signals partitioned into input and output signals, and $\lambda : T \to A \times \{+, -\}$ is the labelling function that assigns a signal edge name to each transition in T. An STG is thus a labelled PN, used to describe the behaviour of asynchronous circuits at the logic level. The set of transitions represents signal changes, i.e. their rising (a_i+) and falling (a_i-) edges. The notation (a_i*) is used to indicate a signal transition regardless of the direction of the change. Given a Petri net element $x \in T \cup P$, its predecessor and successor sets are denoted $\bullet x$ and $x\bullet$ respectively. It is assumed that for any transition $t \in T : \bullet t \neq 0$ and $t\bullet \neq \emptyset$.

An STG is called *k-bounded* iff the number of tokens in any place $p \in P$ at any reachable marking does not exceed k. Boundedness guarantees that the behaviour specified by an STG can be implemented into a finite sized circuit. An STG is *output signal persistent* iff no output signal transition a_i* excited at any reachable marking can be disabled by the transition of another signal a_j*. If an STG is output signal persistent, then it can be implemented without producing unspecified changes to the output signals, thus not introducing hazards.

To obtain an implementation for an STG the State Graph (SG) is derived by constructing the Reachability Graph (RG) for the STG (representing all reachable states), and assigning a binary code $v \in \{0,1\}^n, n = |A|$ to each state. Thus an SG is a triple $SG = \langle S, E, \gamma \rangle$, where S is a set of binary encoded states $s = (m, v)$, E is a set of transitions between states, and $\gamma : E \to A \times \{+, -\}$ is a function that labels the arcs between states with signal transitions. The binary codes v must be assigned to their markings m consistently. An SG is *consistent* if for each transition $s \stackrel{e}{\to} s'$ the following conditions hold:

- if $\gamma(e) = x+$, then $s(x) = 0$ and $s'(x) = 1$
- if $\gamma(e) = x-$, then $s(x) = 1$ and $s'(x) = 0$
- in all other cases $s(x) = s'(x)$.

An STG is called *consistent* if its SG has a consistent state assignment. An STG has the *Unique State Coding* (USC) property when two different states in the SG do not have the same binary coding. When two different states are given the same binary representation, the digital circuit cannot distinguish the two states from each other. Therefore, this is a requirement that should be satisfied before logic equations can be derived. The USC requirement is sufficient, but not necessary to get a hazard-free implementation. A necessary and sufficient condition is the *Complete State Coding* (CSC) property. Two states may have the same code iff the transitions of non-input signals that are enabled in the two states are the same.

2.2 STG Unfolding

An STG unfolding built for an STG N is $N' = \langle T', P', F', \Lambda \rangle$ where T', P' and F' are sets of transitions, places and the flow relation, respectively; and Λ is a labelling function which labels each element of N' as an instance of elements of N. N' is a partial order obtained from an STG N by the process of its unfolding which starts from its initial marking. The unfolding process uses the structural properties of the constructed partial order to determine the relation of *precedence*, *conflict* and *concurrency* between instances. These relations are constructed during the unfolding process from the basic flow relation F', built from the flow relation F from the original STG. More formally, a transitively closed (w.r.t. F', which defines immediate predecessors of a place or transition instance) set of unfolding elements for an instance x' is called the *history of* x'. For any pair of instances $x'_1, x'_2 \in P' \cup T'$ in the unfolding, three relations can be defined:

- *Precedence*, denoted as $x'_1 \Rightarrow x'_2$, iff x'_1 belongs to the history of x'_2.
- *Conflict*, denoted as $x'_1 \# x'_2$, iff there exist two distinct transitions t'_1 and t'_2 in the histories of x'_1 and x'_2, respectively, such that $\bullet t'_1 \cap \bullet t'_2 \neq \emptyset$.
- *Concurrency*, denoted as $x'_1 \parallel x'_2$, iff x'_1 and x'_2 are neither in precedence nor in conflict.

In contrast to PN unfolding [2][3], the STG unfolding takes into account specific signal interpretations of PN transitions and keeps track of the binary codes reached by the transition firing. However, it explicitly represents only a subset of all reachable states of N and thus is typically more compact than an SG.

The set of causal predecessor transitions of t' of the STG unfolding is called the *local configuration* of t' , denoted as $\Rightarrow t'$. A set of place instances reached by firing all transitions in $\Rightarrow t'$ is called the postset of $\Rightarrow t'$, denoted as $(\Rightarrow t') \bullet$. Mapping a postset onto places of the original STG gives a marking of the original STG, called a *basic marking* denoted as $m (\Rightarrow t')$. Any non-conflicting and transitively closed (w.r.t. the precedence relation) subset of transitions $T'_1 \subseteq T'$ is called a *configuration*.

Each instance t' of the STG unfolding has a binary code $v (\Rightarrow t')$ which is reachable by firing transitions in $\Rightarrow t'$. The postset $(\Rightarrow T'_1) \bullet$ and the binary code $v (\Rightarrow T'_1)$ corresponding to a configuration T'_1 are calculated from $(\Rightarrow t') \bullet$ and $v (\Rightarrow t')$ of transitions comprising it. The pair $(m (\Rightarrow t'), v (\Rightarrow t'))$ is called the *final state* of the local configuration $\Rightarrow t'$. It was shown in [7] that all states of the SG are represented in the STG unfolding as postsets of some configuration.

The process of constructing the STG unfolding (which is a finite object for the bounded PN) is terminated at the transition instances called *cut-off events* , whose final state is equal to the final state of some other instance already presented in the unfolding. There exist several definitions of the cut-off event [2][3], different in their attempts to minimise the size of the truncated PN (or STG) unfolding necessary to fully represent the SG. The initial state of the STG is associated with an imaginary *initial transition* in the unfolding, whose postset is the set of place instances of the places involved in the initial marking.

Cuts To represent a state of the SG in the unfolding the notion of *cut* was introduced. A cut of an STG unfolding is a maximal set, w.r.t. set inclusion, of pairwise concurrent conditions (places). Thus each cut $m' \in P'$ represents some reachable marking of the original STG. Due to the main property of the STG unfolding, which is to represent all reachable states of an STG, for every reachable state in an STG there is a cut in the STG unfolding. However, the unfolding may cover some markings more than once, i.e. several cuts may be mapped to the same marking due to the acyclic nature of the truncated STG unfolding.

Similar to markings, each cut $m' \in P'$ is also associated with a binary code $v (m')$ having the marking of the original STG. Since a cut m' represents a state s, there exists a configuration T'_1 such that $s = (m (\Rightarrow T'_1), v (\Rightarrow T'_1))$. The notation of CSC can therefore be rephrased as follows. Two cuts m'_1 and m'_2 are said to be in a CSC conflict iff $v (m'_1) = v (m'_2)$ and they enable different transitions of output signals.

Slices Slices represent a connected set of states of the SG. A *slice* of an STG unfolding is a set of cuts $S = \langle m'^{min}, M'^{max} \rangle$ defined with the min-cut of the slice m'^{min} and a set of max-cuts M'^{max}, such that $\forall m'_i \in S$ and the following is true: $m'^{min} \Rightarrow m'_i$ and $\exists m'_j \in M'^{max} : m'_i \Rightarrow m'_j$, and no two cuts in the set of max-cuts in M'^{max} are mutually ordered by the precedence relation. Every cut in between the min-cut and max-cut is encapsulated in the slice S.

Since each cut represents some state in the SG, for any two states s_i and s_j represented as ordered cuts in S, all states on any path from s_i to s_j are also represented as cuts encapsulated into S. The number of cuts in the set of max-cuts corresponds to the number of configurations which include configuration producing the min-cut. The elements of the STG unfolding (places, transitions and arcs) bounded by the instances in min-cut and max-cuts are said to belong to the slice.

Due to the binary encoding associated with every cut in an STG unfolding, each slice can be associated with a boolean cover obtained as the sum of the min-terms corresponding to the cuts contained in the slice.

A special kind of a slice is the *marked region* for a place $p' \in P'$, which is the set of cuts to which p' belongs, denoted as $MR(p')$.

Cover approximation The *cover approximation of place* p' is the cube $C(p') = c[1]c[2]...c[n]$, where $n = \mid A \mid$ is the number of signals in the STG and $\forall i : c[i] \in \{0, 1, -\}$, and is computed as follows:

- $c[i] =' -'$ if $\exists a_i *$ such that $a_i * \parallel p'$, and
- $c[i] = v(\Rightarrow t')[i]$, otherwise.

The cover approximation of a place p' is obtained from the binary code assigned to its preceding transition t'. Any marking including p can only be reached from $\Rightarrow t'$ by firing transitions concurrent to p'. The literals corresponding to signals whose instances are concurrent to p' are replaced by '-'. The approximated covers defined above are cubes derived only by knowing local configurations of unfolding transitions and concurrency relations between places and transitions. All information can be derived in polynomial time from the unfolding.

3 Detection of state coding conflicts by STG unfolding

In this section the overview of the approach to detect coding conflicts by STG unfoldings proposed in [6] is presented. A necessary condition to identify coding conflicts by using an approximate state cover approach is described. While this condition includes real conflicts, and is low in complexity, it may hide the so called "fake" conflicts. To resolve this situation a refinement technique, which requires extra computational cost, has to be applied.

3.1 Necessary condition

The information on the state codes in the unfolding can be obtained from place cover approximations avoiding the complete state traversal. First a conservative check for coding conflict is done, based on cover approximation.

Places p'_1 and p'_2 are said to be in *collision* in an STG unfolding if their cover approximation intersect, i.e. $C\,(p'_1) \cap C\,(p'_2) \neq 0$. There are three sources of collision between a pair of place instances:

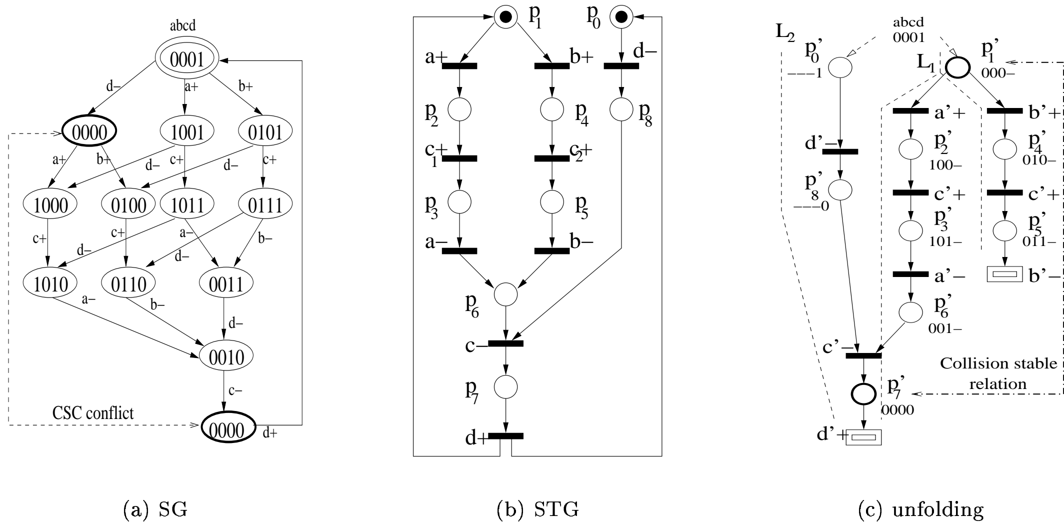
1. The marked regions of places p'_1 and p'_2 contain only cuts that are mapped to the same marking of the original STG, i.e. there is no state coding conflict.
2. The exact boolean covers of the marked regions of a given pair of instances in collision do not contain the same binary codes but the place cover approximations $C\,(p'_1)$ and $C\,(p'_2)$ intersect due to an overestimation. Thus this collision does not correspond to a state coding conflict.

3. In the marked regions of p'_1 and p'_2 there are two cuts that are mapped to two different markings with the same binary encoding. This results in a state coding conflict and may or may not be a CSC conflict depending on whether these markings enable different sets of output signals.

The idea of detection of state coding conflicts by approximate techniques uses collisions, which can be easily analysed, instead of actual coding conflicts. However, this can be overly conservative because only 3 is of interest, while 1 and 2 must be excluded. To make this analysis less conservative, as many fake collisions as possible need to be identified. A collision between places p'_1 and p'_2 is called *fake* if no cut in the marked region of p'_1 is in state coding conflict with cuts from the marked region of p'_2.

The identification of fake collisions can be achieved by exploring information about maximal trees involving places in collisions. A *maximal tree* identifies sets of places that can never be marked together (because they are ordered or are in conflict), and whose marked regions contain all reachable cuts of an unfolding. A maximal, w.r.t. set inclusion, fragment of an unfolding without concurrency is represented by a maximal tree.

A place p' of an STG unfolding is called *collision stable* if every maximal tree passing through p' contains another place p'_1 which is in collision with p'. If an original STG has a state coding conflict then its unfolding contains a pair p'_1, p'_2 of collision stable places. This means that if an STG has coding conflicts, then there are at least two places in the STG unfolding each of which is in collision with another place in every maximal tree. This fact will be used as a characteristic property of a state coding conflict in terms of cover approximation. This property is necessary but not sufficient. The STG with no state coding conflicts may have stable collision places. This can happen due to an overestimation of cover approximations and reflects the conservative nature of this approach.



(a) SG (b) STG (c) unfolding

Figure 1: An example for the approximation technique

Consider the STG and its SG represented in Figure 1(a) and (b), respectively. The SG shows a CSC conflict between the pair of states 0*0*00 and 0000* (output signal d is not enabled in the first state but is enabled in the second). Figure 1(c) shows the unfolding of the STG. The place cover approximation are presented next to their place instances. Place p'_1 is concurrent to $d'-$ and is ordered with the signals of a, b and c, hence the cover approximation is 000-. Finding collision stable places requires the consideration of maximal trees. There are two maximal trees L_1 and L_2 denoted by dashed lines. In the maximal tree L_1 places p'_1 and p'_7 are in collision. The only maximal tree passing through p'_1 is L_1 thus p'_1 is a collision stable place. Place p'_7 belongs
p'_1 The place p' with p'_8 in L_2, hence stable
s not satisfy the CSC property.

A reduction of fake collisions can be achieved in unfoldings, where their size can be reduced by transferring initial markings (see below). The number of multiple place instances are reduced while reducing the size of the unfolding. This results in a reduction of fake collisions between place instances which are mapped on the same place in the original STG. However, the only way to detect all fake collisions is to explore the set of states to which they belong.

Minimising the size of unfolding Reducing the size of the STG unfolding by transferring the initial marking can result in fewer fake conflicts. The size of an unfolding depends onto the choice of the initial marking of the original specification. By choosing a "good" initial marking the size of the constructed unfolding can be reduced. However, the choice of the initial marking must not change the set of markings covered by the unfolding. Thus a necessary condition to change the initial marking from m_{0_1} to m_{0_2} is that in both cases the unfolding should contain all reachable markings of the original STG. This change will be referred as *transferring* the initial marking.

A particular case of STG in which the initial marking can be transferred without changing the specification is an STG with a strongly connected reachability graph. The initial marking of the STG is a *home marking* iff the reachability graph of the STG is strongly connected [8].

In [8] a procedure of finding a "good" initial marking in an STG was proposed which suggests the use of a *stable basic marking* as the initial one. In an unfolding a stable basic marking is a marking by which, in an unfolding construction, a cut-off is made (the final state of local configuration of a cut-off event). The initial marking of the unfolding can be transferred to this marking in order to reduce its size. A new unfolding is constructed using the new initial marking.

3.2 Refinement by partial state construction

A straightforward way to check whether a collision corresponds to a real state coding conflict is to construct all states corresponding to a cube, which is the intersection of the covers of given places in collision, in the marked regions of places belonging to the collision. If conflicting states are found the nature of the conflict (i.e. USC/ CSC conflict) can be determined by comparison of the transitions enabled in these states.

The advantage of this approach is that it gives the exact information on coding conflicts, while its difficulty lies in the high cost of the state construction, which is exponential to the specification size. However, in practice the marked region of a place often contains much less states than the entire unfolding. Furthermore, only parts of these states belong to the intersection of covers.

The task of detecting coding conflicts from a pair of collision places p'_1 and p'_2 consists of the following steps:

1. Let $C(p'_1)$ and $C(p'_2)$ be the covers approximating p'_1 and p'_2 and $c = C(p'_1) \bigcap C(p'_2) \neq 0$.
2. Find the intersection of the ON-set of c with the marked regions of p'_1 and p'_2 denoted by $ON(p'_1)$ and $ON(p'_2)$.
3. Construct the binary states of $ON(p'_1)$ and $ON(p'_2)$.
4. Check for state coding conflicts using the binary states of $ON(p'_1)$ and $ON(p'_2)$.

All the steps of this procedure are trivial to implement with the exception of Step 2. To construct the states corresponding to some cube c it is necessary to identify all regions where the cube c evaluates to 1 in the marked regions of places in a collision called ON-regions.

4 Implementation

This section presents the implementation of the approach described in the previous section. The original algorithms have been refined and are described here in detail. The STG unfolding and the place cover approximations are obtained by PUNT [4]. The procedure which identifies collision stable places and the procedure which constructs the ON-set are presented first, followed by the overall procedure and further refinements.

4.1 Collision stable places

The necessary condition requires the refinement of collisions by collision stable places. The algorithm proposed in [6] uses information extracted from all possible trees, without enumerating all of them. This algorithm looks for one maximal tree where a place is free from collisions. If such a tree exists, the given place is not a collision stable place. In this algorithm, all places which are concurrent with the given place and all places with which the given place is in collision are removed. Other nodes that can not be included in any other maximal tree because of the removal of nodes in the previous step are also removed. If the given place has not been deleted, then it is not a collision stable place.

A recursive algorithm was designed to check if a place p'_{in} is in collision with another place in every maximal tree passing through p'_{in} (Algorithm 1). This algorithm attempts to find a place in collision with p'_{in} in every tree preceding p'_{in} or, if none are found, in every tree succeeding p'_{in}, without enumerating all of them. It is based on node traversal in the unfolding *traverseEvent* and *traverseCondition* (Algorithms 2 and 3). The traversal algorithms apply conditions to traverse a special kind of node and use flags to avoid the repeated traversal of nodes.

The procedure *isCollisionStablePlace* returns "true" if p'_{in} is collision stable, otherwise "false". First the predecessors of p'_{in} are traversed backwards starting from the preceding transition of p'_{in}. If a tree is found in which p'_{in} is free from collision the traversal is stopped and the successors of p'_{in} are explored. Otherwise the traversal is continued until in every tree preceding p'_{in} a place is found which is in collision with p'_{in}. Hence p'_{in} is a collision stable place. Note that during the traversal a visited node is assigned a flag which indicates that it has already been visited. In addition, a node flag records the information to indicate whether its visited part contains collisions with p'_{in}. This ensures that nodes are only visited once.

In the case of the backward traversal not being successful, the successors of p'_{in} are traversed forward. Then every direct successor of p'_{in} is checked for the existence of a place in every tree succeeding $p_{in}\bullet'$ which is in collision with p'_{in}. This is done similarly to the backwards traversal.

Algorithm 1 Identifying collision stable places

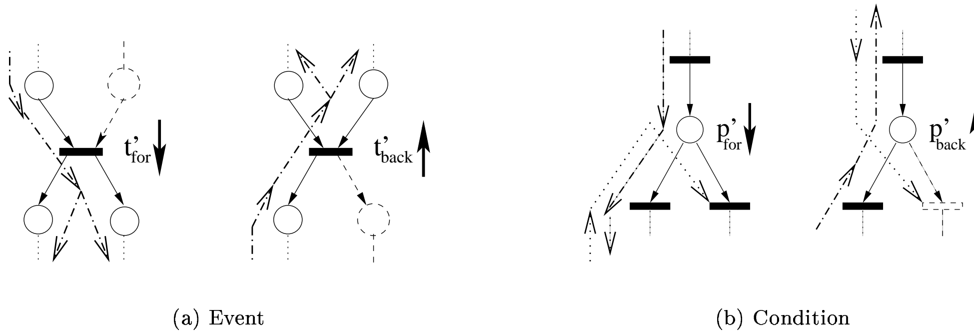
```
isCollisionStablePlace(p'_in)
{
  forall (node ∈ T' ⋃ P' )  //reset node flags in unfolding
    setFlag(node,0)
  if (traverseEvent(bwd, t_pre = getFirst(•p'_in)) == true)   //start traversing t'_pre backward
    return true
  forall (t' ∈ p'_in•)
  {
    if (traverseEvent(fwd, t') == true)   //start traversing t' forward
    {
      setFlag(t',2)
      return true
    }
  }
  return false
}
```

The procedures for traversing an event and a condition are presented in Algorithms 2 and 3. Each procedure requires two input parameters. One parameter is the node to traverse and the other parameter is the traversal

direction, forwards or backwards. The procedures return "true" if a collision is found, otherwise "false". The rules for traversing a node in the unfolding are described below.

Traversing an event Several maximal trees can pass through the transition instance t'. The given place p'_{in} is collision stable if a collision exists with p'_{in} in every tree preceding or succeeding t'. In the case of forward traversal in the Algorithm 2 only the successors of t' are considered. For example the transition t'_{for} in Figure 2(a) is traversed forwards as follows. The postset of t'_{for} is checked for the existence of collisions in such a way that either $t'_{for}\bullet$ are in collision with p'_{in} or there is a collision with p'_{in} in the tree succeeding $t'_{for}\bullet$. This is done by using node flags and the results which were obtained from procedure *traverseCondition*. If the succeeding nodes of t'_{for} have already been traversed they contain information about the existence of collisions. This ensures that these nodes are only traversed once.

The backwards traversal is performed in a similar way. The predecessors of, e.g. the event t'_{back} in Figure 2(a) are checked for the existence of collisions.



(a) Event (b) Condition

Figure 2: Traversing a node

Traversing a condition A place instance p' traversed by the procedure *traverseCondition* (Algorithm 3) can either be in collision with p'_{in} or in collision with places in the trees preceding or succeeding p'. If p' and p'_{in} are not in collision the procedure checks if collisions exist in the preceding and succeeding conditions. Consider the forward traversal of the place p'_{for} in the Figure 2(b). The procedure tries to find a place in collision with p'_{in} in at least one of the choice branches of p'_{for} using the procedure *traverseEvent*. At least one collision is necessary because the succeeding trees of p'_{for} belong to a maximal tree passing through p'_{for}.

The backwards traversal is illustrated in the right part of Figure 2(b). If p'_{back} is not in collision with p'_{in} then the predecessors of p'_{back} are scanned for the existence of collisions. Note that in the unfolding a place instance has always only one direct predecessor. If this fails, the forward traversal is applied to the successors of p'_{back}, where only the choice branches which have not been visited are considered.

4.2 ON-set of a collision relation

The algorithm in [6] calculates the ON-set as follows. It first finds all min-cuts of the given cube c that are reachable from the initial marking without passing through another min-cut. For all these min-cuts it constructs the corresponding ON-slices by calculating the matching set of max-cuts. This is done by restricting the unfolding as follows. The part of the unfolding preceding or in conflict with the given min-cut is irrelevant, since all max-cuts from the matching set succeed they min-cut. The nodes that succeed transitions that reset the cube c are removed from the unfolding. However, the unfolding may have other min-cuts of c that succeed cuts from the set of the obtained min-cuts, the cube c can be set and reset several times. To identify those the procedure of finding the ON-set is iterated.

Algorithm 2 Procedure to traverse an event

```
traverseEvent(direction, t')
{
  if (getFlag(t') == 2)   //check if conflict exists
    return true
  setFlag(t',1)   //tag flag of t' that it is visited
  if (direction == fwd)
  {   //traverse forwards
    if (t' is cut-off event)   //forward traversal possible
      return false
    forall (p' ∈ t'•)   //check if all succeeding trees contain a collision
    {
      if ((getFlag(p') ≠ 1) and (traverseCondition(fwd, p')))
        setFlag(p', 2)
      else
        return false
    }
    return true
  }
  else
  {   //traverse backwards
    if (t' is initial transition)   //backward traversal possible
      return false
    forall (p' ∈ •t')   //check if all preceding trees contain a collision
    {
      if ((getFlag(p') ≠ 1) and (traverseCondition(bwd, p')))
        setFlag(p', 2)
      else
        return false
    }
    return true
  }
}
```

Algorithm 3 Procedure to traverse an condition

```
traverseCondition(direction, p')
{
  if ((getFlag(p') == 2) or (C(p') ⋂ C(p'_in) ≠ 0))   //check if conflict exists
    return true
  setFlag(p', 1)   //tag flag of p' that it is visited
  if (direction == fwd)   //traverse forwards
  {
    forall (t' ∈ p'•)   //check if there is a collision in t'
    {
      if (getFlag(t') ≠ 1) and (traverseEvent(fwd, t'))
      {
        setFlag(t',2)
        return true
      }
    }
  }
  else
  {   //traverse backwards
    t_pre = t' : •p' = {t'}   //in the unfolding a place instance has at most one incoming arc
    if (getFlag(t'_pre) ≠ 1) and (traverseEvent(bwd, t_pre))   //check if there is a collision in t'_pre
    {
      setFlag(t'_pre,2)
      return true
    }   //traverse forwards
    forall (t' ∈ p'• )   //check if there is a collision in t'
    {
      if ((getFlag(t') ≠ 1) and (traverseEvent(fwd, t')))
      {
        setFlag(t',2)
        return true
      }
    }
  return false
  }
}
```

The algorithm used here is based on the original as described above. The main difference is that the marked regions of places involved in a collision are identified first. Then the restriction is applied only to the slices corresponding to these marked regions, using the cube c similar to the original algorithm. This reduction results in a better performance.

Consider a place instance p'. From the definition of marked regions follows that all places which are concurrent to p', and p' itself, belong to the marked region of p'. The final state of local configuration of $\bullet(p')$ corresponds to the first reachable marking after firing $\bullet(p')$. Hence this marking is the min-cut of this slice.

Algorithm 4 Constructing the ON-set of a collision relation

```
setONset(p'_1, p'_2, Mincuts)
{
  c = C(p'_1) ⋂ C(p'_2) ≠ 0   //set cube c
  forall (node ∈ T' ⋃ P' )   //reset node flags in unfolding
    setFlag(node,0)
  setMR(p'_1, p'_2)   //set the marked region of p'_1 and p'_2
  //set first reachable min-cuts of the ON-set to Mincuts
  t'_1 = t' : •p'_1 = {t'}, t'_2 = t' : •p'_2 = {t'}   //in the unfolding a place instance has at most one incoming arc
  setMincuts((m(⇒ (t'_1)), v(⇒ (t'_1))), c, Mincuts )
  setMincuts((m(⇒ (t'_2)), v(⇒ (t'_2))), c, Mincuts )
  //remove nodes which not belong to the ON-set from the remaining parts of MR(p'_1, p'_2)
  cuts = Mincuts
  forall (p' ∈ cut)
    reduceSlice(p', c, cut_i, Mincuts)
}
```

The procedure *setONset* is presented in Algorithm 4. Its first step is to find the marked regions of the pair of collision stable places involved in a collision. This is done by setting the nodes corresponding to the slice by the function *setMR*. The second step transfers the min-cuts of the marked regions to the cuts where cube c evaluates to 1. These cuts represent the "first" min-cuts of the ON-set. During this process the nodes which do not belong to the ON-set are removed, and the min-cuts are set by the procedure *setMincuts*. In the last step, transitions which force c to reset and their postsets are removed. In the case where cube c is set again, the min-cuts are determined and the procedure *reduceSlice* is repeated. The construction of the ON-set is schematically illustrated in Figure 3.

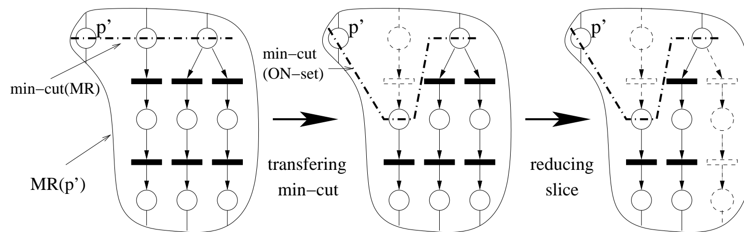


Figure 3: Deriving the ON-set

Algorithms 5 and 6 present the procedure *setMincuts* and *reduceSlice* in detail. The procedure *setMincuts* transfers a given min-cut of the marked region m' to min-cuts of the ON-set M'_{ON-set}. This procedure checks if signals differ from the cube c in the current cut m'. If this is the case, the events which differ from c are determined and are removed from the marked region using flags. In addition the nodes which are in conflict with t' are removed. Then the procedure is called recursively after the *nextCut* (the marking after firing the given transition from m') has been set by the function *setCut*. In the event of the cube c evaluating to 1, the current cut is included in $M'_{ON-set}(Mincuts)$. Note that this procedure only finds min-cuts of the ON-set which are reached from the min-cut of the marked region.

The procedure *reduceSlice* removes transitions which force c to reset, and the postsets of these transitions. The search of those transitions is started from the min-cuts of the ON-set M'_{ON-set} obtained in the previous

step and is applied to the remaining parts of marked regions MR. This procedure checks transitions which are reached from a given cut m', to determine whether they reset c. If such a transition is found, it and its postset is removed from MR and the procedure is called recursively. The case where c is set and reset several times is determined as follows. If the cube c evaluates to 1 in m', and m' cannot be reached from M'_{ON-set} due to the removal of transitions which precede m' , then m' is a min-cut of the ON-set.

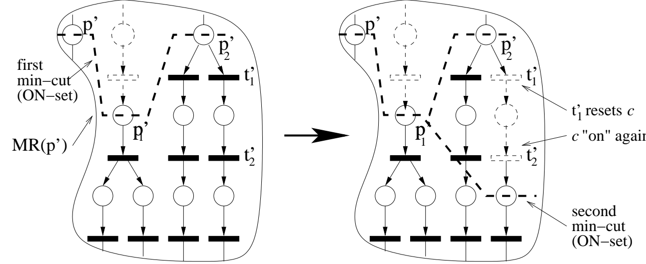


Figure 4: Reduction of a marked region by procedure *reduceSlice*

Figure 4 illustrates the reduction of a marked region by the procedure *reduceSlice*. The search of transitions which reset c starts from the min-cut $m'_{min} = \{p', p'_1, p'_2\}$ of the ON-set corresponding to the marked region $MR(p')$. Suppose transition t'_1 resets c, resulting in the removal of t'_1 and its postset from the marked region. The detecting process is repeated from the marking $m'_1 = \{p', p'_1, t'_1\bullet\}$. Imagine that the transition t'_2 sets c "on" again. Thus the cut $m'_2 = \{p', p'_1, t'_2\bullet\}$ is a min-cut of the ON-set because the successors of t'_2 cannot be reached from m'_{min}. The transition t'_2 is also removed because, until this transition has fired, c is not turned "on". The remaining marked region $MR(p')$ is the ON-set of p'.

Algorithm 5 Setting the "first" min-cuts for cube c

```
setMincuts(cut, c, Mincuts)
{
  if (c evaluates to 0 in cut)
  {  //find signals which differ from c
    forall (p' ∈ cut ) {
      forall (t' ∈ p'• ) {
        if ((t' differs from c) and (getFlag(t') == 1) )
        {
          setFlag(t',0))   //remove t' from MR
          if (t'#x : x ∈ P ∪ T)   //nodes in conflict with t'
            removeConflictingNodes   //remove conflicting nodes from MR
          if (t' is cut-off event)
            return
          nextCut = setCut(t', cut)
          setMincuts(nextCut, c, Mincuts)
        }
      }
    }
  }
  else if (cut ∉ Mincuts)   //cut is a min-cut of ON-set
    add(cut, Mincuts)
}
```

4.3 Overall procedure

The main algorithm to detect state coding conflicts by STG unfoldings is defined in Algorithm 7. First the necessary conditions are determined for every two place instances in the unfolding which are in conflict and are collision stable places. For each condition, the ON-set is set first and then its states are constructed by the function *traverseONset*. These states are checked for the existence of state coding conflicts and if any exist they are stored, otherwise a fake conflict is detected.

Algorithm 6 Reducing slice

```
reduceSlice(p', c, cut, Mincuts)
{
  if (getFlag(p') ≠ 1)    //place does not belong to the remaining part of the marked region MR
    return
  forall (t' ∈ p'•) {
    if ((getFlag(t') == 1) and (t' is enabled in MR)) {}   //t' can be enabled in MR
      if (t' ≠ cut-off event) {
        nextCut = setCut(t', cut)
        if (c evaluates to 1 in nextCut) {
          forall (p'_pre ∈ •t') {   //check if cube c "on" again
            if (getFlag(p'_pre) == 0) {
              setFlag(t',0)   //remove t' from MR
              if (nextCut∉ Mincuts )   //cut is a min-cut of ON-set
                add(cut, Mincuts)
              break
            }
          }
        }  //t' resets cube c
        else
          setFlag(t',0), setFlag(t'•,0)   //remove t' and its postset from MR
        forall (p'_post ∈ t'• )
          reduceSlice(p'_post, c, nextCut, Mincuts)
      }
      else   //cut-off event
        setFlag(t',0)   //remove t' from MR
    }
  }
}
```

Algorithm 7 Detecting state coding conflicts by STG unfolding

```
detectingConflicts(N', place cover approximations)
{
  forall (p'_i ∈ P' ) {
    forall (p'_j ∈ P' where i > j) {
      //necessary condition
      if ((C(p'_i) ⋂ C(p'_j) ≠ 0) and (isCollisionStablePlace(p'_i)) and (isCollisionStablePlace(p'_j))) {
        Mincuts = ∅
        setONset(p'_i, p'_j, Mincuts)   //setting the ON-set
        states = traverseONset(Mincuts)   //constructing the states belonging to the ON-set
        if (states in conflict)   //check of existence of conflicts
          store in data base
        else
          fake conflict
      }
    }
  }
}
```

The original algorithm [6] for the necessary condition constructs a $P' \times P'$ collision matrix which is then refined by the collision stable places. The collisions between places which are not collision stable, and any place which is not concurrent to those places, are also removed from the collision matrix. The approach used here constructs the collision stable relations "on the fly" in such a way that if a collision between places p'_1 and p'_2 exists these places must also be collision stable places. The information as to whether a place is collision stable or not is stored in order to prevent repeated checking.

The constructed states which belong to the ON-set are stored as Binary Decision Diagrams (BDDs). The package BuDDy [9] provides main functions for manipulating BDDs. The existence of state coding conflicts is subsequently determined and the output is stored.

The application of the overall procedure to detect state coding conflicts is illustrated in the example in Figure 5. Consider the collision between p'_2 and p'_5 in the STG unfolding in Figure 5(c). This collision must first be refined by the collision stable places in order to detect state coding conflicts. The procedure *isCollisionStablePlace* is applied to p'_2 and p'_5. The place p'_2 is traversed backwards first. Because its predecessor is the initial transition, no advance from it is possible , thus p'_2 is traversed forward to its direct successor. From this node the node p'_{13} is visited, because the cover approximations of p'_2 and p'_{13} do not intersect the traversal continued to $A'-$ and to p'_{14}. The place p'_{14} is a choice place. The traversal is continued in one of the choice branches first, say to $R'+, p'_4, A'+$ and p'_5. The place p'_5 is in collision with p'_2, thus p'_2 is a collision stable place. The place p'_5 is traversed backwards to $A'+, p'_4$ and $R'+$. The transition $R'+$ has two direct predecessors p'_3 and p'_{14}. These places, or their preceding places, must be in collision with p'_5. This is the case because p'_3 as well as p'_2, which is a predecessor of p'_{14}, are in collision with p'_5, thus p'_5 is also a collision stable place. The intersection of the cover approximations of the places p' [overlapping text] and ... 01100.

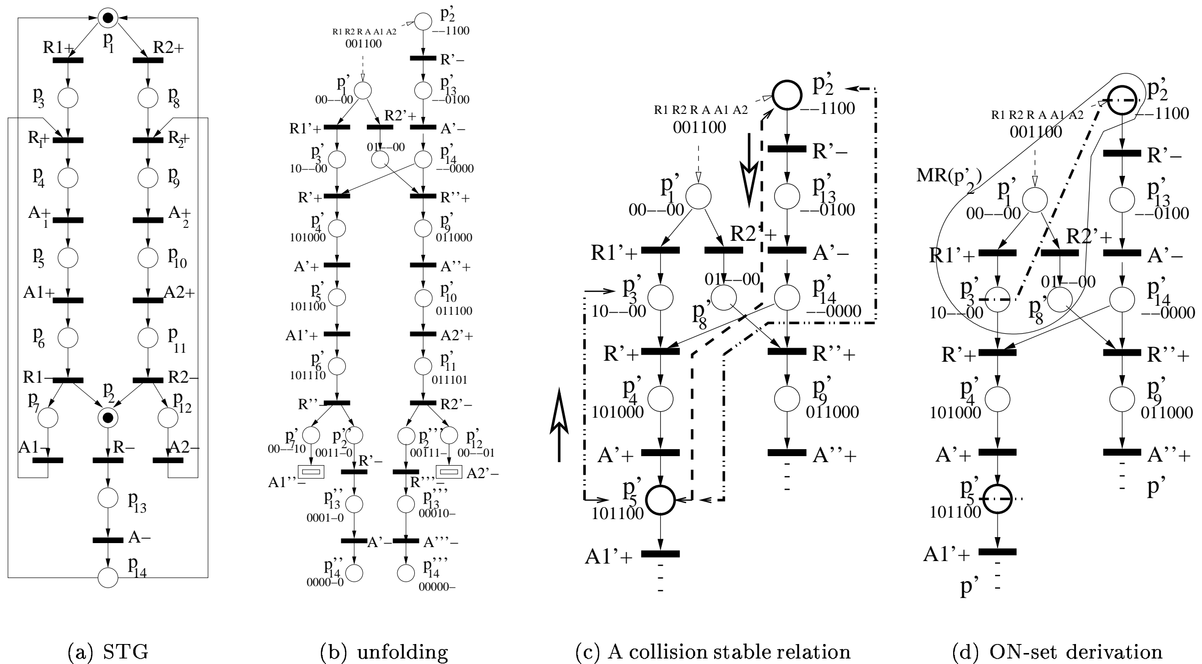


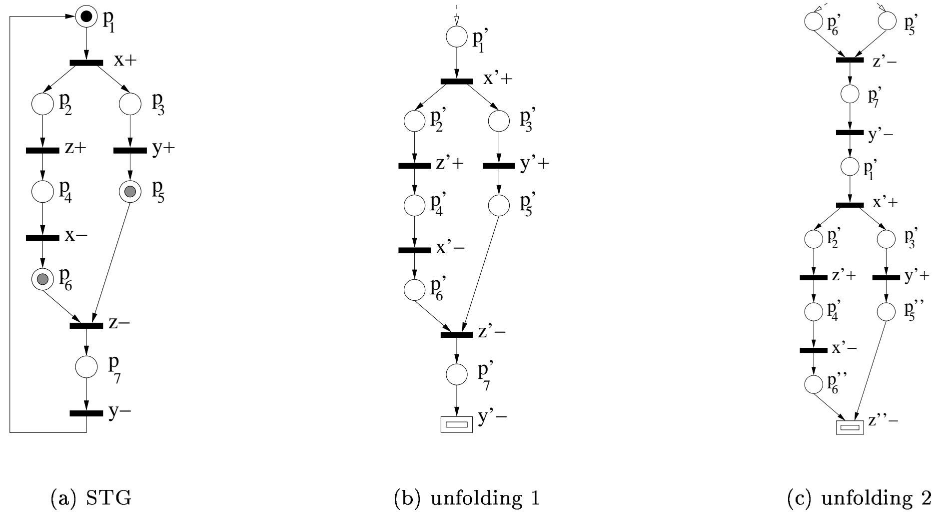
Figure 5: Detection of state coding conflicts

The ON-set derived for this collision stable relation is depicted in Figure 5(d). The marked region of a place is identified by finding its concurrent nodes. The marked region $MR(p'_2)$ includes the following nodes $\{\{p'_1 p'_2 p'_3 p'_8\}, \{R1' + R2'+\}\}$. The min-cut of this slice is the final state of the local configuration of $\bullet p'_2$, which

is the initial marking $p'_2 p'_1$. This is the first slice in which the ON-set of the cube c is constructed. In the initial marking, cube c evaluates to 0. Event $R1'+$ differentiates the binary state of $p'_2 p'_1$ from cube c. Hence the min-cut of $MR(p'_2)$ is transferred immediately after the firing of $R1'+$ to $p'_2 p'_3$, which has a binary state of 101100. In this binary state, cube c evaluates to 1 and therefore is a min-cut of the ON-set. The nodes $R2'+$ and p'_8, which are in conflict in this slice with $R1'+$ can be removed, because they belong to another slice. The remaining part of the slice belongs to the ON-set $ON(p'_2) = \{p'_2 p'_3\}$ with the corresponding binary state {101100}. The marked region of $MR(p'_5)$ includes only p'_5 (p'_5 is not concurrent to other nodes), resulting in the ON-set $ON(p'_5)$ with the binary state {101100}. It is easy to conclude that the collision between
rresponds to a state coding conflict by checking the binary states corresponding to $ON(p'_2)$

nber of fake conflicts

unfolding by transferring the initial marking can result in fewer fake collisions
ped onto the same place in the original STG (multiple instances). Consider the
t unfolding is constructed from the STG using p_1 as initial marking and the
s initial marking. It can be seen that the second unfolding contains two places
hich have multiple instances in the unfolding, resulting in additional collisions.
or p''_6, and p'_6 is in collision either with p''_5 or p''_6. A way to reduce
ppropriate initial marking. The final state of local configuration of
tial marking. A new unfolding is constructed from the new initial
g.



(a) STG (b) unfolding 1 (c) unfolding 2

Figure 6: STG unfoldings with different initial markings

Another way of reducing collisions caused by multiple instances, without having to construct a new unfolding, is to consider only the "live" part of the unfolding. Mapping the postset of the cut-off events into their images results in a "folded" unfolding. The part of the unfolding which does not belong to the folded net can be taken out of the process of detection of state coding conflicts, because this part includes markings which are already contained in the remaining net. For example, in the unfolding in the Figure 6(c) the postset of the cut-off event $z''-$ can be mapped to p'_7 and the nodes p'_5, p'_6 and $z'-$ are excluded from the detection process. The marking p'_5, p'_6 corresponds to the marking p_5, p_6 in the original STG and can be found in the remaining part of the unfolding as marking p''_5, p''_6.

The process of finding the images of the postset of cut-off events requires finding the first image of the final state of local configuration of the cut-off events in the unfolding. The predecessors of such an image can only be discarded if its preceding sub-graph is isomorphic to the one preceding the finite state of local configuration of the given cut-off event.

The software tool offers a possibility to reduce the size of the unfolding using these approaches. Note that such reductions can be only made if the STG has a home marking.

5 Experimental results

The approach described above has been applied to a set of benchmarks. A wide spectrum of benchmarks including Finite State Machines (FSMs), Marked Graphs (MGs), free-choice nets and arbiters have been chosen to explore the efficiency of this approach. The experimental results of the necessary conditions are presented, and then these are followed by the results of the refinements by partial state construction and the reduction of the size of the unfolding.

The results of the detection of state coding conflicts by the necessary condition are shown in Table 1. The number of collision relations is in most cases significantly higher than the the number of collision stable relations. This reflects the conservative estimate of the state space via place cover approximations. Benchmarks based on FSMs, e.g. alloc-outbound, rpdft and seq4, have the same number of collisions and stable collisions. This happens because their place cover approximations contain no “don't cares”.

The number of collision stable relations indicate possible state coding conflicts. It can be seen that four benchmarks have been identified as conflict free. These are FSMs and FSM dominated benchmarks. Indeed, FSMs places coincide with states and therefore place cover approximation do not contain “don't cares”. The refinement using partial state space traversal is applied to the remaining benchmarks.

benchmark	states	STG P/T	Unfolding P/T	collision relations	collision stable relations
adfast	44	15/12	15/12	70	15
alloc-outbound	21	21/22	21/22	1	1
call2	21	14/14	21/20	43	30
chu150	26	16/14	16/14	26	0
dup-4-phase-data-pull.1	169	133/123	133/123	155	35
glc	17	9/8	11/10	23	9
low-lat-nak.7.2	1552	95/85	256/202	3527	2046
master-read	2108	40/28	77/51	2002	1760
nak-pa	58	24/20	24/20	47	0
nowick	20	21/16	21/16	24	0
nrzrz-ring	86	20/18	55/42	284	53
out-arb-conv.1	74	30/26	55/42	328	71
ram-read-sbuf	39	28/22	30/23	64	7
rpdft	22	22/22	22/22	0	0
seq4	16	16/16	16/16	3	3
vbe5a	44	15/12	15/12	70	15
vmebus	24	17/17	22/22	44	31
wrdatab	216	33/24	61/42	730	636

Table 1: Necessary condition

In Table 2 the size of the traversed state space is illustrated. In the columns labelled “MR” and “ON-set”, the traversed state space relating to every collision corresponds to the marked regions and to the ON-sets, respectively. It can be observed from these percentages (to the total state space) that the size of the traversed

state space for each possible conflict is smaller than the entire state space. Furthermore, the number of states corresponding to the ON-set for each collision is in many cases smaller than the number of states corresponding to the marked regions. This reflects the reduction of the marked regions by cubes.

In the Table 3 the relation of collisions and state coding conflicts is presented. It can be seen that the number of fake conflicts is high in several cases. Benchmarks based on, or dominated by, FSM have a small number of fake conflicts, whereas highly concurrent nets have a large number of fake conflicts. This happens because place cover approximations contain many “don’t cares”, resulting in false alarms.

benchmarks	states	collision stable relations	MR states/collision		ON-set states/collision	
			average	max	average	max
adfast	44	15	31%	59%	12%	48%
alloc-outbound	21	1	10%	10%	10%	10%
call2	21	30	27%	33%	9%	14%
dup-4-phase-data-pull.1	169	35	3%	5%	1%	2%
glc	17	9	43%	47%	6%	12%
low-lat-nak.7.2	1552	2046	2%	24%	1%	6%
master-read	2108	1760	8%	78%	3%	26%
nrzrz-ring	86	53	4%	22%	3%	7%
out-arb-conv.1	74	71	4%	54%	3%	14%
ram-read-sbuf	39	7	30%	38%	0%	0%
seq4	16	3	13%	13%	13%	13%
vbe5a	44	15	31%	59%	18%	48%
vmebus	24	31	33%	38%	8%	17%
wrdatab	216	636	12%	70%	2%	39%

Table 2: Size of the traversed state space

benchmarks	states	collision stable relations	fake relations	states in USC conflicts	states in CSC conflicts
adfast	44	15	3	15	15
alloc-outbound	21	1	0	2	0
call2	21	30	26	4	4
dup-4-phase-data-pull.1	169	35	14	10	2
glc	17	9	8	2	2
low-lat-nak.7.2	1552	2046	1616	176	0
master-read	2108	1760	1760	0	0
nrzrz-ring	86	53	53	0	0
out-arb-conv.1	74	71	71	0	0
ram-read-sbuf	39	7	7	0	0
seq4	16	3	0	3	3
vbe5a	44	15	3	15	15
vmebus	24	31	25	6	6
wrdatab	216	636	636	0	0

Table 3: Number of fake conflicts

In Table 4 the relation between collisions and the size of the unfolding was examined for those benchmarks where it is possible to minimise the size of the unfolding. This table shows the reduction of collision relations obtained by transferring the initial marking (transf.) and by using the folding technique (fold.). It can be observed that, using both methods, the majority of collision stable relations are reduced in these benchmarks, but the reduction obtained by transferring the initial marking is greater.

benchmark	collision stable relations	reduction of collision	
		transf.	fold.
call2	30	63%	63%
glc	9	67%	56%
master-read	1760	64%	27%
ram-read-sbuf	7	100%	71%
vmebus	31	71%	55%
wrdatab	636	90%	58%

Table 4: Reduction of collisions

6 Conclusions

The approach to detect state coding conflicts by STG unfolding based on [6] has been implemented as a software tool. This approach uses place cover approximation and the information of maximal trees to estimate the state space, resulting in a necessary condition for state coding conflict to exists. Whilst this condition is computationally efficient it may hide so-called "fake" conflicts. Thus, a refinement technique is applied to resolve such situations at the expense of extra computational costs. This technique limits the search to those parts of the unfolding that may potentially exhibit a fake conflict. Those parts need explicit state traversing, which may be exponentially expensive.

Experiments with a wide spectrum of benchmarks show the reduction of the computational effort for the state space traversal when the necessary condition is used. The experiments can be summarised as follows. A number of benchmarks have been identified as conflict free by the use of the necessary condition only. These are FSM and FSM dominated benchmarks. Other benchmarks indicate that the size of the traversed state space for each conflict is a small fraction of the entire state space. The number of fake conflicts depends on the benchmark type. Benchmarks based on or dominated by FSM have a small number of fake conflicts, whereas highly concurrent nets have a large number of fake conflicts. This number is of the same order as the number of states. Furthermore, the relationship of collisions to the size of the unfolding has been examined and this shows a high reduction of collisions in several benchmarks.

The high incidence of fake conflicts results in a long processing time, even when the size of the traversed partial state space is small. This can be overcome by distributing the task to a net of computers, running the state space traversal for each collision relation identified by the necessary condition and checked independently in a separate computer.

References

[1] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, A. Yakovlev: Petrify: a tool for manipulation concurrent specifications and synthesis of asynchronous controllers, IEICE Transactions on Information and Systems, Vol. E80-D, No. 3, pp. 315-325, March 1997.

[2] K. L. McMillan: Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits, in Bochmann, G.v. Probst, D.K. eds., Computer Aided Verification. Fourth International Workshop, CAV '92, Montreal, Que., Canada, pp. 164-77, 1992.

[3] J. Esparza, S. Römer, W. Vogel: An Improvement of McMillan's Unfolding Algorithm. In T. Margaria, B. Steffen, Proc. of TACAS'96, Vol. 1055 in Lecture Nodes in Computer Science, pages 87-106. Springer-Verlag, 1996.

[4] A. Semenov: Verification and Synthesis of Asynchronous Control Circuits Using Petri Net Unfoldings, PhD thesis, University of Newcastle upon Tyne, Department of Computing Science, July 1997.

[5] A. Madalinski, A. Bystrov, A. Yakovlev: A software tool for State Coding Conflict Detection by Partial Order Techniques, extended abstract, 1st UK ACM SIGDA Workshop on Design Automation, 10th September 2001.

[6] A. Kondratyev, J. Cortadella, M. Kishinevsky, L. Lavagno, A. Taubin, A. Yakovlev: Identifying State Coding Conflicts in Asynchronous System Specifications Using Petri Nets Unfolding. International Conference on Application of Concurrency to System Design, pages 152-163, March 1998.

[7] A. Semenov, A. Yakovlev: Event-based Framework for Verifying High-Level Models of Asynchronous Circuits. Technical Report 487, University of Newcastle upon Tyne, May 1994.

[8] Change-Hee Hwang, Donk-Ik Lee: A Concurrency Characteristic in Petri Net Unfolding. IEICE Trans. Fundamentals, Vol. E81-A, No.4, April 1998.

[9] J. Lind-Nielsen: BuDDy: Binary Decision Diagram package Release 1.9, IT-University of Copenhagen (ITU), August 2000.