Visualization and Resolution of Coding Conflicts in Asynchronous Circuit Design*

Agnes Madalinski[1], Alex Bystrov[1], Victor Khomenko[2], and Alex Yakovlev[1]
[1]School of Electrical, Electronic and Computer Engineering
[2]School of Computing Science
University of Newcastle upon Tyne, NE1 7RU, UK

Abstract

Synthesis of asynchronous circuits from Signal Transition Graphs (STGs) involves resolving state coding conflicts. The refinement process is generally done automatically using heuristics and often produces sub-optimal solutions, which have to be corrected manually. This paper presents a framework for an interactive refinement process aimed to help the designer. It is based on the visualization of conflict cores, *i.e., sets of transitions causing coding conflicts, which are represented at the level of finite and complete prefixes of STG unfoldings.*

1. Introduction

Signal Transition Graphs, or STGs (see, e.g., [1]), are widely used for specifying the behaviour of asynchronous control circuits. STGs are interpreted Petri nets in which transitions are labelled with the rising and falling edges of circuit signals. Synthesis based on STGs involves the following steps: (a) checking the necessary and sufficient conditions for an STG's implementability as a logic circuit; (b) modifying, if necessary, the initial STG to make it implementable; and (c) finding appropriate boolean next-state functions for output and internal signals.

One commonly used tool, PETRIFY [2], performs all these steps automatically, after first constructing the reachability graph of the initial STG specification. To gain efficiency, it uses symbolic (BDD-based) techniques to represent the STG's reachable state space.

While such an approach is convenient for completely automatic synthesis, it has several flaws: state graphs represented explicitly or in the form of BDDs are not visual, and thus prevent efficient interaction with the user. Moreover, the combinatorial explosion of the state space is a serious issue for highly concurrent STGs (e.g., generated from high-level hardware descriptions). This makes alternative techniques, and in particular those based on Petri net unfoldings, very attractive for this task. In [5] the unfolding technique was applied to the implementability analysis in step (a), viz. checking the Complete State Coding (CSC) condition [1], which requires detecting coding conflicts between an STG's states. Since STGs usually exhibit a lot of concurrency, but rather few choice points, their unfolding prefixes are often exponentially smaller than the corresponding state graphs; in fact, in most of the experiments conducted in [5] they are just slightly bigger then the original STGs themselves. Therefore, unfolding prefixes are well-suited for both visualization of an STG's behaviour and alleviating the state space explosion problem.

In this paper, we concentrate on step (b), in particular on enforcing the CSC condition (i.e., on resolving CSC conflicts), which is a necessary condition of the implementability of an STG as a circuit. A CSC conflict arises when semantically different states of an STG have the same binary encoding. To resolve it, new signals, helping to distinguish between these states, must be inserted into the specification in such a way that the behaviour of the transformed specification remains externally equivalent to the original one. (Intuitively, insertion of signals introduces additional memory to the circuit, helping to trace the current state.)

A number of methods for resolving CSC conflicts have been proposed so far (see, e.g., [4] for a brief review). The techniques in [9, 10] concentrate on the introduction of constraints within an STG, using *lock relation* and *coupledness relation*, respectively, as a guidance. Both relations recognize that if all pairs of signals in the STG are 'locked' using a chain of handshaking pairs then the STG satisfies the CSC property. The synthesis tool PETRIFY [2] uses the theory of regions [4] for this purpose.

These techniques work reasonably well. However, they may produce sub-optimal circuits or fail to solve the problem. Therefore, manual design is often crucial for finding good synthesis solutions, particulary in constructing interface controllers, where the quality of the solution is critical for the system's performance. According to practicing designers [7], the synthesis tool should offer a way for the designer to understand the characteristic patterns of a circuit's behaviour and the cause of each coding conflict, and

*This research was supported by EPSRC grants GR/M94366 (MOVIE) and GR/M99293.

let him/her interactively manipulate the model by choosing where in the specification to insert a new signal. The visualization method presented in this paper is aimed at facilitating a manual refinement of an STG with CSC conflicts, and works on the level of unfolding prefixes. In order to avoid the explicit enumeration of coding conflicts, they are visualized by *cores*, i.e., sets of transitions causing one or more of them. All such cores must eventually be eliminated by newly added signals to resolve the coding conflicts. This eventually results in an STG satisfying the CSC property.

2. Basic notions

In this section, we first present basic definitions concerning Petri nets and STGs, and then recall notions related to net unfoldings. For the sake of simplicity, we do not consider STGs with 'dummy' transitions here, but the theory can easily be generalized to include them.

2.1. Petri nets and STGs

A *net* is a triple $N \overset{\text{df}}{=} (S,T,F)$ such that S and T are disjoint sets of respectively *places* (circles) and *transitions* (boxes), collectively known as *nodes*, and $F \subseteq (S \times T) \cup (T \times S)$ is a *flow relation*. We denote ${}^\bullet z \overset{\text{df}}{=} \{y \mid (y,z) \in F\}$ and $z^\bullet \overset{\text{df}}{=} \{y \mid (z,y) \in F\}$, for all $z \in S \cup T$, and assume that ${}^\bullet t \neq \emptyset \neq t^\bullet$, for every $t \in T$. A *marking* of N is a multiset M of places, i.e., $M : S \to \mathbb{N} \overset{\text{df}}{=} \{0,1,2,\ldots\}$.

A *net system* is a pair $\Sigma \overset{\text{df}}{=} (N,M_0)$ comprising a finite net $N = (S,T,F)$ and an (initial) marking M_0. A transition $t \in T$ is *enabled* at a marking M, denoted $M[t\rangle$, if for every $s \in {}^\bullet t$, $M(s) \geq 1$. Such a transition can be *executed*, leading to a marking M' defined by $M' \overset{\text{df}}{=} M - {}^\bullet t + t^\bullet$, where '$-$' and '$+$' stand for the multiset difference and sum respectively. We denote this by $M[t\rangle M'$ or $M[\rangle M'$ if the identity of the transition is irrelevant. The set of *reachable* markings of Σ is the smallest (w.r.t. $\subset$) set $[M_0\rangle$ containing M_0 and such that if $M \in [M_0\rangle$ and $M[\rangle M'$ then $M' \in [M_0\rangle$. For a finite sequence of transitions, $\sigma = t_1 \ldots t_k$, we denote $M[\sigma\rangle M'$ if there are markings $M_0,\ldots,M_k$ such that $M_0 = M$, $M_k = M'$, and $M_{i-1}[t_i\rangle M_i$, for $i = 1,\ldots,k$.

A *Signal Transition Graph (STG)* is a triple $\Gamma \overset{\text{df}}{=} (\Sigma,Z,\lambda)$ such that $\Sigma = (N,M_0)$ is a net system, Z is a finite set of signals, which generate a finite alphabet $Z^{\pm} \overset{\text{df}}{=} Z \times \{+,-\}$ of *signal transition labels*, and $\lambda : T \to Z^{\pm}$ is a labelling function. The signal transition labels are of the form z^+ or z^-, and denote the transitions of signals $z \in Z$ from 0 to 1 (rising edge), or from 1 to 0 (falling edge), respectively.

We associate with the initial marking of Γ a binary vector $v^0 \overset{\text{df}}{=} (v^0_1,\ldots,v^0_{|Z|}) \in \{0,1\}^{|Z|}$, where v^0_i corresponds to the initial value of signal $z_i \in Z$. Moreover, with a sequence of transitions σ we associate an integer *signal change vector* $v^\sigma \overset{\text{df}}{=} (v^\sigma_1, v^\sigma_2,\ldots,v^\sigma_{|Z|}) \in \mathbb{Z}^{|Z|}$, so that each v^σ_i is the difference between the number of the occurrences of z_i^+–labelled and z_i^-–labelled transitions in σ.

Γ is *consistent* if, for every reachable marking M, all firing sequences σ from M_0 to M have the same *encoding* $Code(M) \overset{\text{df}}{=} v^0 + v^\sigma$, and this vector is binary, i.e., $Code(M) \in \{0,1\}^{|Z|}$. Such a property guarantees that, for every signal $z \in Z$, the STG satisfies the following two properties: (i) the first occurrence of z in the labelling of any firing sequence of Γ starting from M_0 has the same sign (either rising of falling); and (ii) the rising and falling labels z alternate in any firing sequence of Γ. All STGs considered in the sequel are assumed to be consistent.

The *state graph* of Γ is a tuple $SG_\Gamma \overset{\text{df}}{=} (S,A,s_0,Code)$ such that: $S \overset{\text{df}}{=} [M_0\rangle$ is the set of *states*; $A \overset{\text{df}}{=} \{M \xrightarrow{t} M' \mid M \in [M_0\rangle \wedge M[t\rangle M'\}$ is the set of *transitions*; $s_0 \overset{\text{df}}{=} M_0$ is the *initial state*; and $Code : S \to \{0,1\}^{|Z|}$ is the *state assignment* function, as defined above for markings.

Signals in Z are partitioned into input signals, Z_I, and output signals, Z_O (the latter may also include internal signals). Input signals are assumed to be generated by the environment, while output signals are produced by the logical gates of the circuit. Logic synthesis derives a boolean function $F_z(z_1,\ldots,z_{|Z|})$ for each output signal $z \in Z_O$, which requires the conditions for enabling of each output signal transition to be determined without ambiguity by the encoding of each reachable state. To capture this, let $Out(M) \overset{\text{df}}{=} \{z \in Z_O \mid \exists t \in T : M[t\rangle \wedge \lambda(t) = z^{\pm}\}$ be the set of enabled output signals, for every reachable state M. Two states of SG_Γ are in *CSC conflict* if they have the same encoding but different sets of enabled output signals. Γ satisfies the *Complete State Coding (CSC)* property if no two states of SG_Γ are in CSC conflict.

An example of an STG for a data read operation in a simple VME bus controller (a standard STG benchmark) is shown in Figure 1(a). Part (b) of this figure illustrates a CSC conflict between two different markings, M_1 and M_2, that have the same code, 10110, but $Out(M_1) = \{lds\} \neq Out(M_2) = \{d\}$. This means that, e.g., the value of $F_{lds}(1,0,1,1,0)$ is ill-defined (it should be 1 according to the state M_1 and 0 according to the state M_2), and thus lds is not implementable as a logic gate. To cope with this, an additional signal helping to resolve this CSC conflict should be added to the STG.

2.2. Branching processes and configurations

Two nodes of a net $N = (S,T,F)$, y and y', are in *structural conflict*, denoted by $y\#y'$, if there are distinct transitions $t,t' \in T$ such that ${}^\bullet t \cap {}^\bullet t' \neq \emptyset$ and (t,y) and (t',y') are in the reflexive transitive closure of the flow relation F, denoted by $\preceq$. A node y is in *structural self-conflict* if $y\#y$.

An *occurrence net* is a net $ON \overset{\text{df}}{=} (B,E,G)$ where B is the

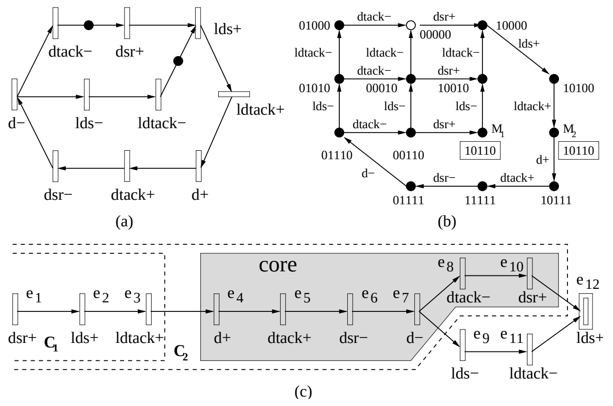


Figure 1. STG model of a simplified VME bus controller (a), its state graph with a CSC conflict between two states (b), and its unfolding prefix, with the corresponding conflict pair of configurations $\langle C_1, C_2\rangle$ (c). The order of signals in the binary codes is: *dsr*, *dtack*, *lds*, *ldtack*, *d*. **The core corresponding to the conflict pair is highlighted. Inputs:** *dsr*, *ldtack*; **outputs:** *lds*, *d*, *dtack*.

set of *conditions* (places) and E is the set of *events* (transitions). It is assumed that: ON is acyclic (i.e., $\preceq$ is a partial order); for every $b \in B$, $|{}^\bullet b| \leq 1$; for every $y \in B \cup E$, $\neg(y \# y)$ and there are finitely many y' such that $y' \prec y$, where $\prec$ denotes the irreflexive transitive closure of G. $Min(ON)$ will denote the minimal elements of $B \cup E$ with respect to $\preceq$. The relation $\prec$ is the *causality relation*. Two nodes are *concurrent*, denoted $y \; co \; y'$, if neither $y \# y'$ nor $y \preceq y'$ nor $y' \preceq y$.

A *homomorphism* from an occurrence net ON to a net system Σ is a mapping $h : B \cup E \to S \cup T$ such that: $h(B) \subseteq S$ and $h(E) \subseteq T$; for all $e \in E$, the restriction of h to ${}^\bullet e$ is a bijection between ${}^\bullet e$ and ${}^\bullet h(e)$; the restriction of h to $e^\bullet$ is a bijection between $e^\bullet$ and $h(e)^\bullet$; the restriction of h to $Min(ON)$ is a bijection between $Min(ON)$ and M_0; and for all $e, f \in E$, if ${}^\bullet e = {}^\bullet f$ and $h(e) = h(f)$ then $e = f$.

A *branching process* of Σ is a quadruple $\pi \overset{\text{df}}{=} (B, E, G, h)$ such that (B, E, G) is an occurrence net and h is a homomorphism from ON to Σ. A branching process $\pi' = (B', E', G', h')$ of Σ is a *prefix* of a branching process $\pi = (B, E, G, h)$, denoted $\pi' \sqsubseteq \pi$, if (B', E', G') is a subnet of (B, E, G) such that: if $e \in E'$ and $(b, e) \in G$ or $(e, b) \in G$ then $b \in B'$; if $b \in B'$ and $(e, b) \in G$ then $e \in E'$; and h' is the restriction of h to $B' \cup E'$. For each Σ there exists a unique (up to isomorphism) maximal (w.r.t. $\sqsubseteq$) branching process, called the *unfolding* of Σ.

A *configuration* of an occurrence net ON is a set of events $\mathcal{C}$ such that for all $e, f \in \mathcal{C}$, $\neg(e \# f)$ and, for every $e \in \mathcal{C}$, $f \prec e$ implies $f \in \mathcal{C}$. A *cut* is a maximal (w.r.t. $\subset$) set of conditions B' such that $b \; co \; b'$, for all distinct $b, b' \in B'$. Every marking reachable from $Min(ON)$ is a cut.

Let $\mathcal{C}$ be a finite configuration of a branching process π. Then $Cut(\mathcal{C}) \overset{\text{df}}{=} (Min(ON) \cup \mathcal{C}^\bullet) \setminus {}^\bullet\mathcal{C}$ is a cut; moreover, the multiset $h(Cut(\mathcal{C}))$ of places is a reachable marking of Σ, denoted $Mark(\mathcal{C})$. A marking M of Σ is *represented* in π if the latter contains a finite configuration $\mathcal{C}$ such that $M = Mark(\mathcal{C})$. Every marking represented in π is reachable, and every reachable marking is represented in the unfolding of Σ.

A branching process $\pi = (B, E, G, h)$ of Σ is *complete* if there is a set $E_{cut} \subseteq E$ of *cut-off* events such that, for every reachable marking M of Σ, there exist a finite configuration $\mathcal{C}$ of π such that $\mathcal{C} \cap E_{cut} = \emptyset$ and $M = Mark(\mathcal{C})$, and for each such $\mathcal{C}$ and every transition t enabled by M, there is an event $e \notin \mathcal{C}$ in π such that $h(e) = t$ and $\mathcal{C} \cup \{e\}$ is a configuration (e may be in E_{cut}).

Although, in general, an unfolding can be infinite, for every bounded net system Σ one can construct a finite complete prefix $Pref_\Sigma$ of the unfolding of Σ, by choosing an appropriate set E_{cut} of cut-off events, beyond which the unfolding is not generated.

A *branching process* of an STG $\Gamma = (\Sigma, Z, \lambda)$ is a branching process of Σ augmented with an additional labelling of its events, $\lambda \circ h : E \to Z^{\pm} \cup \{\tau\}$. One can easily check the consistency of Γ, once its finite and complete prefix has been built (see, e.g., [8]).

3. Coding conflicts in a prefix

In [5] an integer programming technique for detecting coding conflicts, employing STG unfolding prefixes, has been proposed. A CSC conflict can be represented as an unordered *conflict pair* of configurations $\langle \mathcal{C}_1, \mathcal{C}_2 \rangle$ whose final states are in CSC conflict, as shown if Figure 1(c). [5] builds a system of constraints whose set of solutions comprises such conflict pairs.

Note that the set of all such pairs may be quite large, e.g., due to the following 'propagation' effect: if $\mathcal{C}_1$ and $\mathcal{C}_2$ can be expanded by the same event e then $\langle \mathcal{C}_1 \cup \{e\}, \mathcal{C}_2 \cup \{e\} \rangle$ is also a conflict pair (unless these two configurations enable the same set of output signals). Therefore, it is desirable to reduce the number of such pairs to consider, e.g., as follows. A conflict pair $\langle \mathcal{C}_1, \mathcal{C}_2 \rangle$ is called *concurrent* if $\mathcal{C}_1 \not\subseteq \mathcal{C}_2$, $\mathcal{C}_2 \not\subseteq \mathcal{C}_1$, and $\mathcal{C}_1 \cup \mathcal{C}_2$ is a configuration. Below is a slightly modified version of a proposition proven in [5]:

Proposition 1. *Let $\langle \mathcal{C}_1, \mathcal{C}_2 \rangle$ be a concurrent CSC conflict pair. Then $\mathcal{C} \overset{\text{df}}{=} \mathcal{C}_1 \cap \mathcal{C}_2$ is such that either $\langle \mathcal{C}, \mathcal{C}_1 \rangle$ or $\langle \mathcal{C}, \mathcal{C}_2 \rangle$ is a CSC conflict pair.*

Thus the concurrent conflict pairs are 'redundant' and should not be considered. The remaining conflict pairs can be classified as follows:

Conflicts of type I are such that $\mathcal{C}_1 \subset \mathcal{C}_2$.

Conflicts of type II are such that $\mathcal{C}_1 \setminus \mathcal{C}_2 \neq \emptyset \neq \mathcal{C}_2 \setminus \mathcal{C}_1$, and there exist $e_1 \in \mathcal{C}_1 \setminus \mathcal{C}_2$ and $e_2 \in \mathcal{C}_2 \setminus \mathcal{C}_1$ such that $e_1 \# e_2$.

The following notion is crucial for the approach proposed in this paper:

Definition 1. Let $\langle \mathcal{C}_1, \mathcal{C}_2 \rangle$ be a conflict pair. The corresponding *complementary set* is defined as $\mathcal{CS} \overset{\mathrm{df}}{=} \mathcal{C}_1 \triangle \mathcal{C}_2$, where $\triangle$ is the symmetric set difference. $\mathcal{CS}$ is *core* if it cannot be represented as the union of several disjoint complementary sets. A complementary set is of type I/II if the corresponding conflict pair is of type I/II respectively.

Note that for a conflict pair $\langle \mathcal{C}_1, \mathcal{C}_2 \rangle$ of type I, such that $\mathcal{C}_1 \subset \mathcal{C}_2$, the corresponding core is simply $\mathcal{C}_2 \setminus \mathcal{C}_1$.

One can show that every complementary set $\mathcal{CS}$ can be partitioned into $\mathcal{C}_1 \setminus \mathcal{C}_2$ and $\mathcal{C}_2 \setminus \mathcal{C}_1$, where $\langle \mathcal{C}_1, \mathcal{C}_2 \rangle$ is a conflict pair corresponding to $\mathcal{CS}$. Moreover, if $\mathcal{CS}$ is of type I then one of these parts is empty, while the other is $\mathcal{CS}$ itself. An important property of complementary sets is that for each signal $z \in Z$, the difference between the numbers of z^+– and z^-–labelled events in $\mathcal{CS}$ is the same in these two parts, and is 0 if $\mathcal{CS}$ is of type I. This suggests that a complementary set can be eliminated by inserting a new signal into it, which would violate this property.

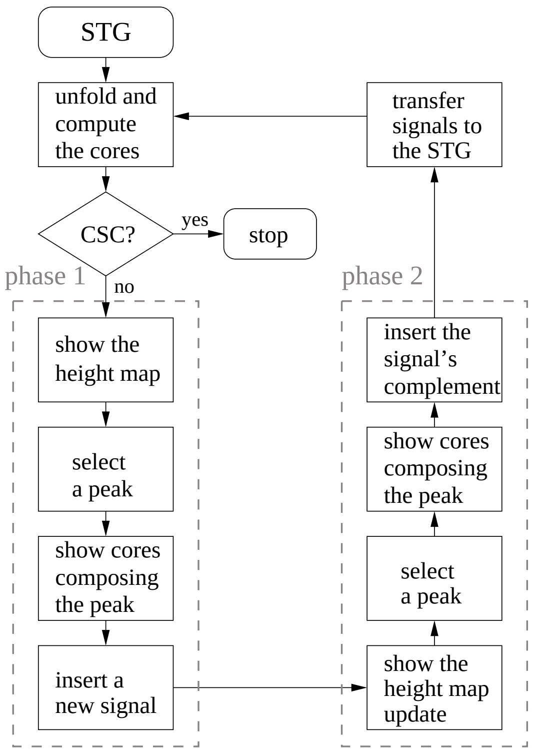
As an example, consider the conflict pair shown in Figure 1(c). The corresponding core is $\{e_4 - e_8, e_{10}\}$.

It is often the case that the same complementary set corresponds to different conflict pairs. For example, the STG shown in Figure 3(a) has four concurrent branches with a CSC conflict in each of them. Due to the mentioned 'propagation' effect, there are altogether 888 conflict pairs, a full list of which is probably too long for the designer to cope with. Despite of this, there are only 4 cores, as shown in Figure 3(b). (Note that there are 15 complementary sets, which can be obtained by uniting these cores.)

4. Framework for visualization and resolution of coding conflicts

The visualization is based on showing the user the cores in an STG unfolding prefix. Since every element of a core is an instance of the STG's transition, the cores can easily be mapped from the prefix to the STG and vice versa. For example, the core $\{e_4 - e_8, e_{10}\}$ in Figure 1(c) can be mapped to the set of transitions $\{d^+, dtack^+, dsr^-, d^-, dtack^-, dsr^+\}$ of the original STG shown in Figure 1(a).

Cores are important for resolving coding conflicts. By introducing an additional internal signal, say csc^+, one can split a core thus eliminating the corresponding coding conflicts. To preserve the consistency of the STG, the signal's counterpart csc^- must also be added to the specification outside the core, in such a way that it is neither concurrent to nor in structural conflict with csc^+ (it is sometimes possible to insert csc^- into another core thus eliminating it also). Another restriction is that an inserted signal cannot trigger



STG
unfold and compute the cores
transfer signals to the STG
CSC?
yes
stop
no
phase 1
phase 2
show the height map
select a peak
show cores composing the peak
insert a new signal
insert the signal's complement
show cores composing the peak
select a peak
show the height map update

Figure 2. The process of coding conflict resolving by cores.

an input signal (this would impose constraints on the environment which were not present in the original STG, making it wait for the newly inserted signal). More about the formal requirements for the correctness of inserting a new transition can be found in [2].

It is often the case that cores overlap. In order to minimize the number of inserted signals, and thus the area and latency of the circuit, it is advantageous to insert a signal in such a way that as many cores as possible are eliminated by it. That is, a signal should be inserted into the intersection of several cores. As an example, consider the cores shown in Figure 4(b). There are five cores altogether, but exploiting the fact that four of them overlap, it is possible to eliminate them all adding just one new signal: it should be inserted into the intersection of the four cores, and its counterpart — into the remaining core.

A key feature in the visualization process is the *height map*, showing the quantitative distribution of the cores. The events located in conflict cores are highlighted by shades of colours. The shade depends on the *altitude* of an event, i.e., on the number of cores it belongs to. The greater the altitude, the darker the shade. (The analogy is with a geographical map showing the altitudes.) 'Peaks' with the highest altitude are good candidates for insertion of a new signal, since they correspond to the intersection of maximum number of cores.

From this representation, the designer can select an area for inserting a new signal and obtain a local, more detailed description of the cores overlapping with the selection. When an appropriate core cluster is found, the designer can decide how to insert a new signal transition optimally, taking into account the design constraints and his knowledge of the system being developed.

The overview of the process is shown in Figure 2. Given an STG, a finite complete prefix of its unfolding is constructed, and the cores are computed. If there are none, the process stops. Otherwise, the 'height map' is shown to the designer. In phases one and two, a signal splitting some set of overlapping cores and its counterpart, respectively, are inserted (the designer can either accept an automatically proposed solution or check out his/her own). The inserted signals are then transferred to the STG, and the process is repeated. Depending on the number of CSC conflicts, the resolving process can consist of several cycles.

After the insertion of a signal in phase one, the height map is updated in the beginning of phase two in the following way. The altitudes of the events in the core cluster where the new signal has been inserted become negative, to prompt the designer that if the signal's counterpart is inserted there, some of the cores in the cluster will not be eliminated. Moreover, in order to ensure that the insertion of the signal's counterpart preserves consistency, the areas where the signal's counterpart cannot be inserted (in particular, the events concurrent to or in structural conflict with this signal) are faded out.

5. Case studies

In this section, two examples are discussed to demonstrate the proposed framework for resolving coding conflicts, and the resulting STGs are compared against those derived by PETRIFY. The first example is a handshake decoupling element shown in Figure 3(a). It has four mutually concurrent handshakes, which result in 888 CSC conflict pairs. Despite the huge number of these coding conflicts, there are only 4 cores shown in Figure 3(b).

In this case the height map is quite 'plain' since the cores do not overlap and thus no event has altitude greater than one. These cores are concurrent to each other and can be eliminated independently by adding four new signals. Let us start with the elimination of the first core. This process is illustrated in Figure 3(b). The internal transition csc_0^+ is inserted concurrently to b_1^+, between b_0^+ and b_0^-. Since its counterpart csc_0^- cannot be inserted concurrently to csc_0^+, parts of the prefix are faded out. In order to reduce the latency of the circuit, csc_0^- is inserted concurrently to a_0^-.

After transferring these signals to the STG and unfolding the result, we are left with the remaining three cores, which can be eliminated in a similar way. The final STG is presented in Figure 3(c). Note that in order to reduce the fan-in

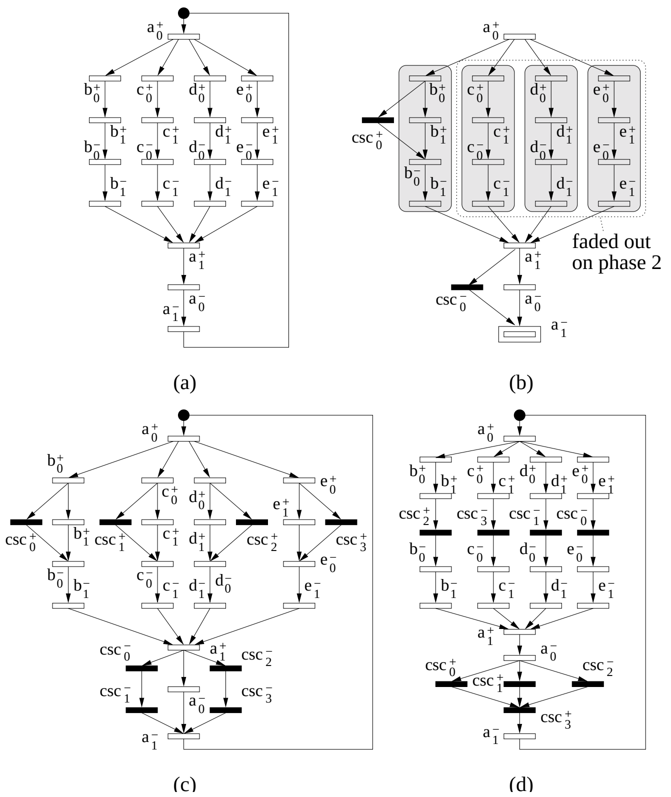


Figure 3. STG transformation: initial STG (a), the process of removal of the first core (b), the final STG (c), and the STG derived by PETRIFY **(d). Inputs:** a_0, b_1, c_1, d_1, e_1**; outputs:** a_1, b_0, c_0, d_0, e_0**; internal:** $csc_0, csc_1, csc_2, csc_3$**.**

at a_1^- and the fan-out at a_1^+, some of the signals' counterparts were inserted in series. Figure 3(d) shows the STG derived by the tool PETRIFY for this example. It also uses four signals to resolve the CSC conflicts, but its quality is not as good: almost all new signals and their counterparts are inserted sequentially, delaying output signals and thus significantly increasing the latency of the circuit.

The second example, shown in Figure 4(a), comes from real design practice. It is a part of the A-D converter proposed in [6]. It contains five type I and three type II CSC conflict pairs. They correspond to two type I and three type II cores, shown in Figure 4(b). The events e_3, e_6, e_9, and e_{13} comprise the highest peak, as each of them belong to four cores. This suggests that one can eliminate these cores by inserting a new signal, say csc^-, somewhere into this peak. In order to reduce the latency, it should be inserted concurrently to other signals. The only possibility to do so is to insert it concurrently to e_6, between e_3 and e_9, as shown in Figure 4(b). (Note that e_{13} corresponds to an input signal and thus cannot be delayed by a newly inserted event.)

In phase two, the signal's counterpart, csc^+, should be inserted in such a way that the consistency of the STG is preserved. (The events which are concurrent to or in struc-

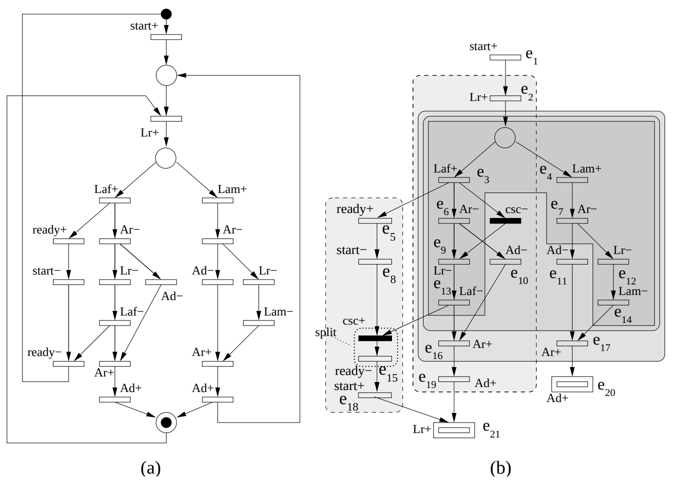


Figure 4. STG transformation: initial STG (a) and its unfolding prefix (b), with the cores and newly inserted signal highlighted. Inputs: *start*, *Lam*, *Laf*, *Ad*; **outputs:** *ready*, *Lr*, *Ar*; **internal:** *csc*.

tural conflict with csc^- are now faded out, as the consistency would be violated if the signal's counterpart is inserted there.) At the same time, one can try to eliminate the remaining core $\{e_5, e_8, e_{15}, e_{18}\}$. Since e_5 and e_8 are concurrent to csc^- and thus faded out, there is no way to insert csc^+ concurrently and one has to split e_{15}, as shown in Figure 4(b). After the inserted transitions are transferred into the STG, it has the CSC property.

The obtained solution is better than that produced automatically by PETRIFY: it inserts both the signal and its counterpart sequentially, by splitting e_9 and e_{15}, increasing the latency of the output Lr^- to three gate delays.

6. Conclusions and future work

In this paper, a framework for interactive refinement aimed at resolving coding conflicts in STGs has been presented. It is based on the visualization of conflict cores, which are sets of transitions involved in state coding conflicts. Cores are represented at the level of the STG unfolding prefix, which is a convenient model for understanding the behaviour of the system due to its simple branching structure and acyclicity.

The advantage of using cores is that only those parts of STGs which cause coding conflicts, rather than the complete list of CSC conflicts, are considered. Since the number of cores is usually much smaller than the number of coding conflicts, this approach saves the designer from analyzing large amounts of information.

The refinement contains several interactive steps aimed to help the designer in obtaining a customized solution. Resolving coding conflicts requires the elimination of cores by introducing additional signals into the STG. The case studies demonstrate the positive features of the interactive refinement process.

Heuristics for signal insertion based on the height map and exploiting the intersections of cores use the most essential information about coding conflicts, and thus should be quite efficient. The conflict resolving procedure can be automated either partially or completely, but in order to obtain an optimal solution, a semi-automated resolving process should be employed. Heuristics would suggest the areas for insertion of new signals, which could be used as guidelines. However, the designer is free to intervene at any stage and choose an alternative location, in order to take into account the design constraints.

We plan to incorporate also the concurrency reduction method of resolving coding conflicts [3] into the proposed framework. Furthermore, a tool is being developed to implement the described technique.

References

[1] T.-A. Chu: *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications*. PhD Thesis, MIT/LCS/TR-393 (1987).

[2] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev: *Synthesis of Asynchronous Controllers and Interfaces*. Springer Verlag (2002).

[3] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev: Automatic Handshake Expansion and Reshuffling Using Concurrency Reduction. Proc. of *HWPN'98*, (1998) 86–110.

[4] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev: A Region-Based Theory for State Assignment in Speed-Independent Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 16(8) (1997) 793–812.

[5] V. Khomenko, M. Koutny, and A. Yakovlev: Detecting State Coding Conflicts in STGs Using Integer Programming. Proc. of *DATE'02*, IEEE Comp. Soc. Press (2002) 338–345.

[6] D. J. Kinniment, B. Gao, A. Yakovlev, and F. Xia: Towards asynchronous A-D conversion. Proc. of *ASYNC'00*, IEEE Comp. Soc. Press (2000) 206–215.

[7] P. Riocreux: *Private communication*. UK Asynchronous Forum (2002).

[8] A. Semenov: *Verification and Synthesis of Asynchronous Control Circuits Using Petri Net Unfoldings*. PhD Thesis, University of Newcastle upon Tyne (1997).

[9] P. Vanbekbergen, F. Catthoor, G. Goossens, and H. De Man: Optimized Synthesis of Asynchronous Control Circuits form Graph-Theoretic Specifications. Proc. of *ICCAD'90*, IEEE Comp. Soc. Press (1990) 184–187.

[10] A. Yakovlev and A. Petrov: Petri Nets and Asynchronous Bus Controller Design. Proc. of *ICATPN'90*, (1990) 244–262.