Visualisation of coding conflicts in asynchronous circuit design*

A. Madalinski, A. Bystrov and A. Yakovlev
Department of Computing Science, University of Newcastle upon Tyne
{A.A.Madalinski, A.Bystrov, Alex.Yakovlev}@ncl.ac.uk

30th May 2002

Abstract

Synthesis of asynchronous circuits from Signal Transition Graphs (STGs) involves solving state coding conflicts. The refinement process is generally done automatically using heuristics and can often produce a sub-optimal solution, which has to be corrected manually by the designer. This paper presents a framework for an interactive refinement process aimed to help the designer. Rather than using actual conflicts, the process is based on the visualisation of conflict cores. Cores are sets of transitions causing state coding conflicts, which are represented at the level of a finite complete prefix (unfolding) of STGs.

1 Introduction

The fundamental problem in the synthesis of asynchronous circuits from their specifications as Signal Transition Graphs (STGs), which are Petri Nets whose events are interpreted with signal transitions of a modelled circuit, is the problem of Complete State Coding (CSC). This problem arises when a pair of semantically different states has the same binary encoding, which results in a CSC conflict. To resolve this problem, new signals must be inserted into the specification in such a way that the behaviour of the transformed specification remains externally equivalent to the original one. The values of these new signals have to be different in each pair of states involved in a CSC conflict.

There are a number of methods for detecting and solving the CSC problem (for a brief review see [2]). The techniques in [3] and [4] concentrate on the introduction of constraints within an STG, using coupledness relation and lock relation, respectively, as a guidance. Both relations recognise that if all pairs of signals in the STG are "locked" using a chain of handshaking pairs, then the STG is conflict free. The synthesis tool Petrify [1] uses the theory of regions [2] for the insertion of new signals into a specification to solve state coding conflicts.

These techniques work well. However, they may produce sub-optimal solutions or sometimes they fail to solve the state coding conflicts. Manual design often yields a more compact and elegant solution. Thus a synthesis tool should offer a way for the designer to understand the characteristic patterns of the circuits behaviour in order to manipulate the model more interactively.

The visualisation method presented in this paper is aimed to facilitate a manual refinement of STGs with a CSC problem. It is based on the relation conditions in [3] and [4]. Conflicts are visualised by cores, which represent sets of transitions involved in one or more conflicts, thus avoiding the enumeration of each coding conflict explicitly. This refinement technique contains several interactive steps to resolve coding conflicts manually.

Conflict cores are visualised at the level of the STG unfolding, which offers a more convenient form for understanding the behaviour of the system. An STG unfolding is a branching process

*This work is supported by EPSRC grant GR/M94366

generated by an STG partial order execution, which is constructed for the given initial marking. Its finite complete prefix, a truncated unfolding, contains all reachable markings of the STG. The size of the finite complete prefix is typically larger than the STG and significantly smaller that its state space.

In order to eliminate state coding conflicts by adding auxiliary signals one must know where in the specification to insert a new signal, and how to insert a new signal. The former affects the quality of the implementation and the latter affects its correctness. Solving state coding conflicts manually requires that the designer understands the behaviour of the specification and the cause of the CSC problem.

The identification of an appropriate "area" to insert a signal in the specification is the basic operation of the transformation of the initial STG into an STG satisfying the CSC property. The visualisation method described here addresses this problem. It represents such "areas" by means of conflict cores at the level of STG unfolding. It offers a global representation of the CSC problem, which shows the distribution of conflicts in the specification.

Using this representation the designer can navigate through the specification. In addition, the designer can select a particular location where conflicts occur and obtain a local, more detailed description of the selected sub-graph. The visualisation enables the designer to understand the problem, and thus it helps to resolve these conflict cores individually, satisfying the given constraints of the design. A conflict core can be eliminated by separating the set of transitions belonging to it by inserting an auxiliary transition and by adding its complement to the remaining part of the STG.

The process of solving CSC conflicts, by analysing and solving the visualised conflict cores, may be repeated until the transformed STG satisfies the CSC property. Reducing the number of conflict cores results eventually in the elimination of all CSC conflicts. Currently this visualisation method is applied to a sub-class of Petri Nets, Marked Graphs [8] with multiple transitions. Figure 1 shows the location of the visualisation of state coding conflicts in the design process of asynchronous circuits.

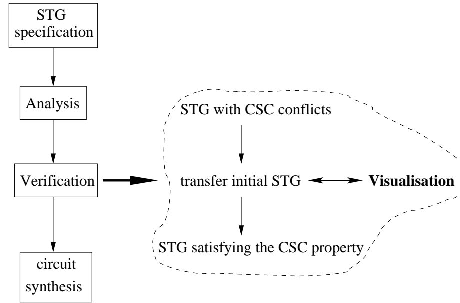


Figure 1: Location of visualisation in the design process

This paper is organised as follows. The basic theoretical background concerning STGs and their unfoldings is provide in section two. In section three the concept of visualisation of state coding conflicts based on cores is described. Section four shows an approach to manual STG transformation based on core visualisation to ensure the CSC requirement. A case study which emphasises the features of the manual transformation process is presented in section five. The conclusions are discussed in section six.

2 Basic notions

This section introduces the basic definitions and notations used in this paper. These include models, such as Signal Transition Graphs (STGs) and STG unfoldings, target properties, such

as Complete State Coding (CSC) and important notions supporting the visualisation method.

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 $\lambda : T \rightarrow A \times \{+, -\}$ 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. In the graphical representation of STGs places with one input and one output transition (implicit places) are depicted as an arc connecting these two transitions.

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 $t \bullet \neq 0$.

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 $\gamma : E \rightarrow A \times \{+, -\}$ 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 $s \xrightarrow{e} s'$ 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. Logic equations are derived for the non-input signals. Input signals are assumed to be generated by the environment. Thus 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.

In an STG a *complementary set* $B \subseteq T$ is defined such that $\forall t \in T : x = t \subset B \Leftrightarrow \overline{t} \in B$. B contains both the rising and falling transitions. A set of transitions $D \subseteq T$ in a live STG is *feasible* iff there exists a state from which the transitions in D can be fired without firing a transition not belonging to D. The sequence in which these transitions are fired are irrelevant. An STG has two states with the same coding iff $\exists B \subset T$, which is a feasible complementary set. By firing a complementary set, the binary encoding of the states before and after the firing of

the set are the same. This is because if any transition t_i in B occurs, then $\overline{t_i}$ must also have occurred. The states, however, are semantically different resulting in a CSC or USC conflict depending on the enabled transitions.

An STG is implementable as a *speed-independent* circuit iff its SG is finite, consistent, output-persistent and satisfies CSC. A circuit is speed-independent if its functional behaviour does not depend on the delay of its gates.

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 $T' \cup P'$ as an instance of elements of $T \cup P$. 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 $\bullet t'_1 \cap \bullet t'_2 \neq 0$.
- *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 [5][6], 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.

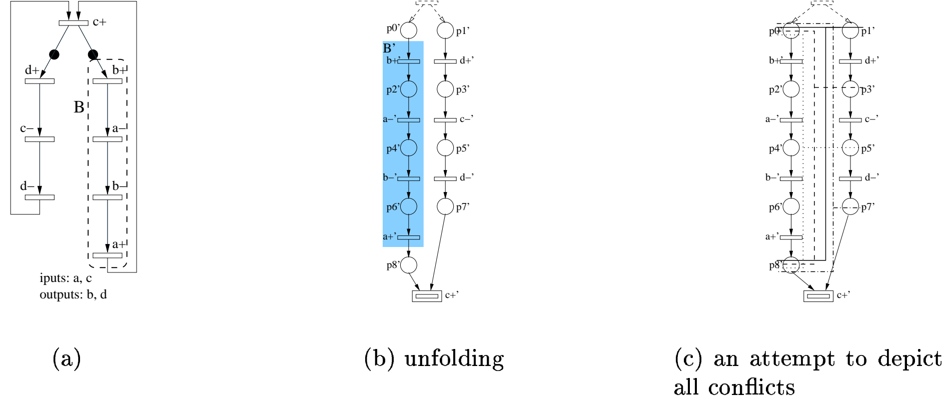
The set of causal predecessor transitions of t' of the STG unfolding is called the *local configuration* of t' , denoted as $\Rightarrow t'$. A set of place instances reached by firing all transitions in $\Rightarrow t'$ is called the postset of $\Rightarrow t'$, denoted as $(\Rightarrow t')\bullet$. 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 [5][6], 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.

3 Visualisation of coding conflicts

In this section, the visualisation of state coding conflicts in asynchronous circuits design is presented. The representation of all possible conflicts is not efficient as illustrated in Figure 2(c), where the conflict pairs $\langle\{p0', p1'\}, \{p8', p1'\}\rangle$, $\langle\{p0', p3'\}, \{p8', p3'\}\rangle$, $\langle\{p0', p5'\}, \{p8', p5'\}\rangle$ and $\langle\{p0', p7'\}, \{p8', p7'\}\rangle$, which corresponding to pairs of conflicting states, are presented. It can be seen that already a small number of conflicts is difficult to depict. Visualising only the essential parts involved in conflicts (Figure 2 (b)) offers a much more elegant solution, which avoids the explicit representation of conflicts.



(a) (b) unfolding (c) an attempt to depict all conflicts

Figure 2: Visualisation of conflicts

The visualisation is based on complementary sets which are represented at the level of the STG unfolding. Since every element in an STG unfolding is an instance of an element in the STG, the complementary sets can be mapped easily from STGs to their unfoldings and vice versa. Figure 2(a) shows an STG with a CSC problem. The CSC problem is caused by the feasible complementary set $B = \{b+, a-, b-, a+\}$, which can be mapped to the events of the unfolding (Figure 2(b)) corresponding to the transitions in B, resulting in $B' = \{b+', a-', b-', a+'\}$. Note that in further representations of complementary sets at the STG unfoldings, they will be referred to as the transitions of the STG rather than as their instances unless it creates confusion.

Definition 1. A feasible complementary set K is called *composite* iff it is an union of complementary sets $B = \{B_1, B_2, ...B_n\}$ such that $B_i \cap B_j = \emptyset$, $\bigcup_{i=1,n} B_i \subseteq K$ and $K \subseteq \bigcup_{i=1,n} B_i$, where n is a positive integer.

The example in Figure 3(a) has several CSC conflicts caused by three complementary sets $B_1 = \{Dr+, Da+, Dr-, Da-\}$, $B_2 = \{Zr+, Za+, Zr-, Za-\}$ and $B_3 = B_1 \cup B_2$. The set B_3 is a composite complementary set, because it is a union of B_1 and B_2, which are non-intersecting sets. Note that for simplicity the implicit conditions of the unfolding are not represented.

Definition 2. A *conflict core* (or simply core) is a feasible complementary set which is not a composite complementary set.

The complementary sets B in Figure 2(b) and B_1 and B_2 in Figure 3(a) are conflict cores. Conflict cores are important in the elimination of state coding conflicts. This process requires the introduction of additional internal signals to resolve conflicts by splitting the cores. It is described in detail in the next section. In this process the composite complementary sets do not have to be considered, because they completely consist of other cores involved in the solving process. Eliminating those cores results in eliminating the composite complementary sets.

(a)

(b)

Figure 3: Visualisation of cores

Definition 3. The complement of a core B, denoted by $\overline{B}$, is the complement of B w.r.t. T, i.e. $T \setminus B$.

The set $\{ai+, ai-\}$ is a complement of the core B_2 in Figure 3(b). Since the set of transitions in the STG is a complementary set, the complement of a core is also a complementary set, but not necessarily a core.

Definition 4. A *core cluster* is a set of cores that form a connected sub-graph of the unfolding prefix.

A core cluster $Z = \{\{ri-, ri+\}, \{ri-, ro-, ri+, ro+\}\}$ is illustrated in Figure 3(b). In the solving process core clusters are used for the visualisation of conflicts. A core cluster can be eliminated by at least one additional signal, and is therefore a useful representation in the solving process.

4 Solving coding conflicts

In this section the solving process based on the visualisation of cores is described. The process contains several interactive steps which are aimed to facilitate the designer to resolve the coding conflicts manually. An intuitive description of the solving process is presented followed by the elimination of conflict cores by transition insertion, and by an example to illustrate the solving process.

Solving process The overview of the solving process is shown in Figure 4. Given an STG with CSC conflicts, its unfolding (finite complete prefix) is constructed. This model is used to visualise conflict cores. Phases one and two enable the designer to determine a location where an auxiliary transition and its complement, respectively, can be added. After resolving a conflict core, or a core cluster, by an insertion, the designer can either continue the process by repeating phases one and two, or by transferring the inserted transitions to the STG to check if conflicts are eliminated. The former requires an updating of the visualised cores, i.e. the cores which have been eliminated in the previous step are removed from the representation. Typically, the first option can be selected if the eliminated core or core cluster is non-overlapping with some other core. Depending on the CSC problem, the solving process can consist of several cycles. Solving CSC at the level of cores does not guarantee total CSC but it probably guarantees some progress.

The insertions in phases one and two are similar and contain the following steps. First, a "height map" is presented, which shows a quantitative distribution of cores. The events of the unfolding which correspond to conflict cores are highlighted by shades of colours. The shade

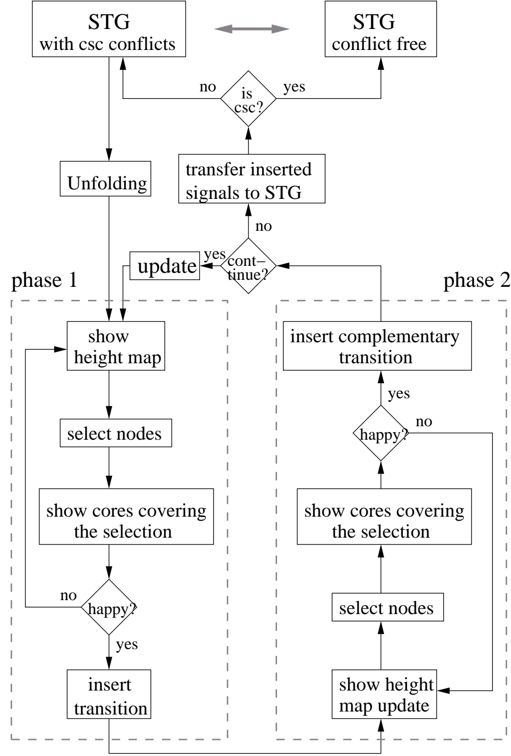


Figure 4: The process of coding conflict solving by cores

depends on the weight of an event, which corresponds to the number of cores containing an event. The greater the weight the darker the shade. The conflict height map is equivalent to a geographical map showing the different altitudes.

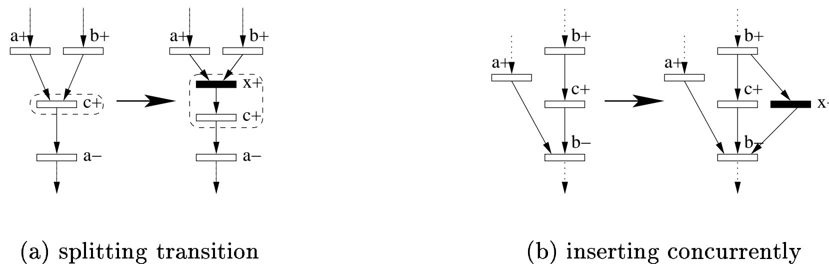
The purpose of the height map is to depict a global representation of conflicts, so that the designer understands how the conflicts are distributed in the specification. From this representation, the designer can select particular events which are involved in a conflict, to obtain a local, more detailed description of a core cluster (the cores corresponding to the selection). This can be repeated until an appropriate core cluster for transition insertion is found. The designer can now decide where and how to insert a new signal transition.

The difference between the two phases is that the visualisation of cores in phase two depends on the inserted transition t in phase one. The core cluster eliminated by the insertion is removed and its complementary representation is added to the set of cores. Furthermore, the events which are concurrent to t are faded out from the representation. This ensures that the insertion of the complement of t in phase two is not added concurrent to t, preserving consistency.

Elimination of conflict cores State coding conflicts can be efficiently resolved by introducing "additional memory" to the system in the form of internal signals. This approach requires preserving behavioural equivalence of the specification and guaranteeing that the signals are implemented without hazards, preserving speed independence.

In order to modify the conflicting states, caused by a conflict core B, into distinguishable binary states, one needs to add an internal signal x into the specification. The core B can be destroyed by inserting a transition of x, say $x+$, anywhere in B ensuring that it is no longer a complementary set. To preserve consistency $x-$ is also added to the specification in such a way that $x-$ is not concurrent to $x+$.

A transition can be inserted to an STG in two ways. One way is to split an existing transition and insert a internal transition before or after it. The inserted transition cannot point to a



(a) splitting transition (b) inserting concurrently

Figure 5: Transition insertion

transition of an input signals because the circuit may not impose constraints on the environment. The Figure 5(a) shows the splitting of the non-input transition $c+$ by the insertion of the internal transition $x+$.

The other way is to insert a transition concurrent to an existing transition in such a way that the outgoing arc of the inserted transition is connected to a non-input transition. This is illustrated in the example in Figure 5(b), where $x+$ is added concurrently to the input transition $c+$ between the non-input transitions $b+$ and $b-$. The inserted transition $x+$ is executed concurrently with the environment operation, resulting in reduced latency of the circuit.

An example The example in Figure 6 illustrates the process of solving CSC conflicts using the visualisation method described above. Starting from the initial STG, which does not satisfy the CSC property, an unfolding is constructed. The unfolding is used as a model for the visualisation process.

The height map in phase one shows the distribution of conflict cores. Every event is associated with a weight. The events with weight "W2" indicate that at least two cores cause the conflicts. To obtain a clearer representation in this example, only those events with the highest weight are selected. It can be seen that the conflicts are caused by the core cluster Z containing two cores $B_1 = \{ri-, ri+\}$ and $B_2 = \{ri-, ro-, ri+, ro+\}$. Note that $ri-$ has two instances in the unfolding $ri-'$ and $ri-''$.

A way to eliminate Z is to separate both cores by adding a transition $csc+$, say by splitting $ri+'$. In the next phase $csc-$ is inserted using a new height map, which is updated as follows. The cluster Z is removed from the representation and the complement of the cores in Z, $\overline{B_1} = \{ro-, ai-, ro+, ai+\}$ and $\overline{B_2} = \{ai-, ai+\}$, is added. The nodes which are concurrent to $csc+$ are faded out. Consider the detailed description of the updated representation, which shows the complement of the core Z without taking $ai-'$ to account, which is concurrent to $csc+$. Splitting $ai+'$ would result in destroying B_1 and B_2. Thus $csc-$ is inserted just before $ai+'$. In the case of another location being preferred for the insertion, e.g. this would result in repeating the solving process, because not all complementary sets would be destroyed. The inserted signal csc is transferred to the STG resulting in a new STG satisfying the CSC property.

5 Case study

In this section three interesting examples will be discussed to demonstrate the proposed solving process based on visualisation of conflict cores. In addition, the STGs derived by this process will be compared against STGs derived by the tool Petrify.

Imec-nowick The initial STG of a benchmark "imec-nowick" is given in Figure 7(a). After constructing the unfolding from the given STG the solving process begins. Figure 8(a) illustrates

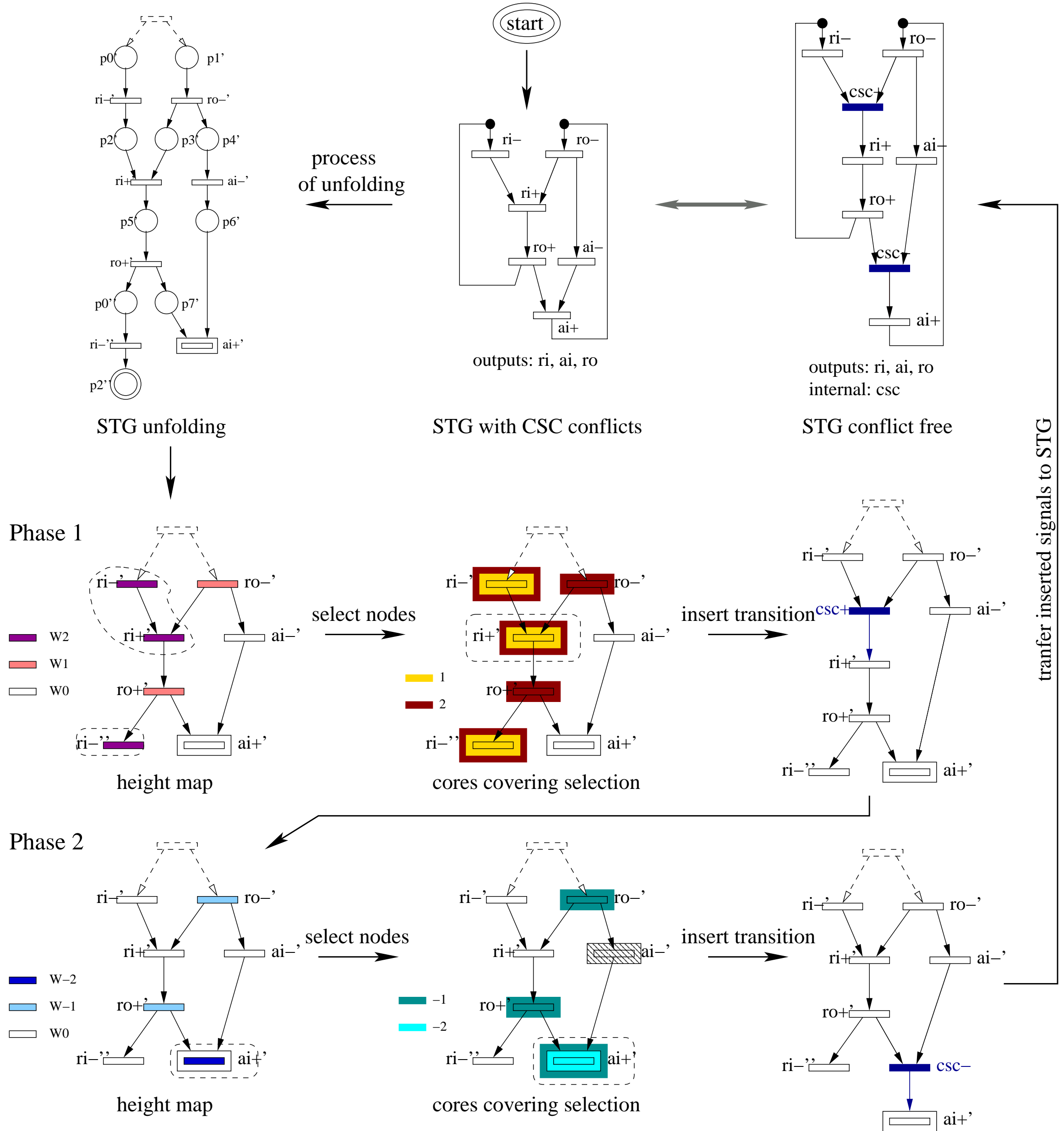
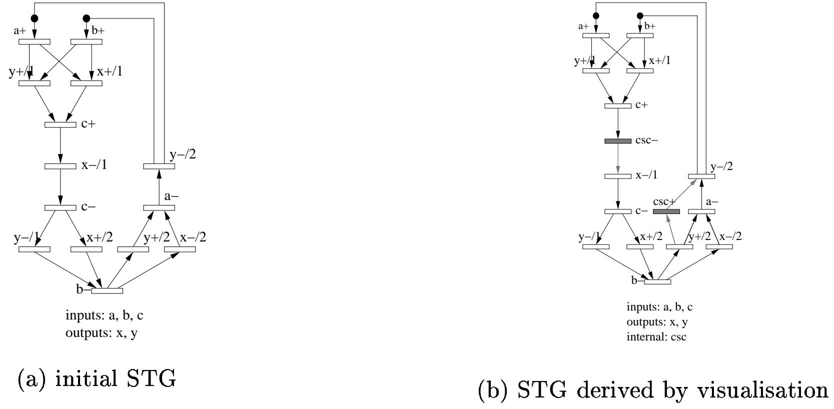


Figure 6: Solving coding conflict by cores an example

phase one, the insertion of the first transition. The height map depicts the distribution of cores and it can be seen that the events $c+'$, $x-'$ and $c-'$ have the highest weight (W3). The selection of these events results in the core cluster Z containing five conflict cores (1-5). These cores are nested or overlapping, e.g core 4 is nested in core 1 and cores 2 and 3 overlapping. The obvious way to eliminate Z is to insert a transition where all cores intersect. Because c is an input signal, only the event $x-'$ can be split and an internal transition, say $csc-$, can be inserted before $x-'$.

The second phase, inserting $csc+$, is illustrated in Figure 8(b). The updated height map contains the complement of Z. The weights and the complementary cores of Z are represented by negative values, which correspond to the weight and cores in phase one, e.g. the core -1 is the complement of the core 1.

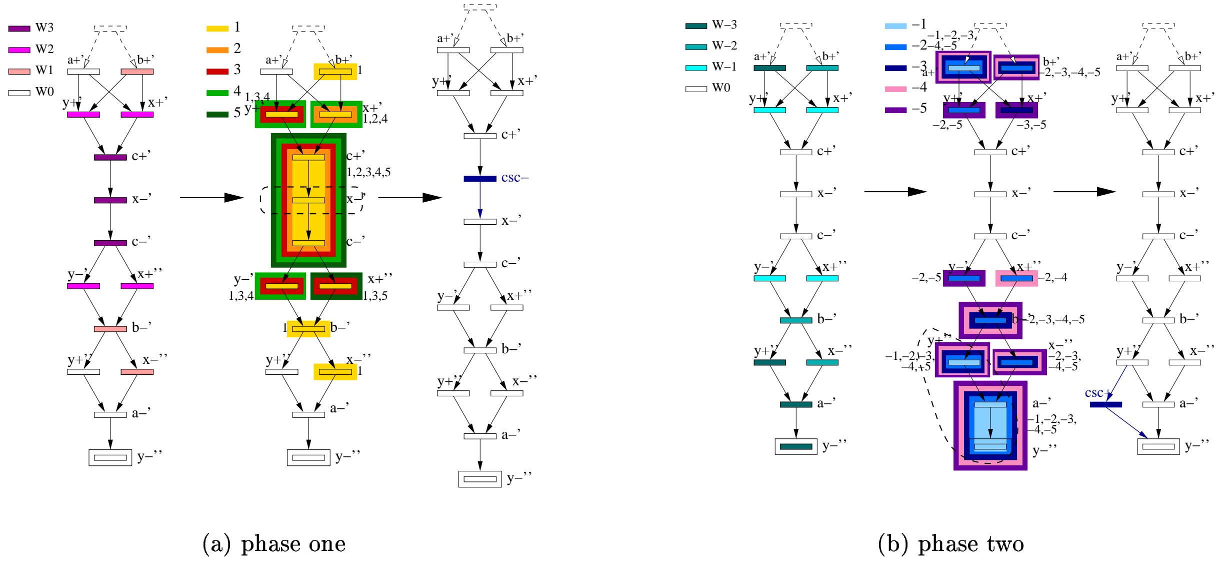
In order to eliminate Z, $csc+$ can only be inserted between the events with the highest negative weight (W-3), which represents the intersection of the complementary cores in Z. One



(a) initial STG

(b) STG derived by visualisation

Figure 7: STG transformation



(a) phase one

(b) phase two

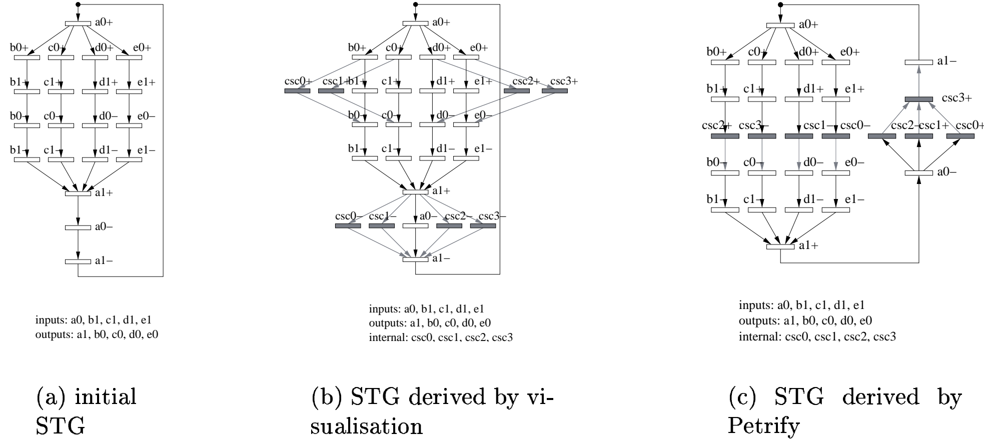
Figure 8: Solving CSC conflicts

way is to insert $csc+$ concurrently with $a-'$ between $y+''$ and $y-''$ or another option is to split either $y+''$ or $y-''$. The former is chosen because it gives a better performance. The inserted transitions are transferred to the STG resulting in an STG satisfying CSC, which is depicted in 7(b).

Petrify derives a similar STG. It also uses one signal to resolve the CSC problem. One transition is inserted before $x-/1$ and its complement is inserted before $y-/2$.

A handshake decoupling element In Figure 9(a) the initial STG of a handshake decoupling element is presented. It has four handshakes which are concurrent with each other, resulting in a large number of CSC conflicts. In fact, there are 888 CSC conflicts and 15 complementary sets, of which 11 are composite sets and 4 are conflict cores.

The solving process starts with the examination of the height map in phase one. The weights on the height map indicate only one weight (W1), which gives no information about the number of cores involved in the CSC problem. Only selecting the events with this weight results in the representation of the cores (1-4). These cores are concurrent with each other and can be eliminated independently by adding four signals. In this example core 1 is eliminated first. This



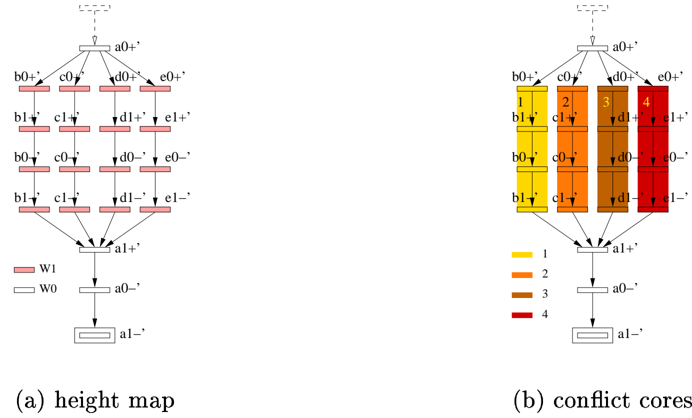
(a) initial STG

(b) STG derived by visualisation

(c) STG derived by Petrify

Figure 9: STG transformation

step is illustrated in Figure 11(a). The internal transition $csc0+$ is inserted concurrent to $b1+'$ between the output transitions $b0+'$ and $b0-'$. $csc0-$ cannot be inserted concurrently with $csc0+$. Thus the representation of the complementary core -1 does not include the concurrent part. $csc0-$ should be added to the core -1, e.g. inserted concurrent to $a0-'$. It was chosen to add these transitions concurrently to reduce the latency.



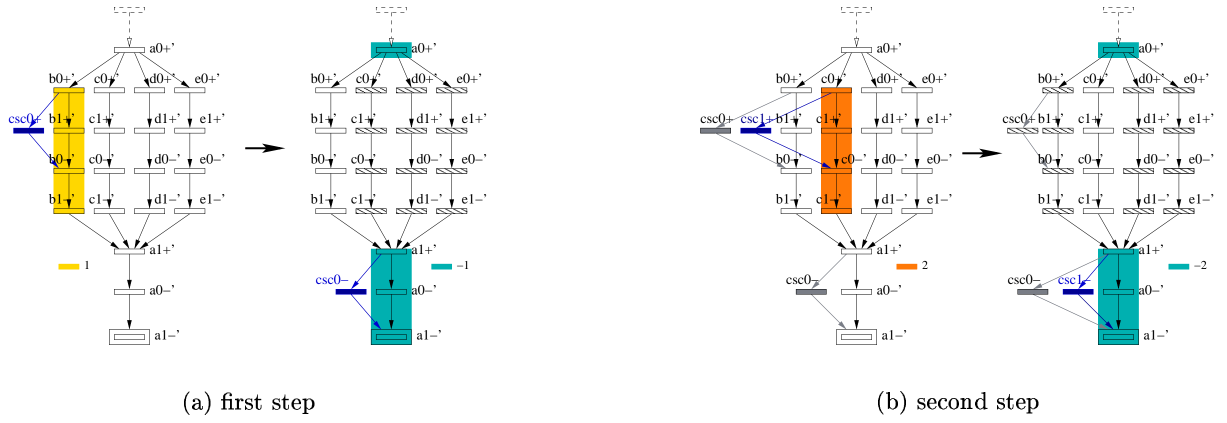
(a) height map

(b) conflict cores

Figure 10: Solving process (part one)

Because the cores are independent, the process can be repeated without constructing the transformed STG. Figure 11(b) shows the second step, the elimination of core 2. The transition $csc1+$ is inserted concurrent to $c1+'$. $csc1-$ is inserted in the same way as in the first step. Cores 3 and 4 are eliminated similarly to cores 1 and 2. The transformed STG is presented in Figure 9(b). This solution produces four fan-ins and fan-outs of the signal $a1$, which may not be desired. To avoid this, the negative internal transition ($csc0-$- $csc3-$) could be partitioned twice in two transitions which are connected in serial, and which are inserted concurrently with $a0-$.

Figure 9(c) shows the STG derived by the tool Petrify. This solution also uses four signals to resolve the CSC problem. It can be seen that the inserted transitions in the concurrent part are included before an output, increasing the latency of the corresponding output.

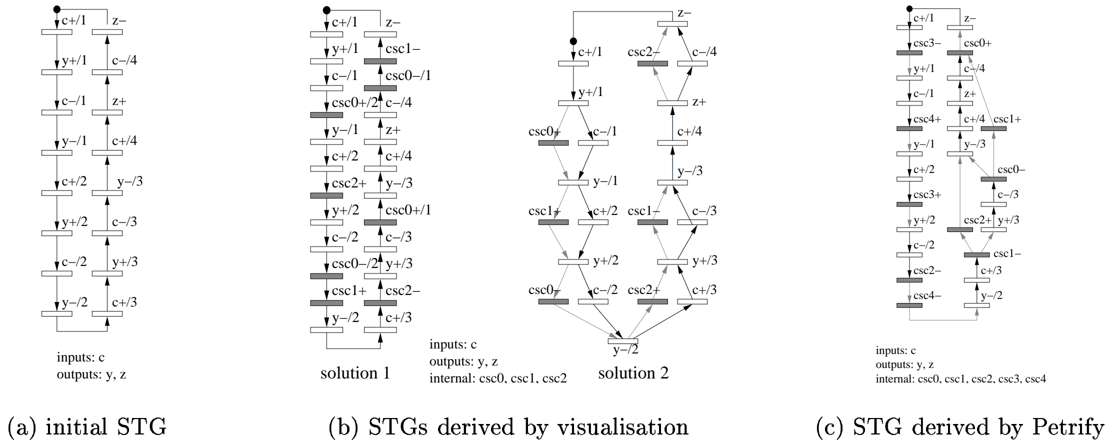


(a) first step

(b) second step

Figure 11: Solving process (part two)

Modulo four counter The initial STG of a modulo four counter is depicted in Figure 12(a). It contains several CSC conflicts and can be solved in many way, e.g. as shown in Figure 12(b). These solutions have been obtained by the visualisation method. In the first solution the additional signals are connected in serial, whereas in the second solution the signals are connected concurrently. One can also obtain a solution which is a combination of both, tailored to the requirements of the design. Both solutions use three additional signals to solve the CSC problem. Petrify, however uses five additional signals as shown in Figure 12(c).



(a) initial STG

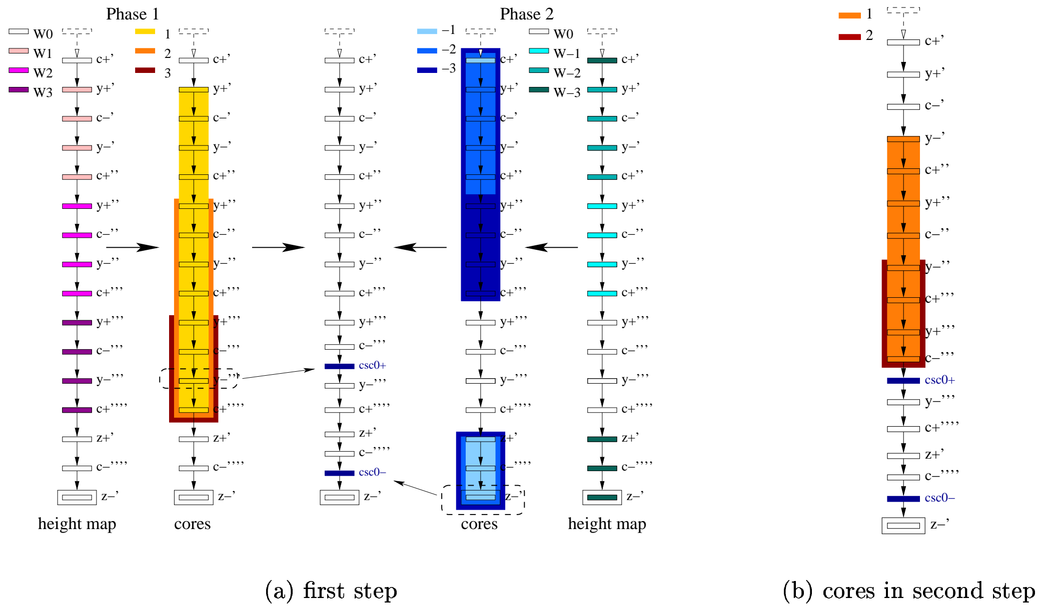
(b) STGs derived by visualisation

(c) STG derived by Petrify

Figure 12: STG transformation

The first solution in Figure 12(b) and the solution in 12(c) have sequences of several inserted signals which are placed before an output, thus significantly increasing the latency of the corresponding output. The solution in 12(b) could be improved by splitting the sequence *csc0-/2→csc1+* and *csc1-→csc0-/1* and inserting these transitions concurrently.

Figure 13 indicates a possible initial step in the solving process. The first step is illustrated in Figure 13(a). From the height map in phase one the events with the highest weights are selected, resulting in a core cluster containing three cores (1-3). One way to eliminate this cluster is to insert a transition, say $csc0+$, to a location where all cores intersect, e.g. $csc0+$ is inserted before $y-'''$. In the second phase a location is found for the insertion of $csc0-$. The transition $csc0-$ is inserted before $z-'$, which is the location where the complementary cores -1 to -3 intersect, resulting in the elimination of cores 1-3. After transferring the inserted transitions to the STG,



(a) first step

(b) cores in second step

Figure 13: Part of a solving process

the STG still does not satisfy the CSC property, even though all cores have been eliminated. This happens because after the insertion some USC conflicts become CSC conflicts. These new conflicts are depicted in Figure 13(b). The solving process has to be continued until the CSC property is satisfied.

6 Conclusions and future work

In this paper a framework has been presented for the interactive insertion of signals into STGs to satisfy the CSC (or USC) requirement. It is based on the visualisation of conflict cores, which are sets of transitions involved in state coding conflicts, thus avoiding the explicit representation of conflicts at the level of states pairs. Cores are represented at the level of the STG unfolding, which is a convenient model for understanding the behaviour of the system due to its simple branching structure and acyclicity.

The signal insertion is performed in several interactive steps aimed to help the designer to obtain a customised solution. The steps include a global representation of conflict cores, in the form of a height map, from which a local, more detailed description of cores can be obtained. This helps to locate the areas in the unfolding prefix which containing the most of the conflicts, and to visualise the causes of these conflicts. Solving conflicts requires the elimination of cores by the introduction of additional transitions into the system.

The case studies demonstrate the positive features of the interactive refinement process. The extension of the current work will be to visualise STG which include conditional behaviour. Furthermore, a software tool is being developed to implement the described visualisation technique.

The conflict solving procedure can be automated either partially or completely. Heuristics could use the highest weights of the height map for signal insertion. The advantage of using cores is that only those parts which cause conflicts are considered rather than the complete set of conflicts as in e.g. Petrify [1]. The former set is usually smaller than the latter. Thus, it is expected that the efficiency of heuristics will be improved because they will use the most essential information about conflicts.

To obtain an optimal solution a semi-automated solving process could be employed. Heuris-

tics would provide the designer with possible locations for the insertions, which could be used as guidelines. However, the designer is free to choose an alternative location at any time to take into account the design constraints.

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] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, A.Yakovlev: A Region-Based Theory for State Assignment in Speed-Independent Circuits, IEEE Transactions on Computer-Aided Design of Integrated circuits and Systems, Vol. 16. No. 8, August 1997.

[3] A. Yakovlev, A. Petrov: Petri Nets and asynchronous bus controller design, in Proc. 11th Int. Conference on Application and Theory of Petri Nets, Paris, France, pp 244-262, June 1990.

[4] P. Vanbekbergen, G. Goossens, F. Catthoor, H. J. De Man: Optimized Synthesis of Asynchronous Control Circuits from Graph-Theoretic Specification, IEEE Transactions on Computer-Aided Design, Vol. 11, No. 11, November 1992.

[5] 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.

[6] 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.

[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] T. Murata. Petri nets: Properties, analysis and applications. Proceedings of the IEEE, Vol. 77, pp.541-580, April 1989.