ConfRes Documentation and User Guide
Version 1.4

A. Madalinski

University of Newcastle upon Tyne

December 2004

1 Legal information

See file LICENSE.CONFRES for the legal information and copyright notice.

2 Installation

The following tools are required:

PUNT Petri net unfolder [6]

CLP linear programming model checker [5]

DOT graph drawing software by AT&T [11]

DRAW_ASTG draws a Signal Transition Graph included in PETRIFY [3]

Note if you use PETRIFY [3], DOT and DRAW_ASTG are included in it. Copy the executable ('ConfRes' , 'g2myg' and 'mci2dot' into a folder listed in the value of the PATH variable.

3 Description

CONFRES [4, 12] supports semi-automated resolution of Complete State Coding (CSC) conflicts in asynchronous circuit specification given as Signal Transition Graphs (STGs) and display them as partial orders (finite and complete prefixes of STG unfoldings). Being more efficient than the automated methods the manual approach requires a significant effort from the designer. The tool CONFRES assists the designer by visualising the conflicts cores, their superposition and the constraints on signal insertion.

Cores extend the known concept of complementary sets. Only those complementary sets are used which are not combinations of others. The advantage of using cores is that only those parts of STGs are considered, which cause coding conflicts, rather than the complete set of conflicts. Since the number of cores is usually much smaller than the number of coding conflicts, this approach saves the designer from analysing large amounts of information. Moreover, 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.

Cores are important for resolving coding conflicts. They can be eliminated by adding auxiliary signals and by concurrency reduction, respectively. The former introduces additional internal signals to disambiguate encoding conflicts and the latter reduces the state space in the STG's reachability graph and thus potential encoding conflicts. The resolution process uses the partial order model and employs the visualisation concept as follows: (a) shows the superposition of cores by means of a 'height map' (b) only those cores are displayed which are relevant to selected part of the specification, i.e. cores which are extracted from the height map and (c) the constraints on insertion identify a small part of the specification rather than the entire design. In addition, heuristics are used to pre-compute solutions, which are suggested to the designer. These can be used as guidelines, however, the designer is free to intervene at any stage and choose an alternative solution in order to account the design constraints.

ConfRes depends on other tools as depicted in Figure 1. It takes an STG in *astg* format supported by PETRIFY [2], an STG-based synthesis tool. It uses PUNF [6], a Petri net unfolder,

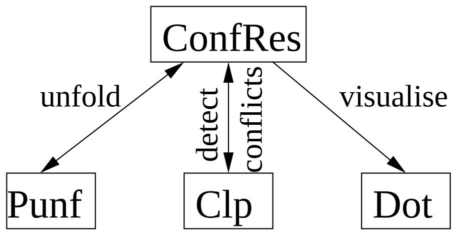


Figure 1: ConfRes dependencies

to produce a finite and complete prefix of the STG, and Clp [5], a linear programming model checker, to detect coding conflicts in the STG. After the detection of conflicts, cores are computed and the resolution process is applied. The tool guides the designer through all the stages. During this process the cores and the corresponding height map are visualised using Dot [11], a graph drawing software by AT&T , and the designer can interactively resolve conflicts to obtain a customised solution.

4 Interactive coding conflict resolution

This section presents a framework for an interactive interactive resolution process of encoding conflict in STGs. It is based on the visualisation of conflict cores, which are represented at the level of finite and complete prefixes of STG unfolding.

4.1 Introduction

Signal Transition Graphs (STGs) [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. There exists a number of methods (reviewed in[2]) for the synthesis of circuits from STG specifications.

Complete State Coding (CSC) is an STG property crucial for the implementability of next-state functions as circuits. A CSC conflict arises when semantically different states of an STG have the same binary encoding. This means that the binary encoding alone cannot determine the future behaviour of the system, hence ambiguities arise when trying to define the next-state functions. Figure 2(b) shows the state graph of the STG in Figure 2(a) with a CSC conflict between states M_1 and M_2.

A common approach to detect and solve CSC conflicts is to construct the reachable state space of the initial STG specification to implement it. While such an approach can be used in completely automatic synthesis, it has several flaws: state graphs are not visual, which prevents efficient interaction with the user. Moreover, the combinatorial explosion of the state space is a serious issue for highly concurrent STGs. This makes alternative techniques, and in particular those based on Petri net unfoldings, very attractive for this task.

A *finite and complete unfolding prefix* of an STG Γ is a finite acyclic net which implicitly represents all the reachable states of Γ together with transitions enabled at those states. Intuitively, it can be obtained through *unfolding* Γ, by successive firings of transition, under the following assumptions: (i) for each new firing a fresh transition (called an *event*) is generated; (ii) for each newly produced token a fresh place (called a *condition*) is generated. The unfolding is infinite whenever Γ has an infinite run; however, if Γ has finitely many reachable states then the unfolding eventually starts to repeat itself and can be truncated (by identifying a set of *cut-off* events) without loss of information, yielding a finite and complete prefix. Figure 2(c) shows a finite and complete unfolding prefix (with the only cut-off event depicted as a double box) of the STG shown in Figure 2(a).

In [7] the unfolding technique was applied to the implementability analysis, checking the CSC condition, which requires detecting coding conflicts between an STG's states. Since STGs usually exhibit a lot of concurrency, but have 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 [7] they are just slightly bigger then the original STGs themselves. Therefore, unfolding prefixes are well-suited for both visualisation of an STG's behaviour and alleviating the state space explosion problem.

Enforcing the CSC condition by completely automatic synthesis, which uses heuristics, may produce sub-optimal circuits or fail to solve the CSC problem. Therefore, manual intervention is often crucial for finding good synthesis solutions, particularly when system's performance is of paramount importance. As the result designers would like to interactively manipulate the model by choosing where to insert a new signal. This would facilitate to trade-off various constraints of the design. Thus 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.

ConfRes 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 visualised by cores [13], which are the essential causes of conflicts. All such cores must eventually be eliminated by either newly added signals or concurrency reduction. This eventually results in an STG satisfying the CSC property.

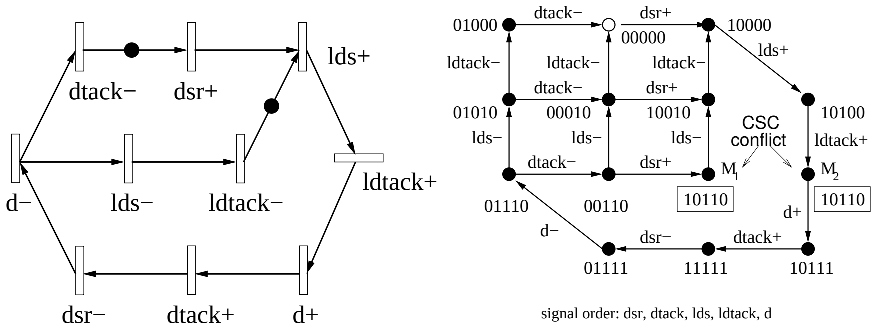
4.2 Representation of encoding conflicts

Due to structural properties of unfoldings (such as acyclicity), the reachable markings of Σ can be represented using *configurations*. A configuration C is a downward-closed set of events (being downward-closed means that if $e \in C$ and f is a causal predecessor of e then $f \in C$) without *structural conflicts* (i.e. for all distinct events $e, f \in C$, there is no condition c in the unfolding such that the arcs (c, e) and (c, f) are in the unfolding). For example, in Figure 7(b) $\{e_0, e_1, e_2, e_4, e_7, e_8\}$ is a configuration whereas $\{e_0, e_1, e_2, e_3, e_4\}$ and $\{e_0, e_2\}$ are not (the former includes a structural conflict between the events e_2 and e_3, while the latter does not include e_1, a causal predecessor of e_2). Intuitively, a configuration is a partial-order execution, i.e. an execution where the order of firing of concurrent events is not important; i.e. the configuration $\{e_0, e_1, e_2, e_4, e_7, e_8\}$ corresponds to two totally ordered executions: $e_0e_1e_2e_4e_7e_8$ and $e_0e_1e_2e_4e_8e_7$.

In an unfolding prefix a CSC conflict can be represented as an unordered *conflict pair* of configurations $\{C_1, C_2\}$ whose final states are in CSC conflict, as shown if Figure 2(c). In [5, 8] two techniques for detecting CSC conflicts (based, respectively, on integer programming and SAT) were proposed. Essentially, they allow for efficiently finding such conflict pairs in STG unfolding prefixes.

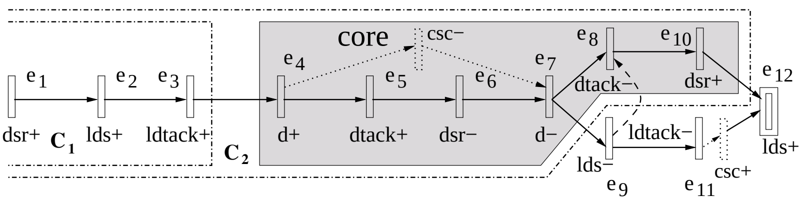
In [13] conflict pairs are classified as follows:

Conflicts of type I are such that $C_1 \subset C_2$.



(a) STG of the read cycle

(b) state graph with CSC conflict between M1 and M2



(c) unfolding prefix with the corresponding core

inputs: *dsr*, *ldtack*; **outputs:** *lds*, *d*, *dtack*; **internal:** *csc*

Figure 2: VME bus controller

Conflicts of type II are such that $C_1 \setminus C_2 \neq \emptyset \neq C_2 \setminus C_1$ and there exist $e_1 \in C_1 \setminus C_2$ and $e_2 \in C_2 \setminus C_1$ such that $e_1 \# e_2$.

At the level of unfoldings, encoding conflicts can be compactly represented using conflict *cores* [13].

Definition. Core
Let $\{C_1, C_2\}$ be a conflict pair. The corresponding *complementary set* is defined as $CS = C_1 \triangle C_2$, where $\triangle$ is the symmetric set difference. 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.

For example, the core corresponding to the conflict pair shown in Figure 2(c) is $\{e_4, \ldots, e_8, e_{10}\}$ (note that for a conflict pair $\{C_1, C_2\}$ of type I, such that $C_1 \subset C_2$, the corresponding complementary set is simply $C_2 \setminus C_1$), and the core corresponding to the conflict pair $\langle \{e_0, e_1, e_3, e_6, e_{11}, e_{13}\}, \{e_0, e_1, e_2, e_4, e_8, e_{12}\} \rangle$ in Figure 7(a) is $\{e_2, e_3, e_4, e_6, e_8, e_{11}, e_{12}, e_{13}\}$.

One can show that every complementary set CS can be partitioned into $C_1 \setminus C_2$ and $C_2 \setminus C_1$, where $\{C_1, C_2\}$ is a conflict pair corresponding to CS. Moreover, if CS is of type I then one of these parts is empty, while the other is 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 CS is the same in these two parts, and is 0 if s 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.

5 Resolution of encoding conflicts

Encoding conflicts can be resolved by either adding auxiliary signals or by concurrency reduction. The former approach was studied in [13], where additional signals are employed to disambiguate states having the same binary encodings. The latter [9] makes some of the states unreachable and thus can eliminate encoding conflicts.

Table 1 shows some of the transformations which can be used to eliminate conflict cores. They include signal insertion (either sequential or concurrent) and concurrency reduction. The sequential signal insertion "splits" an existing transition and inserts a new transition either before or after it, and the concurrent insertion inserts a new transition concurrently to at least one existing transition (optionally, tokens are added into one of the newly created places to ensure that w has not become dead). The concurrency reduction introduces a new causal constraint in form of a new place (with, perhaps, a few tokens) between existing transitions. The concurrency reduction depicted in the last row of Table 1 will be denoted by $U \overset{n}{\dashrightarrow} t$, where $U \neq \emptyset$, $t \notin U$ and $n \in \mathbb{N}$. We will write $U \dashrightarrow t$ instead of $U \overset{0}{\dashrightarrow} t$ and $u \overset{n}{\dashrightarrow} t$ instead of $\{u\} \overset{n}{\dashrightarrow} t$. Note that concurrency reduction cannot add new behaviour to the system — it can only restrict it.

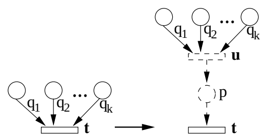
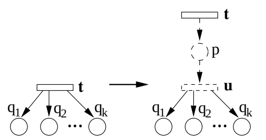
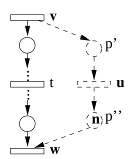
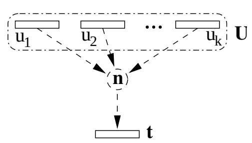
insertion before t	
insertion after t	
insertion concurrently to t between v and w	
concurrency reduction	

Table 1: Transformations used to resolve encoding conflicts

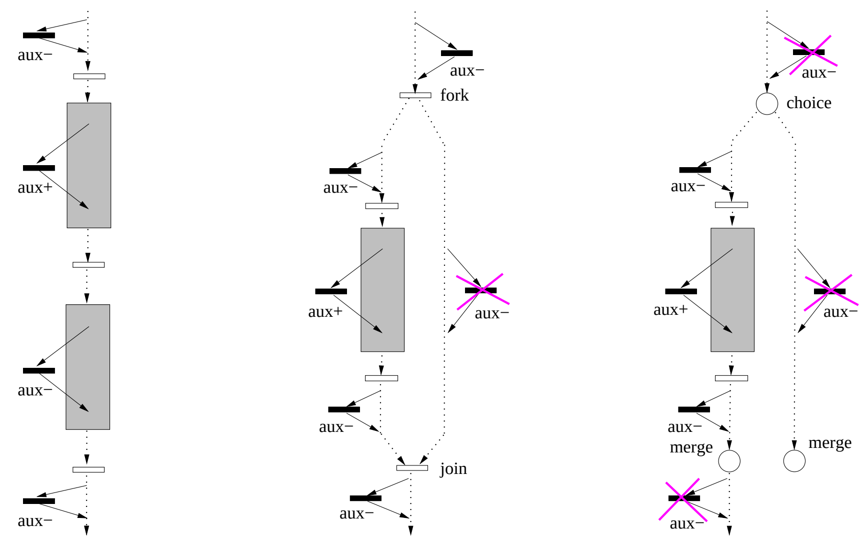
Usually concurrency reduction leads to smaller circuits compared to signal insertion, and it may also be the case that the resulting circuit is faster due to simplification of the gates. Thus, even though the system manifests less concurrency, it might be actually faster due to the events taking less time to fire. On the other hand, there are situations when signal insertion produces better solutions. A combined framework is presented here, which uses both signal insertion and concurrency reduction to eliminate cores and the corresponding encoding conflicts. This allows to explore a larger design space.

It is often the case that cores overlap. In order to minimise the number of transformations, and thus the area and/or latency of the circuit, one should try to eliminate as many conflict cores as possible on each step of the resolution process. That is, a transformation should be performed in *the intersection of several cores* whenever possible. In [13] the exploitation of core overlaps is implemented by means of a *height map* showing the quantitative distribution of the cores. Each event in the prefix is assigned an *altitude*, i.e. the number of cores it belongs to. (The analogy with a topographical map showing the altitudes may be helpful here.) 'Peaks' with the highest altitude are good candidates for insertion, since they correspond to the intersection of maximum number of cores.

When modifying an STG to resolve encoding conflicts, an important restriction is that the performed transformations must not 'delay' any input signal transition. The reason is that this would impose constraints on the environment which were not present in the original STG, making it 'wait' for some signal.

5.1 Core elimination by signal insertion

A framework for visualisation and manual resolution of encoding conflicts was presented in [13], where cores were eliminated by signal insertion. We briefly describe it here.



(a) sequential (b) concurrent (c) structural conflict

Figure 3: Strategies for core elimination by signal insertion

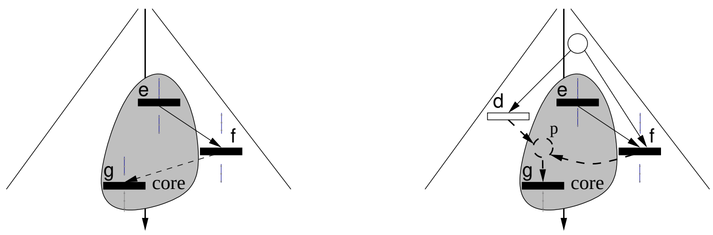
By introducing an additional internal signal and insertion of its transition, say csc^+, into the core one can destroy it eliminating thus the corresponding encoding conflicts. To preserve the consistency of the STG, the transition's counterpart csc^- must also be inserted *outside the core,* in such a way that it is neither concurrent to nor in structural conflict with csc^+.

For example, the core in Figure 2(c) can be eliminated by inserting a transition of a new signal, *csc*, somewhere in the core, i.e. concurrently to e_5 and e_6 between e_4 and e_7, and by inserting its complement outside the core, e.g. after e_{11}. After transferring this signal into the STG, it satisfies the CSC property. (Other ways of inserting a signal to resolve the encoding conflict are also possible.)

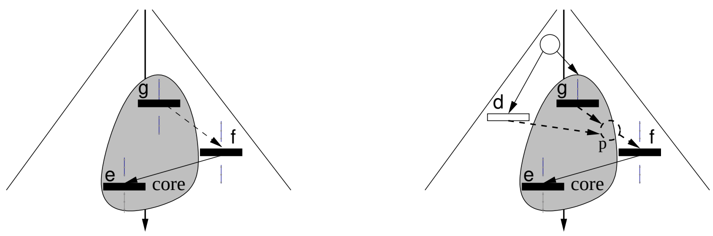
The elimination of encoding conflicts by signal insertion is schematically illustrated in Figure 3, which represent typical cases in STG specifications (see [13] for more details).

5.2 Core elimination by concurrency reduction

Concurrency reduction removes some of the reachable states of the STG and thus can be used for the resolution of encoding conflicts. The elimination of conflict cores by concurrency reduction [9] involves the introduction of additional ordering constraints, which fix some order of execution of concurrent transitions. In an STG, a fork transition defines the starting point of concurrency and a join transition defines the end point. Existing signals can be used to disambiguate the conflicting states in a core by delaying the starting point or bringing forward the ending point of concurrency. If there is an event concurrent to the core, and a starting or ending point of concurrency is in the core, then this event can be forced into the core by an additional ordering constraint, thus destroying it. For example, in Figure 2(c), e_9 is concurrent to some of the events in the core, and the starting point of concurrency is in the core, so the concurrency reduction shown by the dashed line in this figure can be used to eliminate the core by 'dragging' e_9 into it.



(a) forward



(b) backward

Figure 4: Core elimination by concurrency reduction.

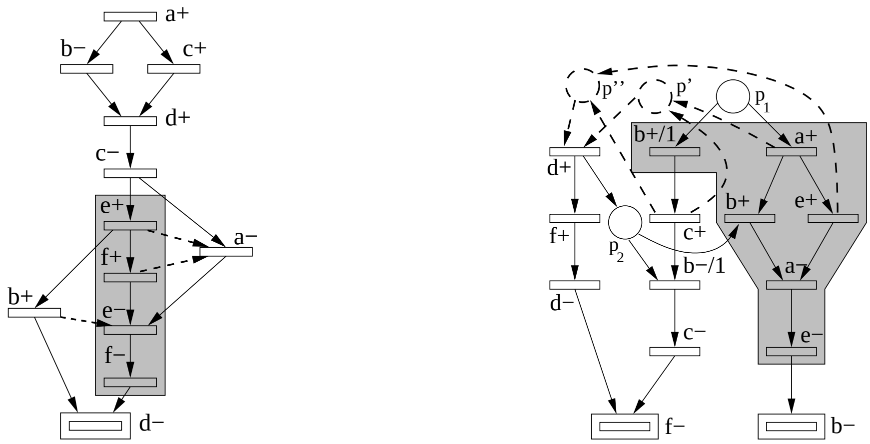
Two kinds of concurrency reduction based transformations for core eliminations are described below (where h is the mapping from the nodes of the prefix to the nodes of the STG).

Forward concurrency reduction illustrated in Figure 4(a) performs the concurrency reduction $h(E_U) \overset{n}{\dashrightarrow} h(g)$ in the STG, where E_U is a maximal (w.r.t. $\subset$) set of events outside the core which are in structural conflict with each other and concurrent to g — an event in the core. It is assumed that e is in the core and either precedes g or is concurrent to g, and for exactly one event $f \in E_U$, e precedes f.

Backward concurrency reduction illustrated in Figure 4(b) works in a similar way, but the concurrency reduction $h(E_U) \overset{n}{\dashrightarrow} h(f)$ is performed. It is assumed that e is in the core, f is an event outside the core such that f precedes e, E_U is a maximal (w.r.t. $\subset$) set of events which are in structural conflict with each other and concurrent to f, such that exactly one event $g \in E_U$ is in the core, and g either precedes e or is concurrent to e.

In both cases the core is destroyed by additional ordering constraints 'dragging' f into the core.

The above two rules are illustrated by the examples in Figure 5(a,b), where they are applied to cores of types I and II, respectively. In Figure 5(a) instances of b^+ and a^- are concurrent to the core. The forward concurrency reduction $b^+ \dashrightarrow e^-$ can be applied, because b^+ succeeds e^+ and e^- succeeds e^+. This 'drags' b^+ into the core, destroying it. Note that f is an input and hence must not be delayed, and so the concurrency reductions $b^+ \dashrightarrow f^+$ and $b^+ \dashrightarrow f^-$ would be invalid. The backward concurrency reductions $e^+ \dashrightarrow a^-$ and $f^+ \dashrightarrow a^-$ can also be applied to eliminate the conflict core, because a^- precedes e^-, and both e^+ and f^+ are in the core and precede e^-. Either of these reductions 'drags' a^- into the core, destroying it.



inputs: b, c, f; **outputs:** a, d, e

forward reduction: $b^+ \dashrightarrow e^-$

backward reductions: $e^+ \dashrightarrow a^-$; $f^+ \dashrightarrow a^-$

(a)

inputs: a, b, f; **outputs:** c, d, e

backward reductions: $\{a^+, c^+\} \dashrightarrow d^+$; $\{c^+, e^+\} \dashrightarrow d^+$

(b)

Figure 5: Elimination of type I (a) and type II (b) cores

In Figure 5(b) the type II core can be eliminated by backward concurrency reduction, because d^+ is concurrent to a^+ and e^+ in the core and precedes b^+ in the core. The only events in the core which either precede or are concurrent to b^+ are a^+ and e^+, and either of them can be used to eliminate the core. However, both reductions $a^+ \dashrightarrow d^+$ and $e^+ \dashrightarrow d^+$ are invalid, since they introduce deadlocks. (Note that these transformations are ruled out by the maximality requirement in the definition of a backward concurrency reduction.) Thus c^+ should be involved, yielding the following two backward concurrency reductions eliminating the core: $\{a^+, c^+\} \dashrightarrow d^+$ and $\{c^+, e^+\} \dashrightarrow d^+$. Note that the reductions $\{a^+, b^+/1\} \dashrightarrow d^+$ and $\{b^+/1, e^+\} \dashrightarrow d^+$ do not eliminate the core, because d^+ is 'dragged' into both branches of the core, and so the net sum of signals in these two branches remains equal. (And our backward concurrency reduction rule does not allow to use these two transformations, since only one event from the set E_U is allowed to be in the core.)

5.3 Implementation

In our framework, encoding conflicts can be eliminated by the introduction of auxiliary signals and concurrency reductions. A cost function was developed to heuristically select on each iteration of the encoding conflict resolution process the best transformation. It has the form

$$cost = \alpha_1 \cdot \Delta\omega + \alpha_2 \cdot \Delta\text{logic} + \alpha_3 \cdot \Delta cores\ .$$

The first part of the cost function, $\Delta\omega$, estimates the delay caused by a transformation. A delay model where each transition of the STG is assigned an individual delay is considered; e.g. input signals usually take longer to fire than non-input ones, because they often denote the end of a certain computation in the environment. (This delay model is similar to that in [2].) It is quite crude, but it is hard to significantly improve it, since the exact time behaviour is only known after the circuit and its environment are synthesised. In our experiments we assumed that the delay of input signal transitions is 3 time units and the delay of output and internal signal transitions is 1 time unit (these can easily be adjusted by the designer).

The second part of the cost function, Δlogic, estimates the increase in the complexity of the logic. The logic complexity is estimated using the number of *triggers* of each non-input signal z (i.e. signals whose firing can enable z), which can be computed on the unfolding prefix. Note that all the triggers of z are always in the support of the complex gate implementing z. In addition to triggers, other signals, called *context signals* can also be in the support of z, so the estimate of logic complexity based on triggers is also quite crude. However, the context signals cannot be computed for z until all the encoding conflicts for z are resolved.

The last part of the cost function, $\Delta cores$, is the number of cores eliminated by the transformation.

The parameters $\alpha_{1,2,3} \geq 0$ are given by the designer and can be used to direct the heuristic search towards reducing the delay inflicted by the transformation (α_1 is large compared with α_2 and α_3) or the estimated complexity of logic (α_2 and α_3 are large compared with α_1). (See [9] for more details.)

The resolution process involves finding an appropriate transformation for the elimination of cores in the STG unfolding prefix, as was explained earlier. The following steps are used to resolve the CSC conflicts:

1. Construct an STG unfolding prefix.

2. Compute the cores and, if there are none, terminate.

3. Choose areas for transformation (the 'highest peaks' in the height map corresponding to the overlap of the maximum number of cores are good candidates).

4. Compute valid transformations for the chosen areas and sort them according to the cost function; if no valid transformation is possible then

    - change the transformation areas by including the next highest peak and repeat step 4;
    - otherwise manual intervention by the designer is necessary; the progress might still be possible if the designer relaxes some I/O constraints, uses timing assumptions, etc.

5. Select the best according to the cost function transformation; if it is a signal insertion then the location for insertion of the counterpart transition is also chosen.

6. Perform the selected transformation and continue with step 1.

5.4 Interactive resolution of encoding conflicts: an example

This example comes from the design of an asynchronous AD converter described in [2, 10]. The structure of the converter is shown in Figure 6. It consists of a comparator, a 4-bit register and control logic. The comparator uses analog circuitry and a bistable device which has a completion detector capable of resolving metastable states.

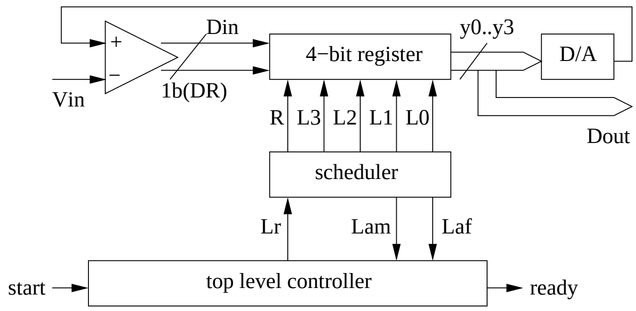
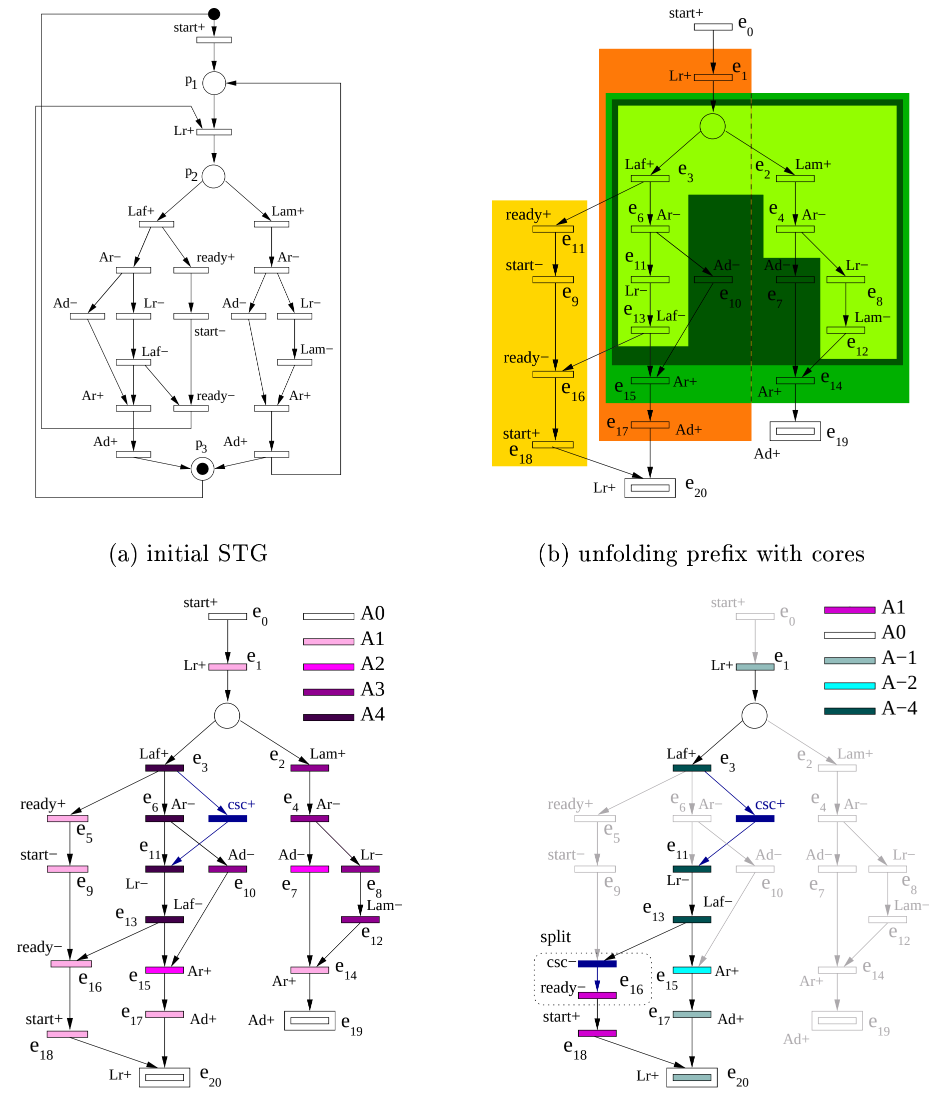


Figure 6: Block diagram of the AD converter

The 4-bit register is effectively a combination of a demux and a register with serial dual-rail input and parallel single-rail output. It is built from transparent latches. The register is controlled by five command signals arriving from the control logic in a one-hot (non-overlapping) manner. The first command signal, *R*, is used to reset the register to the initial state, immediately before starting the conversion. For all other command signals, *Li*, the register latches the value from input *Din* to the appropriate bit *y(i)* and set the lower bit, *y(i-1)* to 1.

The control logic consist of a five-way scheduler and a top level controller. The latter is described in Figure 7(a) and works as follows. The environment issues the *start+* signal when the analog value is ready for conversion. Then the controller issues a request to the scheduler, using the signal *Lr* (loud request), and when the appropriate latching of the next bit into the register has occurred, it receives an acknowledgement, *Lam* (load acknowledgement for middle). Now the controller executes a handshake with the comparator to force the next bit out. This is repeated until the last bit has been produced, and then the scheduler issues another acknowledgement signal, *Laf* ((load acknowledgement for final), which is mutually exclusive with respect to *Lam*. Finally the controller generates the *ready+* signal to the environment, indicating that the 4-bit code is available on the outputs of the register.



(a) initial STG

(b) unfolding prefix with cores

height map phase one

height map phase two

(c) transformation: $Laf^+ \longrightarrow\!\!\!\!\rightarrow Lr^-$ and $\rightarrow\!\wr ready^-$

inputs: *start*, *Lam*, *Laf*, *Ad*; **outputs:** *ready*, *Lr*, *Ar*; **internal:** *csc*

Figure 7: STG transformation of the top level controller

concurrency reduction				
#	causal constraint		lits	delay
1	$h(e_9) \dashrightarrow h(e_{11})$	3; -1; -5	11	1.1ns
2	$h(e_5) \dashrightarrow h(e_{11})$	0; 2; -5	14	1.2ns
3	$h(e_5) \dashrightarrow h(e_6)$	1; 2; -5	14	1.5ns
4	$h(e_9) \dashrightarrow h(e_6)$	4; 2; -5	11	1.4ns
5	$h(e_{10}) \dashrightarrow h(e_{11})$	3; 0; -1	n/a	n/a

(a) possible concurrency reductions

signal insertion					
#	phase 1	phase 2		lits	delay
6	$\parallel h(e_3)$ to $h(e_{11})$	before $h(e_{16})$	1; 3; -5	16	1.2ns
7	after $h(e_3)$	before $h(e_{16})$	3; 2; -5	15	1.5ns
8	before $h(e_6)$	before $h(e_{16})$	2; 3; -5	16	1.2ns
9	before $h(e_{11})$	before $h(e_{16})$	2; 4; -5	15	1.3ns
10	after $h(e_3)$	after $h(e_9)$	3; 3; -5	18	1.5ns
11	after $h(e_3)$	$\parallel h(e_5)$ to $h(e_{16})$	2; 4; -5	20	1.6ns

(b) possible signal insertions

$$\begin{aligned} \text{ready} &= \text{Laf} \\ \text{Lr} &= \text{start}\cdot\text{Ad}\cdot\text{Ar} + \text{Laf}\cdot(\text{Ar}+\overline{\text{start}}) \\ \text{Ar} &= \overline{\text{Lam}}\cdot\overline{\text{Laf}}\cdot(\text{Ar}+\overline{\text{Ad}}) \end{aligned}$$

equations for solution 1

$$\begin{aligned} \text{ready} &= \text{start}\cdot\text{ready} + \text{Laf} \\ \text{Lr} &= \text{start}\cdot\text{Ad}\cdot\overline{\text{ready}}\cdot\text{Ar} + \text{Laf}\cdot(\text{Ar}+\overline{\text{ready}}) \\ \text{Ar} &= \overline{\text{Lam}}\cdot\overline{\text{Laf}}\cdot(\text{Ar}+\overline{\text{Ad}}) \end{aligned}$$

equations for solution 2

$$\begin{aligned} \text{ready} &= \text{Laf} + \text{csc} \\ \text{Lr} &= \text{start}\cdot\text{Ad}\cdot\text{Ar}\cdot\overline{\text{csc}} + \text{Laf}(\overline{\text{csc}}+\text{Ar}) \\ \text{Ar} &= \overline{\text{Lam}}\cdot\overline{\text{Laf}}\cdot(\text{Ar}+\overline{\text{Ad}}) \\ \text{csc} &= \text{start}\cdot\text{csc} + \text{Laf} \end{aligned}$$

equations for solution 6

$$\begin{aligned} \text{ready} &= \text{csc} \\ \text{Lr} &= \text{Ar}\cdot(\text{start}\cdot\overline{\text{csc}}\cdot\text{Ad} + \text{Laf}) \\ \text{Ar} &= \overline{\text{Lam}}\cdot\overline{\text{Laf}}\cdot(\text{Ar}+\overline{\text{Ad}}) + \text{Laf}\cdot\overline{\text{csc}} \\ \text{csc} &= \text{start}\cdot\text{csc} + \text{Laf} \end{aligned}$$

equations for solution 7

$$\begin{aligned} \text{ready} &= \text{Laf} + \text{csc} \\ \text{Lr} &= \text{Ar}\cdot(\text{start}\cdot\overline{\text{csc}}\cdot\text{Ad} + \text{Lr}) \\ \text{Ar} &= \overline{\text{Lam}}\cdot\overline{\text{Laf}}\cdot(\text{Ar}+\overline{\text{Ad}}) + \text{Laf}\cdot\overline{\text{csc}} \\ \text{csc} &= \text{start}\cdot\text{csc} + \text{Laf} \end{aligned}$$

equations for solution 8

$$\begin{aligned} \text{ready} &= \text{Laf} + \text{csc} \\ \text{Lr} &= \overline{\text{csc}}\cdot(\text{start}\cdot\text{Ar}\cdot\text{Ad} + \text{Laf}) \\ \text{Ar} &= \overline{\text{Lam}}\cdot\overline{\text{Laf}}\cdot(\text{Ar}+\overline{\text{Ad}}) \\ \text{csc} &= \text{start}\cdot\text{csc} + \text{Laf}\cdot\overline{\text{Ar}} \end{aligned}$$

equations for solution 9

(c) a selection of equations

inputs: *start*, *Lam*, *Laf*, *Ad*; **outputs:** *ready*, *Lr*, *Ar*; **internal:** *csc*

Figure 8: Valid transformations for top level controller

The example shown in Figure 7(a) contains two type I and three type II cores shown in Figure 7(b). The valid transformations are presented in the tables in Figure 8(a,b), together with the values of the three components of the cost function, the total number of literals in the corresponding equations, and the worst-case input-to-output delays obtained by analog simulation (using the AMS-0.35μ CMOS technology). The equations for most interesting of these solutions are shown in Figure 8(c).

The events e_3, e_6, e_{11} and e_{13} comprise the highest peak, as each of them belongs to four cores. They can be eliminated by a forward concurrency reduction, since events e_5 and e_9 are concurrent to the events in the peak and the concurrency starts in the peak. The first four solutions in the table in Figure 8(a) eliminate all the cores in the peak, and the last one eliminates only one core. Incidentally, the first four solutions eliminate the remaining core as well, because the corresponding ordering constraints also act as backward concurrency reductions.

The first solution introduces a large delay (e_{11} is delayed by an input event e_9) but no additional triggers (in fact, the number of triggers of Lr is reduced, since Ar ceases to be its trigger). As a result, this is a very area-efficient solution — its literal count is just 11. The simulated worst-case input-to-output delay for this solution is 1.1ns ($Lam^+ \rightarrow Ar^- \rightarrow Lr^-$). However, it is somewhat misleading, since it does not take into account that the introduced causal constraint indirectly delays Laf^- by the input event $start^-$, which can be slow. The delay penalty estimate in out cost function better reflects the real situation.

The second solution does not delay e_{11} but introduces an additional trigger to Lr^-. As a result, its literal count, 14, is larger than that for the first solution, but its delay estimate in the cost function is better. The simulated worst-case input-to-output delay for this solution is 1.2ns ($Lam^+ \rightarrow Ar^- \rightarrow Lr^-$). However, it is achieved not where the concurrency was reduced. In fact, the delays in the paths $Laf^+ \rightarrow ready^+ \rightarrow Lr^-$ and $Laf^+ \rightarrow Ar^- \rightarrow Lr^-$ are 1.0ns.

The third solution delays e_6 by e_5, and the fourth solution delays e_6 by e_5 and e_9; moreover, these two solutions introduce an additional trigger to Ar (which already had three triggers), and thus are inferior according to both our cost function and the simulation results.

Alternatively, the encoding conflicts can be solved by inserting a transition of a new signal *csc* into the peak and its counterpart outside the cores belonging to the peak, preserving the consistency and ensuring that the cores are destroyed. Recall that input signal transitions must not be delayed by newly inserted transitions, i.e. in the peak the transition of *csc* must not delay e_3 and e_{13}. Then the parts of the prefix which are concurrent to or in structural conflict with the inserted transition are faded out, as the consistency would be violated if the counterpart transition of *csc* is inserted there. At the same time, one can try to eliminate the remaining core $\{e_5, e_9, e_{16}, e_{18}\}$. The signal insertion solutions are presented in the table in Figure 8(b).

Solution 6 (see Figure 7(c)) introduces the smallest (among all signal insertions) delay (only $ready^-$ is delayed), and its literal count is 16. Its worst-case input-to-output delay is 1.2ns ($Lam^+ \rightarrow Ar^- \rightarrow Lr^-$), but it is achieved not in the branch where *csc* was inserted; the delays in the paths $start^- \rightarrow csc^- \rightarrow ready^-$ and $Laf^- \rightarrow csc^- \rightarrow ready^-$ are just 0.7ns.

Solution 7 has the smallest (among all signal insertions) estimated logic complexity, but quite a large delay (the insertion delays $ready^+$, Ar^- and $ready^-$). Its literal count is 15 and the worst-case input-to-output delay is 1.5ns ($Laf^+ \rightarrow csc^+ \rightarrow Ar^- \rightarrow Lr^-$).

Solution 8 has the literal count 16 and the worst-case input-to-output delay of 1.2ns, which is as good as the delay of solution 6. However, unlike the worst-case delay in solution 6, it is achieved in the path $Laf^- \to csc^+ \to Ar^- \to Lr^-$, i.e. where csc was inserted.

Solutions 10 and 11 are inferior to other solutions in the table, which is in good agreement with our cost function. However, the literal count of solution 9 (which is also PETRIFY's solution) is 15, which is lower than our cost function suggests — in fact, it is equal to that of solution 7. This shows that our cost function is not perfect, since it uses quite a rough estimate of complexity, not taking the context signals into account. However, it worked well in the other cases and it is not trivial to significantly improve it without introducing a considerable time overhead. The worst-case input-to-output delay for this solution is 1.3ns ($Laf^+ \to Ar^- \to csc^+ \to Lr^-$).

Note that our combined framework explored quite a large design space and produced a wide range of solutions, allowing the designer to exploit the speed/area trade-off and make an informed choice about which of them is the most appropriate for a given application (or allow the tool to chose the transformation using the cost function).

6 Usage

In this Section the usage of CONFRES is described, which include the command-line parameters, the interaction with the tool and an example to demonstrate the resolution process.

6.1 Command-line parameters

Usage:
ConfRes [-h]
or
ConfRes [options] [infile]

Infile is a Signal Transition Graph described in *astg* format (SIS compatible)

Options (may be given in any order)

-[α_1,α_2,α_3] α-parameters for cost function (default [1,1,1]).

-a Use only all minimal solutions in Clp (default all solutions).

-I Depict unfolding prefix with implicit places (default without implicit places).

-i Depict unfolding prefix with implicit conditions (default without implicit conditions).

-h Help mode, print the usage.

-R Usage with -S option to include concurrency reduction in addition to signal insertions. (default signal insertion).

-S Semi-automated resolution mode. In this mode transformations (signal insertions) are pre-computed and sorted according to the cost function. Note that this type of resolution is experimental and might contain invalid transformation. (default manual resolution).

-U USC property mode (default CSC property).

6.2 Interaction with ConfRes

The manual resolution works as shown in the process in Figure 9. The designer can view the cores and the height map, from which a set of intersecting cores of interest can be selected and visualised. In this way the designer is able to look into the cause of the CSC conflicts and decide how to resolve them. After examination the designer is free to choose a suitable location for a transformation, either signal insertion or concurrency reduction. In the case of signal insertion the phase two (insertion of the counterpart transition) has to be performed. The tool updates the height map and calculates constraints on the insertion of the counterpart transition, such that the designer can examine, in the same way as previously, where to insert the transition.

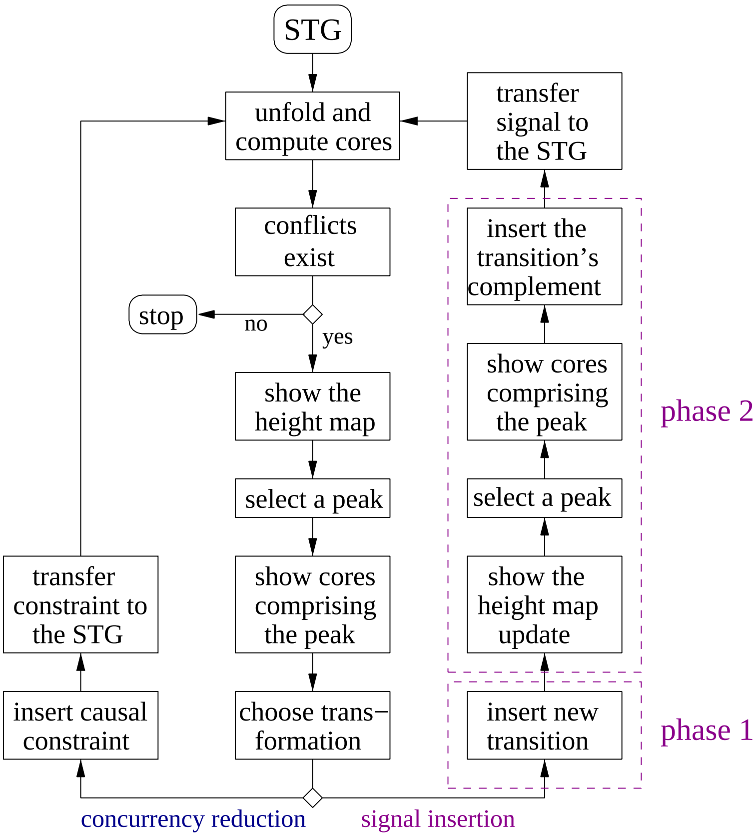


Figure 9: The resolution process of encoding conflicts

The semi-automated resolution works in a similar way. At every stage of the resolution process possible transformation are pre-computed, which are sorted according to the cost function, and act as a guideline for the designer. Note that for signal insertion the transformations include the insertions of both phases. The examination and visualisation of cores works in a same way as in the manual resolution, except for signal insertion phase one and two are joined together.

ConfRes is command-line driven and it uses menus for the interaction with the user. The main menu, shown in Figure 11 (a), is used for transformations. This menu is organised as follow. Option 1-3 prints out information about the number of cores, its distribution and the high-density areas. The cores and the height map can be depicted with option 4 and 5,

```
Main menu phase 1/2:
  please select:
    (0) quit

    (1) print statistics
    (2) print cores
    (3) print height distribution

    (4) depict all cores
    (5) depict height map

    (6) select events + solving
    (7) insert transition (without selection)

    (8) undo phase 1 / undo previous step

enter selection (0-4):
```

(a) main menu

```
Selection menu phase 1/2:
  please select:
    (0) quit
    (1) depict selected cores
    (2) depict height map of selected cores
    (3) insert transition
    (4) make new selection
enter selection (0-4):
```

(b) selection menu

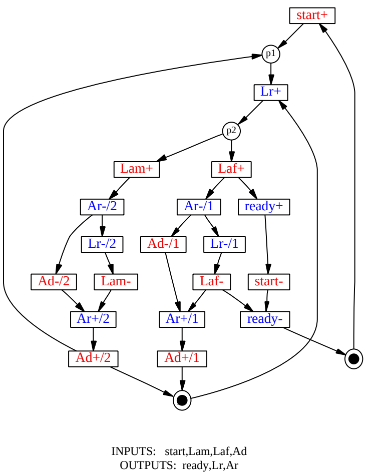
Figure 10: Interaction with ConfRes

respectively. The tool produces a postscript file which is automatically displayed. The file name combines the specification name and the stage, the phase and whether it is a core or height map. A transformation can be done by selecting a 'peak' (option 6) or directly (option 7). The former bring up the selection menu, shown in Figure 11 (b). In this menu the cores and the height map corresponding to the selection can be depicted by option 1 and 2, respectively, a transition can be inserted using option 3 and a new selection can be made by option 4. In the main menu and the selection menu the tool can be terminated by choosing option 0. In the semi-automatic mode (-S option) the option 6 in the main menu pre-computes possible transformations (if -R with -S option is used than also concurrency reduction is employed in addition to signal insertion).

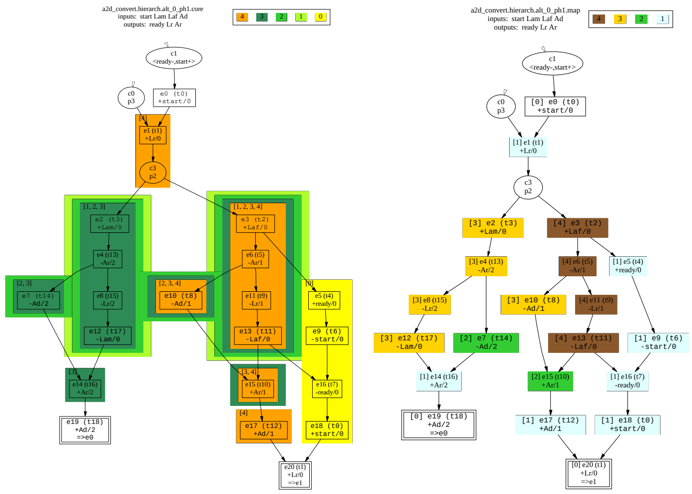
6.3 Example of usage

The top level controller of the AD converter presented in section 5.4 is used to demonstrate the usage of the tool ConfRes. The STG in Figure 11(a) does not satisfy the CSC property and the designer would like to intervene in the refinement process. (This example can be found in the "example" directory). The core representation in Figure 11(b) shows the designer the cause of the CSC conflicts. Additionally, the designer can examine the cores, e.g. by selecting the type I (core 0 and 4) and type II (core 1-3) cores and depict them separately. This feature is useful to distinguish set of cores which are related, e.g. cores 1-3 consist of extensions of core 1.

The height map shows the distribution of the core overlaps with the highest peak labeled by "4" corresponding to four cores (core 1-4). The pre-computed transformations are presented in Figure 11(c) showing five forward concurrency reductions (0-2, 9 and 10) and six signal insertions (3-8) with their corresponding cost functions. Note that the transformations are



(a) initial STG



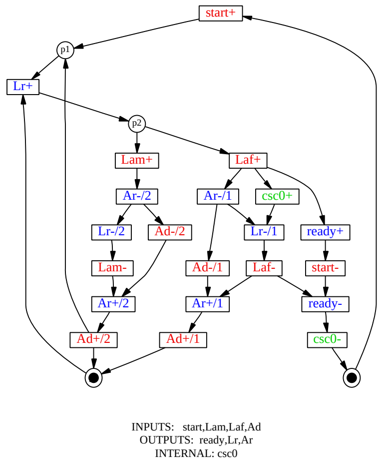
core map height map

(b) phase one

transformations:
- cost [cost: 1*deltaOmega,1*deltaLogic,1*deltaCore]
- signal insertion:
  * split: ->e (before e) , e-> (after e)
  * concurrent: e1 -> e2 (between e1 and e2)
- concurrency reduction: e1 ->> e2 (arc from e1 to e2)
  * / (forward)
  * \ (backward)

0: [-2: 0,2,-4] / e5 ->> e11
1: [-1: 1,2,-4] / e5 ->> e6
2: [-1: 3,0,-4] / e9 ->> e11
3: [-1: 1,3,-4] ph1 e3 -> e11, ph2 ->e16
4: [0: 3,2,-5] ph1 e3->, ph2 ->e16
5: [0: 2,3,-5] ph1 ->e6, ph2 ->e16
6: [1: 3,3,-5] ph1 e3->, ph2 e9->
7: [1: 2,4,-5] ph1 e3->, ph2 e5 -> e16
8: [1: 2,4,-5] ph1 ->e11, ph2 ->e16
9: [2: 4,2,-4] / e9 ->> e6
10: [2: 3,0,-1] / e10 ->> e11

(c) possible transformations



(d) final STG after insertion No 3

Figure 11: Interaction with ConfRes: an example

sorted according to the cost function, where $\alpha_1 = \alpha_2 = \alpha_3 = 1$. The designer is free to select a transformation, for example the transformation 3 is selected. The resulting conflict-free STG is depicted in Figure 11(d) with the new inserted signal *csc*. The resulting STG can be synthesised by either PETRIFY or VERISAT.

7 Bug report

Please submit bug reports to A.A.Madalinski@ncl.ac.uk with the STG and a short description of the problem.

References

[1] Tam-Anh Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications.* PhD thesis, MIT Laboratory for Computer Science, June 1987.

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

[3] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alexandre Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. In *XI Conference on Design of Integrated Circuits and Systems*, Barcelona, November 1996.

[4] D. A. Edwards and W. B. Toms. *Design, Automation and Test for Asynchronous Circuits and Systems.* Information society technologies (IST) programme concerted action thematic network contract, second edition, February 2003.

[5] V Khomenko. *Model Checking Based on Petri Net Unfolding Prefixes.* PhD thesis, University of Newcastle upon Tyne, 2002.

[6] V Khomenko. *Punf Documentation and User Guide*, June 2002.

[7] V. Khomenko, M. Koutny, and A. Yakovlev. Detecting State Coding Conflicts in STGs Using Integer Programming. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 338–345. IEEE Computer Society Press, 2002.

[8] V. Khomenko, M. Koutny, and A. Yakovlev. Detecting State Coding Conflicts in STGs Unfoldings using SAT. In *Int. Conf. on Application of Concurrency to System Design*, pages 51–60. IEEE Computer Society Press, June 2003.

[9] V. Khomenko, A. Madalinski, and A. Yakovlev. Resolution of Encoding Conflicts by Signal Insertion and Concurrency Reduction Based on STG Unfoldings. Technical Report CS-TR 858, Department of Computing Science, University of Newcastle upon Tyne, September 2004.

[10] D. J. Kinniment, B. Gao, A. Yakovlev, and F. Xia. Towards asynchronous A-D conversion. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 206–215. IEEE Computer Society Press, 1998.

[11] E. Koutsofios and S. North. *Dot User's Manual*, 2002.

[12] A. Madalinski. ConfRes: Interactive Coding Conflict Resolver based on Core Visulation. In *Int. Conf. on Application of Concurrency to System Design*. IEEE Computer Society Press, June 2003.

[13] A. Madalinski, A. Bystrov, V. Khomenko, and A. Yakovlev. Visualization and Resolution of Coding Conflicts in Asynchronous Circuit Design. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 926–931. IEEE Computer Society Press, March 2003.