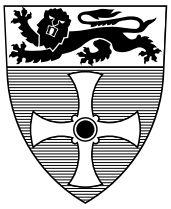
UNIVERSITY OF
NEWCASTLE



University of Newcastle upon Tyne

COMPUTING SCIENCE

Interactive Resolution of Encoding Conflicts in Asynchronous Circuits Based on STG Unfoldings

A.Madalinski, V. Khomenko and A. Yakovlev

TECHNICAL REPORT SERIES

No. CS-TR-944 Febuary, 2006

TECHNICAL REPORT SERIES

No. CS-TR-944 **Febuary, 2006**

Interactive Resolution of Encoding Conflicts in Asynchronous Circuits Based on STG Unfoldings

Agnes Madalinski, Victor Khomenko and Alex Yakovlev

Abstract

The synthesis of asynchronous circuits from STGs involves the resolution of encoding conflicts by means of refining the STG specification. The refinement process is generally done automatically using heuristics and offers little or no feedback to the designer making it difficult to intervene. Better synthesis solutions are obtained by involving human knowledge into the process. A framework is presented for an interactive refinement aimed to help the designer. It is based on the visualisation of several types of conflict cores, showing the cause of the encoding conflicts, which are presented at the level of finite and complete prefixes of the STG unfolding.

Printed and published by the University of Newcastle upon Tyne,
Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.

Bibliographical details

MADALINSKI, A., KHOMENKO, V., YAKOVLEV, A.

Interactive Resolution of Encoding Conflicts in Asynchronous Circuits Based on STG Unfoldings
[By] A, Madalinski, V. Khomenko, A. Yakovlev.

Newcastle upon Tyne: University of Newcastle upon Tyne: Computing Science, 2006.

(University of Newcastle upon Tyne, Computing Science, Technical Report Series, No. CS-TR-944)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE
Computing Science. Technical Report Series. CS-TR-944

Abstract

The synthesis of asynchronous circuits from STGs involves the resolution of encoding conflicts by means of refining the STG specification. The refinement process is generally done automatically using heuristics and offers little or no feedback to the designer making it difficult to intervene. Better synthesis solutions are obtained by involving human knowledge into the process. A framework is presented for an interactive refinement aimed to help the designer. It is based on the visualisation of several types of conflict cores, showing the cause of the encoding conflicts, which are presented at the level of finite and complete prefixes of the STG unfolding.

About the author

Victor Khomenko Obtained MSc with distinction in Computer Science, Applied Mathematics and Teaching of Mathematics and Computer Science in 1998 from Kiev Taras Shevchenko University, and PhD in Computing Science in 2003 from University of Newcastle upon Tyne. From September 2005 Victor is a Royal Academy of Engineering/EPSRC Post-doctoral Research Fellow, working on the DAVAC project.

Alex Yakovlev is a Professor of Computer System Design in the School of Electrical, Electronic and Computer Engineering, at the University of Newcastle.

Suggested keywords

LOGIC SYNTHESIS,
COMPLETE STATE CODING,
CSC CONFLICT,
NORMALCY,
ASYNCHRONOUS CIRCUITS,
SIGNAL TRANSITION GRAPH,
STG, PETRI NET UNFOLDING

Interactive Resolution of Encoding Conflicts in Asynchronous Circuits Based on STG Unfoldings

Agnes Madalinski Victor Khomenko Alex Yakovlev

***Abstract*— The synthesis of asynchronous circuits from STGs involves the resolution of encoding conflicts by means of refining the STG specification. The refinement process is generally done automatically using heuristics and offers little or no feedback to the designer making it difficult to intervene. Better synthesis solutions are obtained by involving human knowledge into the process. A framework is presented for an interactive refinement aimed to help the designer. It is based on the visualisation of several types of conflict cores, showing the cause of the encoding conflicts, which are presented at the level of finite and complete prefixes of the STG unfolding.**

***Index Terms*— Logic synthesis, complete state coding, CSC conflict, normalcy, asynchronous circuits, signal transition graph, STG, Petri net unfolding.**

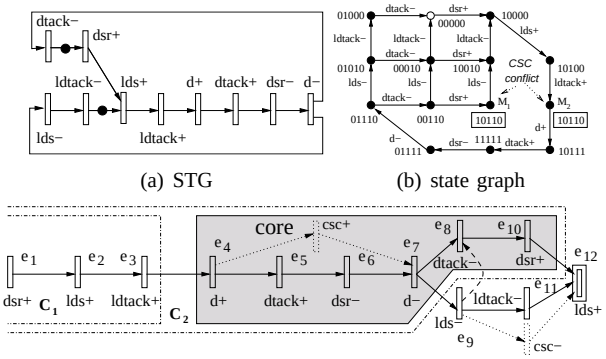
I. Introduction

SIGNAL Transition Graphs, or STGs [1], [2], are widely used for specifying the behaviour of asynchronous control circuits. They are interpreted Petri nets in which transitions are labelled with the rising and falling edges of circuit signals. The key steps in this method are the generation of a state graph, which is a binary encoded reachability graph of the underlying Petri net, and deriving Boolean equations for the non-input signals via their next-state functions obtained from the state graph.

During the logic synthesis, resolution of encoding conflicts is a fundamental problem. The implementability of the STG, as well as the complexity, monotonicity and robustness of the derived functions depends on this. The resolution process is generally done automatically using heuristics and it can often produce sub-optimal solutions. Manual intervention by the designer may be required to obtain optimal solutions. This is particularly critical for interface controllers [2], which determin the performance of the whole system. According to a practising designer [3], a synthesis tool should offer a way for the user to understand the characteristic patterns of a circuit's behaviour and the cause of each encoding conflict, in order to allow the designer to manipulate the model interactively.

At the level of the STG model the states are represented by markings; however, at the circuit level only the corresponding binary codes are represented. An encoding conflict may arise when two semantically different states of an STG have equal binary codes and hence are indistinguishable at the circuit level. More precisely, two distinct reachable states of an STG are in *Unique State Coding (USC) conflict* if they are assigned the same binary codes; they are in *Complete State Coding (CSC) conflict* if in addition different sets of output signals are enabled at them. An STG satisfies the *USC/CSC property* if no two of its reachable states are in USC/CSC conflict. The absence of CSC conflicts is a necessary condition for the implementability of the STG [1], [2]. Fig. 1(b) shows the state graph of the STG in Fig. 1(a) with a CSC conflict between states M_1 and M_2.

Another type of encoding conflict is defined by the property of *normalcy* [4], which is a necessary condition for an STG to be implementable as a logic circuit built of monotonic gates (perhaps, with output inverters). Such a circuit does not use inverters (which are assumed to have negligible delays [2]) at the gate inputs in order to guarantee the absence of hazards. Formally, an STG satisfies the *positive normalcy (p-normalcy)* (respectively, *negative normalcy (n-normalcy)*) condition w.r.t. an output signal z if for every pair of reachable states s' and s'' such that the encoding of s' is not smaller (as a bitvector) than the encoding of s'', the value of the next-state function of z at s' is not smaller (respectively, not greater) than that at s''. An STG is *normal* if each output signal is either p-normal or



(a) STG (b) state graph

(c) unfolding prefix with corresponding conflict core

$$\begin{aligned}[lds] &= csc \cdot dsr + d\\ [dtack] &= d\\ [d] &= dsr \cdot d + ldtack \cdot csc\\ [csc] &= csc \cdot \overline{d} + \overline{ldtack}\end{aligned}$$

(d) signal insertion

$$\begin{aligned}[lds] &= dsr \cdot (lds + \overline{ldtack}) + d\\ [dtack] &= dtack \cdot lds + d\\ [d] &= ldtack \cdot dsr \cdot lds\end{aligned}$$

(e) concurrency reduction

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

Fig. 1. VME bus controller: read cycle (the order of signals in the binary encodings is: *dsr, ldtack, dtack, lds, d, csc*).

A. Madalinski and A. Yakovlev are affiliated with School of Electrical, Electronic and Computer Engineering, University of Newcastle upon Tyne, UK. Their research is supported by the EPSRC grant GR/S12036 (STELLA). E-mail: {A.A.Madalinski,Alex.Yakovlev}@ncl.ac.uk

V. Khomenko is a Royal Academy of Engineering/EPSRC Post-Doctoral Research Fellow. He is affiliated with School of Computing Science, University of Newcastle upon Tyne, UK. His research is supported by Royal Academy of Engineering/EPSRC grant EP/C53400X/1 (DAVAC). E-mail: Victor.Khomenko@ncl.ac.uk

n-normal. Note that normalcy implies CSC [4].

In this paper, a framework for visualisation and resolution of encoding conflicts is presented. It extends [5] by incorporating a wider class of encoding conflicts and the use of concurrency reduction in addition to signal insertion for resolving them. It is based on visualisation of *cores*, i.e., sets of transitions 'causing' one or more encoding conflicts, enabling the designer to comprehend their cause. It works on the level of a finite and complete prefix of the STG unfolding and avoids the explicit enumeration of encoding conflicts.

A *finite and complete unfolding prefix* [6] of an STG is a finite acyclic net which implicitly represents all the STG's reachable states together with transitions enabled at those states. Intuitively, it can be obtained through *unfolding* the STG, by successive firings of transitions, under the following assumptions: (a) for each new firing a fresh transition (called an *event*) is generated; (b) for each newly produced token a fresh place (called a *condition*) is generated. The unfolding is infinite whenever the STG has an infinite run; however, if the STG 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 beyond which it is not generated) without loss of information, yielding a finite and complete prefix.

Complete prefixes are often exponentially smaller than the corresponding state graphs, especially for highly concurrent Petri nets, because they represent concurrency directly rather than by multidimensional 'diamonds' as it is done in state graphs. For example, if the original Petri net consists of 100 transitions which can fire once in parallel, the state graph will be a 100-dimensional hypercube with 2^{100} nodes, whereas the complete prefix will coincide with the net itself. Therefore, unfolding prefixes are well-suited for both visualisation of an STG's behaviour and alleviating the state space explosion problem. The experiments conducted in [7]–[9] have shown that for practical STGs unfolding prefixes are often much smaller than the corresponding state spaces. This can be explained by the fact that practical STGs usually contain a lot of concurrency but relatively few choices, and thus the prefixes are in many cases not much bigger then the STGs themselves. As a result, unfolding-based methods had a clear advantage over the BDD-based techniques both in terms of memory usage and running time. Such methods are also advantageous for interactive procedures and manual interventions based on visualisation.

The resolution of encoding conflicts by signal insertion is illustrated in Fig. 1(c), where the new signal *csc* is helping to distinguish between the states involved in the encoding conflict. (Intuitively, insertion of signals introduces additional memory into the circuit, helping to trace the current state.) It was inserted concurrently to existing transitions in order to minimise the latency, and in such a way that the 'external' behaviour of the STG does not change. Alternatively, the encoding conflict can be solved by reducing the concurrency between lds^- and $dtack^-$ (as shown by the dashed arc in Fig. 1(c)) so that state M_1 is removed from the reachability graph shown in Fig. 1(b), which in turn resolves the encoding conflict. The logic equations corresponding to these solutions are shown in Fig. 1(d,e).

Based on the concept of cores a refinement procedure is also presented. It involves the transformation of the STG into a conflict free one either by the introduction of auxiliary signals or by concurrency reduction, exploring thus a larger design space. A manual refinement is used with the aim of obtaining an optimal solution within the design constraints. The refinement involves the analysis of encoding conflicts by the designer, who can choose an appropriate transformation. The method can also work in a completely automatic or semi-automatic manner, making it possible for the designer to see what is happening and intervene at any stage during the encoding conflict resolution process. The proposed method has been implemented in the CONFRES tool [10].

II. REPRESENTATION OF ENCODING CONFLICTS

At the level of unfoldings, CSC conflicts can be compactly represented using conflict *cores* [5]. This paper extends that technique to other classes of encoding conflicts.

Due to its structural properties (such as acyclicity), the reachable states of an STG can be represented using *configurations* of its unfolding. 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 Fig. 2(b) $\{e_1,e_3,e_4\}$ is a configuration whereas $\{e_1,e_2,e_3,e_4\}$ and $\{e_3,e_4\}$ are not (the former includes a structural conflict between the events e_1 and e_2, while the latter does not include e_1, a causal predecessor of e_3 and e_4). Intuitively, a configuration is a partial-order execution, i.e., an execution where the order of firing of concurrent events is not important; e.g., the configuration $\{e_1,e_3,e_4,e_6,e_8\}$ corresponds to the following three totally ordered executions: $e_1e_3e_4e_6e_8$, $e_1e_4e_3e_6e_8$ and $e_1e_4e_6e_3e_8$.

A. Encoding conflicts in a prefix

Encoding conflicts can be classified as USC, CSC, p-normalcy or n-normalcy conflicts. Let two states be in an $\mathcal{X}$ *conflict*, where $\mathcal{X}$ is either USC, CSC, p-normalcy or n-normalcy. An $\mathcal{X}$ conflict can be represented as an unordered $\mathcal{X}$ *conflict pair* of configurations $\langle C_1,C_2\rangle$ whose final states are in an $\mathcal{X}$ conflict. For example, in Fig. 1(c) $\langle C_1,C_2\rangle$ is a CSC conflict pair. In [6], [8] two techniques for detecting USC, CSC and normalcy conflicts (based, respectively, on integer programming and SAT) were proposed. Essentially, they allow for efficiently finding such conflict pairs in STG unfolding prefixes.

The set of all $\mathcal{X}$ conflict pairs may be quite large, e.g., due to the following 'propagation' effect: if $\langle C_1,C_2\rangle$ is an $\mathcal{X}$ conflict pair and C_1 and C_2 can both be expanded by the same event e, then $\langle C_1\cup\{e\},C_2\cup\{e\}\rangle$ is often an $\mathcal{X}$ conflict pair as well. For example, in Fig. 2(c) $\langle\{a_0^+\},\{a_0^+,b_0^+,b_1^+,b_0^-,b_1^-\}\rangle$ is a CSC conflict pair, and adding, e.g., the event labelled c_0^+ to both these configurations leads to a new CSC conflict pair $\langle\{a_0^+,c_0^+\},\{a_0^+,b_0^+,b_1^+,b_0^-,b_1^-,c_0^+\}\rangle$. Therefore, it is desirable to reduce the number of conflict pairs which need to be

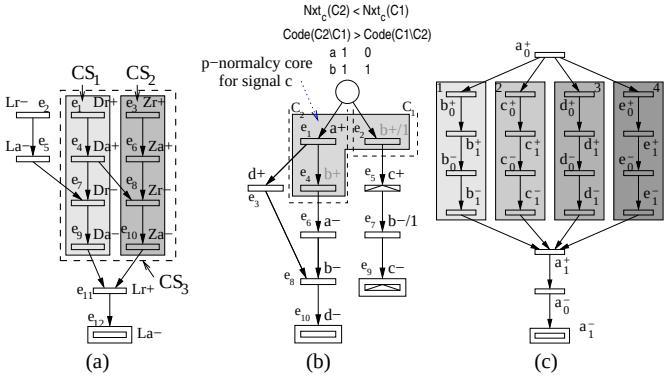


Fig. 2. Visualisation examples of $\mathcal{X}$ cores.

considered as follows. An $\mathcal{X}$ conflict pair $\langle C_1, C_2\rangle$ is called *concurrent* if $C_1 \not\subseteq C_2$, $C_2 \not\subseteq C_1$ and $C_1 \cup C_2$ is a configuration. Below is a slightly modified version of propositions proven in [6].

Proposition 1: Let $\langle C_1, C_2\rangle$ be a concurrent $\mathcal{X}$ conflict pair. Then $C = C_1 \cap C_2$ is a configuration such that $\langle C, C_1\rangle$ or $\langle C, C_2\rangle$ is an $\mathcal{X}$ conflict pair.

Thus concurrent $\mathcal{X}$ conflict pairs are 'redundant' and should not be considered. The remaining $\mathcal{X}$ conflict pairs can be classified as follows:

- $\mathcal{X}$ **conflicts of type I** are such that either $C_1 \subset C_2$ or $C_2 \subset C_1$ (i.e., configurations C_1 and C_2 are ordered).
- $\mathcal{X}$ **conflicts of type II** are such that $C_1 \setminus C_2 \neq \emptyset \neq C_2 \setminus C_1$ and there exist $e' \in C_1 \setminus C_2$ and $e'' \in C_2 \setminus C_1$ such that e' and e'' are in structural conflict.

An example of a type I CSC conflict is illustrated in Fig. 1(c) and an example of a type II p-normalcy conflict for signal c is illustrated in Fig. 2(b).

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

Definition 1 ($\mathcal{X}$ core): Let $\langle C_1, C_2\rangle$ be an $\mathcal{X}$ conflict pair. The corresponding $\mathcal{X}$ *conflict set* is defined as $\mathcal{CS} = C_1 \triangle C_2$, where $\triangle$ denotes the symmetric set difference. $\mathcal{CS}$ is an $\mathcal{X}$ *core* if it cannot be represented as the union of several disjoint $\mathcal{X}$ conflict sets. An $\mathcal{X}$ conflict set is of type I/II if the corresponding $\mathcal{X}$ conflict pair is of type I/II, respectively. $\Diamond$

For example, in Fig. 2(a) the CSC conflict sets $\mathcal{CS}_1$ and $\mathcal{CS}_2$ are CSC cores, whereas $\mathcal{CS}_3$ is not, because $\mathcal{CS}_3 = \mathcal{CS}_1 \cup \mathcal{CS}_2$. The CSC core corresponding to the CSC conflict pair shown in Fig. 1(c) is $\{e_4 - e_8, e_{10}, e_{12}\}$. (Note that for an $\mathcal{X}$ conflict pair $\langle C_1, C_2\rangle$ of type I, such that $C_1 \subset C_2$, the corresponding $\mathcal{X}$ core is simply $C_2 \setminus C_1$.) The type II p-normalcy conflict core for signal c in Fig. 2(b) corresponding to the p-normalcy conflict pair $\langle \{e_2\}, \{e_1, e_4\}\rangle$ is $\{e_1, e_2, e_4\}$.

Every $\mathcal{X}$ conflict set $\mathcal{CS}$ can be partitioned into $C_1 \setminus C_2$ and $C_2 \setminus C_1$, where $\langle C_1, C_2\rangle$ is an $\mathcal{X}$ conflict pair corresponding to $\mathcal{CS}$. Moreover, if $\mathcal{CS}$ is of type I then one of these partitions is empty, while the other is $\mathcal{CS}$ itself. In this case $Code(C_2 \setminus C_1) = Code(C_2) - Code(C_1)$, where $C_1 \subset C_2$. If $\langle C_1, C_2\rangle$ is a type II conflict pair then $Code(C_2 \setminus C_1) = Code(C_2) - Code(C_1 \cap C_2)$.

Important properties of $\mathcal{X}$ conflicts are described below:

- USC/CSC conflict: $Code(C_1 \setminus C_2) = Code(C_2 \setminus C_1)$, and if $\mathcal{CS}$ is of type I then $Code(\mathcal{CS}) = \mathbf{0}$. This suggests that $\mathcal{CS}$ can be eliminated, for example, by the introduction of an auxiliary signal, in such a way that one of its transitions is inserted into $\mathcal{CS}$, as this would violate the stated property, as illustrated in Fig. 1(c).
- Normalcy conflict: $Code(C_1 \setminus C_2)\ \mathcal{R}\ Code(C_2 \setminus C_1)$ and if $\mathcal{CS}$ is of type I then $Code(\mathcal{CS})\ \mathcal{R}\ \mathbf{0}$, where $\mathcal{R} \in \{\leq, \geq\}$. An example of a type II p-normalcy core for signal c is presented in Fig. 2(b). Note that if the encodings are equal then $\mathcal{CS}$ corresponds to a CSC conflict. To resolve the conflicts caused by $\mathcal{CS}$ the encodings must be made incomparable. This can be done, for example, by the introduction of an auxiliary signal, in such a way that one of its transitions with the appropriate polarity is inserted into $\mathcal{CS}$, as this would violate the stated property, as illustrated in Fig. 4.

It is often the case that the same $\mathcal{X}$ conflict set corresponds to different $\mathcal{X}$ conflict pairs. For example, the STG of the unfolding prefix shown in Fig. 2(c) has four concurrent branches with a CSC core in each of them. Due to the above mentioned 'propagation' effect, there are altogether 888 CSC conflict pairs, a full list of which is clearly too long for the designer to cope with. Despite this, there are only four CSC cores, as shown in Fig. 2(c). (Note that there are 15 CSC conflict sets, which can be obtained by uniting these cores.)

B. Visualisation of encoding conflicts

The visualisation of encoding conflicts is based on showing the designer the $\mathcal{X}$ cores in the STG's unfolding prefix. Since every element of an $\mathcal{X}$ core is an instance of the STG's transition, the $\mathcal{X}$ cores can be easily mapped from the prefix to the STG, and thus unfolding prefixes can be used for choosing an appropriate STG transformation. $\mathcal{X}$ cores are crucial for the resolution of $\mathcal{X}$ conflicts: eliminating a core results in eliminating all the associated $\mathcal{X}$ conflicts.

$\mathcal{X}$ conflicts can be visualised by $\mathcal{X}$ cores. However, certain normalcy violations admit a simpler visualisation. In particular, sometimes it is possible to use simple heuristics based on *triggers* (i.e., transitions firing of which can enable a given signal) to establish a normalcy violation for a non-input signal. Thus, showing the triggers involved would be another type of visualisation. The visualisation of normalcy conflicts is described in detail below.

It is often the case that $\mathcal{X}$ cores overlap. In order to minimise the number of transformations, and thus the area and latency of the circuit, it is advantageous to transform a specification in such a way that as many cores as possible are eliminated. That is, a transformation should be performed at *the intersection of several cores* whenever possible.

To assist the designer in exploiting $\mathcal{X}$ core overlaps, another key feature of our method, viz. the *height map* showing the quantitative distribution of the $\mathcal{X}$ cores, is employed in the visualisation process. The events located in $\mathcal{X}$ cores are highlighted by shades of colours. The shade depends on the

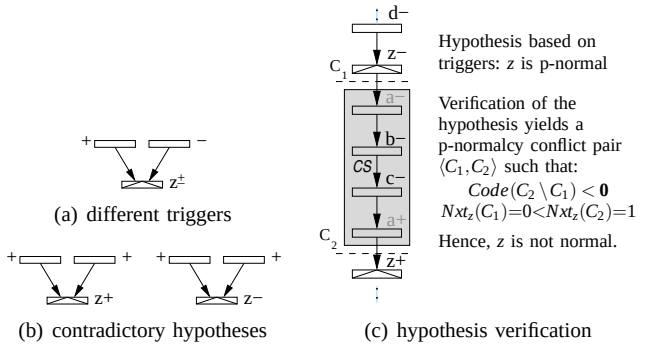


Fig. 3. Visualisation of normalcy violations.

altitude of an event, i.e., on the number of cores to which it belongs. (The analogy with a topographical map showing altitudes may be helpful here.) The greater is the altitude, the darker is the shade. 'Peaks' with the highest altitude are good candidates for transformation, since they correspond to the intersection of maximum number of cores. The height map corresponding to the cores in Fig. 11(a) is shown in Fig. 11(b). Peaks with the highest altitude are labelled with 'A4', which corresponds to the overlap of four cores.

Normalcy conflicts visualisation

In [7], [8], when checking the normalcy, a hypothesis about the normalcy type of each output and internal signal is made. It is based on the triggers of the events labelled by such a signal. If a z-labelled event has triggers with the same (respectively, opposite) sign as the event itself then a hypothesis is made that z is p-normal (respectively, n-normal).

With the exception of certain pathological cases (e.g., when some signal is set at the beginning and is never reset), for each output or internal signal at least one such hypothesis can be made. However, it is sometimes possible to make contradictory hypotheses. In such a case the STG is not normal. Furthermore, the violation of CSC implies a violation of normalcy. Thus, normalcy violations can be caused by the following factors:

Different triggers If an event has triggers with different signs, then the corresponding signal is neither p-normal nor n-normal, see Fig. 3(a).

Contradictory hypotheses Contradictory hypotheses based on triggers can be made about the normalcy type of a signal, so the signal is neither n-normal nor p-normal. This is the case if a signal transition has triggers with the same sign (which means that the signal cannot be n-normal) whilst another instance of this signal has triggers with the opposite sign (i.e., the signal cannot be p-normal), see Fig. 3(b).

CSC conflict There is a CSC conflict.

Hypothesis verification If none of the above holds, the hypotheses about the normalcy type of each signal are verified using the definition, see Fig. 3(c).

Hence normalcy violations can be visualised by either highlighting events together with their triggers (see Fig. 3(a,b)), CSC cores, or normalcy cores, where the transition instances corresponding to signals x such that $Code_x(\mathcal{CS}) = 0$ are faded out. Moreover, events corresponding to signals for which the normalcy core is built are drawn as patterned bars. For example, in the type I core shown in Fig. 3(c) $Code_a(\mathcal{CS}) = 0$, and so a is faded out.

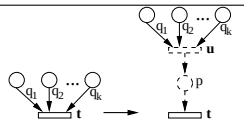
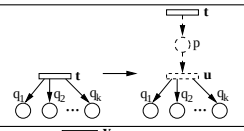
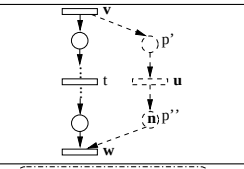
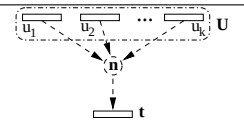
transformation		notation
insertion before t		$\wr t$
insertion after t		$t\wr$
concurrent insertion		$v \xrightarrow{n} w$
concurrency reduction		$U \xrightarrow{n} t$

TABLE I
NET TRANSFORMATIONS.

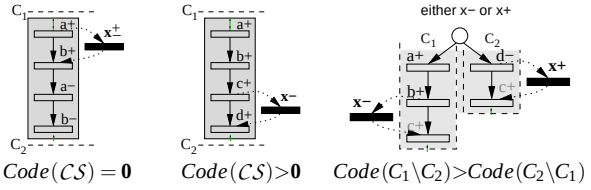
III. RESOLUTION OF ENCODING CONFLICTS

$\mathcal{X}$ conflicts can be efficiently resolved by adding auxiliary signals and by concurrency reduction. 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 in this paper, which uses both signal insertion and concurrency reduction to eliminate cores and the associated encoding conflicts. This allows to explore a larger design space.

A. Core elimination: a general approach

The resolution of encoding conflicts depends on their type. In the case of normalcy violations it is necessary to make $Code(C_1 \setminus C_2)$ and $Code(C_2 \setminus C_1)$ incomparable, whereas in the USC/CSC violations it is necessary to make $Code(C_1 \setminus C_2)$ and $Code(C_2 \setminus C_1)$ distinct.

The strategies for eliminating $\mathcal{X}$ cores using signal insertion are shown in Fig. 4. Recall that each core can be partitioned into $C_1 \setminus C_2$ and $C_2 \setminus C_1$. If the encodings of $C_1 \setminus C_2$ and $C_2 \setminus C_1$ are equal then they must be made distinct by adding a transition into the core (its polarity is not important). In Fig. 4(a), the USC/CSC core $\mathcal{CS} = \{a^+, b^+, a^-, b^-\}$ can be eliminated by adding a transition into the core, which makes the conflicting states corresponding to C_1 and C_2 distinct.



(a) USC/CSC core (b) type I and II normalcy cores

Fig. 4. Strategies for eliminating $\mathcal{X}$ cores.

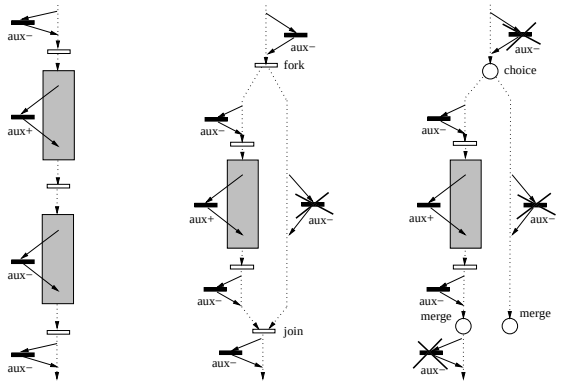
A normalcy core can be eliminated by incorporating a transition with a negative (respectively, positive) polarity into the part with the greater (respectively, smaller) encoding. If the normalcy core is of type I then one of these parts is empty, and the other is the core $\mathcal{CS}$. Thus, if $Code(\mathcal{CS}) > \mathbf{0}$ (respectively, $Code(\mathcal{CS}) < \mathbf{0}$) then a transition with a negative (respectively, positive) polarity is added. The polarity is important, because the additional bit of the encoding corresponding to the newly inserted signal should make the encodings incomparable, eliminating thus the core. The elimination of normalcy cores is schematically illustrated in Fig. 4(b): adding a transition x^- into the type I core $\mathcal{CS} = \{a^+, b^+, c^+, d^+\}$ makes the encodings of $C_1 \setminus C_2$ and $C_2 \setminus C_1$ incomparable. In type II normalcy cores either a negative auxiliary transition is added in the partition with the greater encoding or a positive auxiliary transition is added in the partition with the smaller encoding. Note that if $Code(C_1 \setminus C_2) < Code(C_2 \setminus C_1)$ in an n-normalcy core for a signal z than $Nxt_z(C_1) < Nxt_z(C_2)$, and consequentially, if $Code(C_1 \setminus C_2) < Code(C_2 \setminus C_1)$ in a p-normalcy core for a signal z than $Nxt_z(C_1) > Nxt_z(C_2)$. Moreover, the polarity of the inserted transitions should be such that the trigger-based hypotheses made about the normalcy type of the newly inserted signal are consistent.

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.

B. Signal insertion

Possible signal insertions (either sequential or concurrent) are shown in the first three parts of Table I. 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).

It is important to preserve the *consistency* when a new signal is inserted into the STG. An STG is *consistent* if, for every its signal z, the following two conditions are satisfied: (i) the first occurrence of z in the labelling of any firing sequence of the STG has the same sign (either rising of falling); and (ii) the transitions corresponding to the rising and falling edges of z alternate in any firing sequence of the STG. In this paper it is assumed that all the STGs considered are consistent.[1]



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

Fig. 5. Strategies for $\mathcal{X}$ core elimination.

An $\mathcal{X}$ core can be eliminated by inserting a transition of an auxiliary internal signal *aux*, say aux^+, somewhere in the core to destroy it. To preserve the consistency of the STG, the signal transition's counterpart aux^- must also be added to the specification *outside the core,* in such a way that it is neither concurrent with, nor in structural conflict with aux^+ (otherwise the STG becomes inconsistent). Another restriction is that an inserted signal transition cannot trigger an input signal transition. A transition can be inserted either by splitting an existing transition, or by inserting a transition concurrently with existing transitions (see Table I).

Fig. 5 shows the insertion possibilities in typical cases in STG specifications. $\mathcal{X}$ cores in sequence can be eliminated in a one-hot manner as depicted in Fig. 5(a). Each $\mathcal{X}$ core is eliminated by one signal transition, and its complement is inserted outside the $\mathcal{X}$ core, preferably into another non-adjacent one.[2] An STG that has an $\mathcal{X}$ core in one of the concurrent branches can also be tackled in a 'one-hot' way, as shown in Fig. 5(b). Note that in order to preserve the consistency the transition's counterpart cannot be inserted into the concurrent branch, but can be inserted before the fork transition or after the join one. In a branch which is in a structural conflict with another branch, the transition's counterparts must be inserted in the same branch somewhere between the choice and the merge points, as shown in Fig. 5(c).

Obviously, the described cases do not cover all possible situations or all possible insertions (e.g., a signal transition can sometimes be inserted before the choice point or after the merge point and its counterparts inserted into *each* branch, etc.), but they give an idea how $\mathcal{X}$ cores can be eliminated.

For example, the core in Fig. 1(c) can be eliminated by inserting a transition of a new signal, *csc*, somewhere in the core, e.g., concurrently to e_5 and e_6 between e_4 and e_7, and by inserting its complement outside the core, e.g., between e_9 and e_{12}. After transferring this signal into the STG, it satisfies

[1]The consistency of an STG can easily be checked during the process of building its finite and complete prefix [11].

[2]The union of two adjacent $\mathcal{X}$ cores is usually an $\mathcal{X}$ conflict set which will not be eliminated if both the transition and its counterpart are inserted into it.

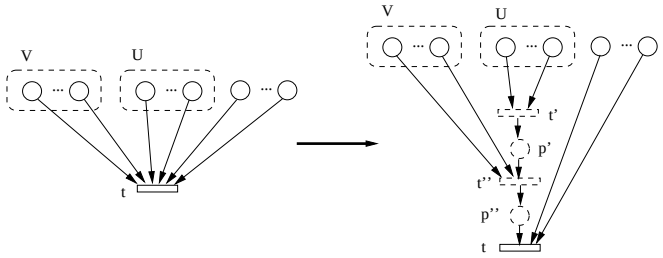


Fig. 6. Flip-flop insertion.

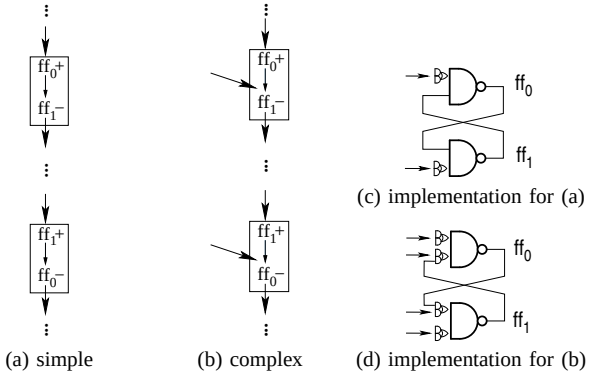


Fig. 7. Strategy for flip-flop insertion.

the CSC property. (Other ways of inserting a signal to resolve the encoding conflict are also possible.)

Flip-flop insertion

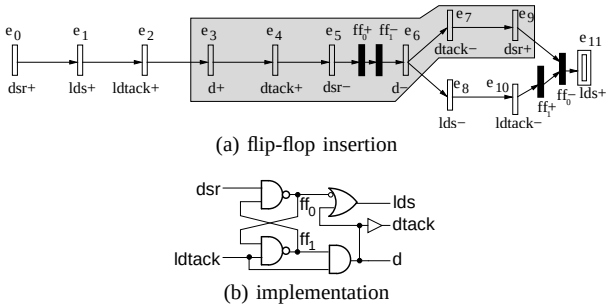
The insertion of a pair of sequential signals, $ff_0^{\pm}$ and $ff_1^{\mp}$, in the STG corresponds to adding a pair of gates to the circuit forming a flip-flop. It can play the role of a memory element allowing the circuit to distinguish between the conflicting states [4]. This type of insertion often results in simpler logics compared to the usual signal insertion, and thus gives the designer opportunities for potential improvements to the circuit.

The flip-flop insertion is illustrated in Fig. 6. Formally, given an STG Σ, one of its transitions t, and two disjoint sets of places $U \neq \emptyset$ and V such that $U \cup V \subseteq {}^\bullet t$, a *flip-flop insertion* $[U,V] \wr\wr t$ is defined as the transformation which:

- removes from Σ the arcs (p,t) for all $p \in U \cup V$;
- adds to Σ two new places p' and p'' and two new transitions t' and t'';
- adds to Σ the new arcs (p,t') for all $p \in U$ and (p,t'') for all $p \in V$, and the arcs (t',p'), (p',t''), (t'',p'') and (p'',t).

We will write $U \wr\wr t$ instead of $[U,\emptyset] \wr\wr t$; in such a case the transformation is called a *simple* flip-flop insertion, to distinguish it from a *complex* flip-flop insertion $[U,V] \wr\wr t$ where $V \neq \emptyset$.

Let t_1', t_1'' and t_2', t_2'' be the transitions added by a pair of flip-flop insertions, and ff_0 and ff_1 be a pair of new internal signals in the STG. If t_1' and t_2'' correspond to ff_0 and have the opposite signs, and t_1'' and t_2' correspond to ff_1 and have the opposite signs, then these flip-flop insertions are



(a) flip-flop insertion

(b) implementation

inputs: *dsr,ldtack;* **outputs:** *lds,d,dtack;* **internal:** ff_0, ff_1

Fig. 8. Example: flip-flop insertion.

complimentary. The complementary flip-flop insertions in the simple and complex cases, together with the corresponding implementations, are illustrated in Fig. 7 (note that additional logic is sometimes needed at the inputs of the flip-flop, as well as at the criss-cross inputs in the complex case).

A flip-flop insertion eliminating the core in Fig. 1(c) is shown in Fig. 8. Such a transformation, in general, should result in a complex flip-flop because ff_0^- has an additional trigger dsr^+. However, due to the fact that dsr^- is already a trigger of ff_0^+, the flip-flop does not need any additional control logic: it is set by *dsr* and reset by *ldtack*. The resulting implementation is depicted in Fig. 8(b).

C. Concurrency reduction

The concurrency reduction in the last part of Table I introduces a new causal constraint in form of a new place (with, perhaps, a token) between existing transitions. Formally, given an Petri net Σ, a set of its transitions $U \neq \emptyset$, its transition $t \notin U$ and $n \in \mathbb{N}$, a concurrency reduction $U \overset{n}{\dashrightarrow} t$ is defined as the transformation adding to Σ a new place p, which initially has n tokens, the arc (u,p) for each transition $u \in U$ and the arc (p,t). 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.

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 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 transitions can be used to eliminate 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 Fig. 1(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.

Two kinds of concurrency reduction based transformations for core elimination are described below (where h is the

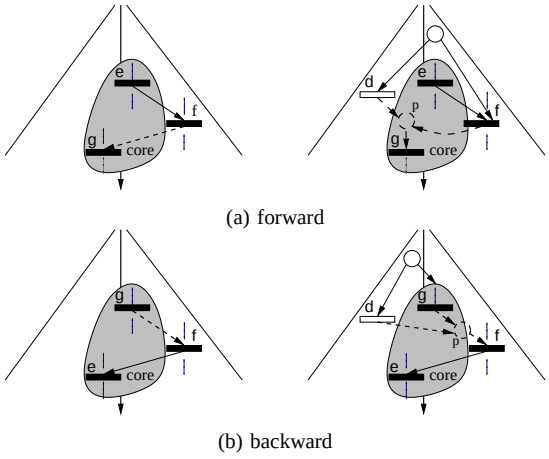


Fig. 9. X core elimination by concurrency reduction.

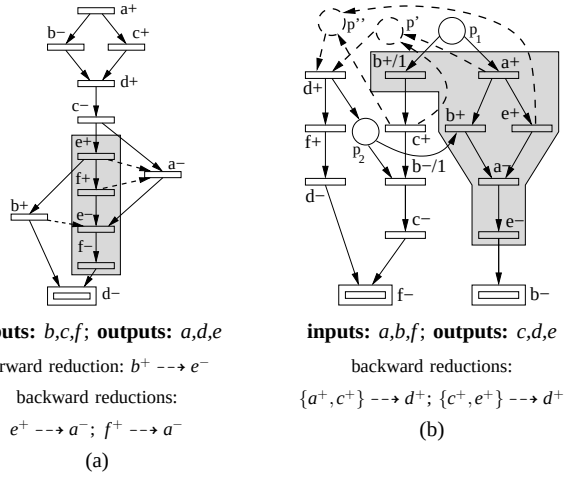


Fig. 10. Elimination of CSC cores.

mapping from the nodes of the prefix to the nodes of the STG).

Forward concurrency reduction illustrated in Fig. 9(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 Fig. 9(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 Fig. 10, where they are applied to cores of types I and II, respectively. In Fig. 10(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.

In Fig. 10(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.)

D. Validity of transformations

The notion of validity for signal insertion is relatively straightforward — one can justify such a transformation in terms of weak bisimulation, which is well-studied. For a concurrency reduction (or transformations in general), the situation is more difficult: the original and transformed systems are typically not even language-equivalent; deadlocks can disappear (e.g., the deadlocks in Dining Philosophers can be eliminated by fixing the order in which forks are taken); deadlocks can be introduced; transitions can become dead; even the language inclusion may not hold (some transformations, e.g., converting a speed-independent circuit into a delay-insensitive one [12] can increase the concurrency of inputs, which in turn *extends* the language).

In [13] we propose an elegant bisimulation-style notion which allows one to justify both reduction of concurrency for outputs and increase of concurrency for inputs, as well as signal insertion. We also propose there criteria which can be applied to check the validity of a concurrency reduction.

E. 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

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

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, $\Delta logic$, 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 [14] 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 encoding 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.

IV. Case study

In this section, three examples demonstrating the proposed combined framework for the resolution of encoding conflicts are discussed. The simulation times are obtained by the analog simulation using the AMS-0.35μ CMOS technology. The examples were chosen having in mind the importance of finding an optimal solution in terms of speed and area. These examples are small-size controllers, where the cost of a wrong design choice can be critical. This situation is different from large-size controllers for data processing blocks, where alternative approaches such as BALSA [15], structural methods [16] and direct mapping [17] are used.

A. AD converter

The example shown in Fig. 11 is a part of the A/D converter proposed in [18]. It contains two type I and three type II CSC cores shown in Fig. 11(a). The valid transformations are presented in the tables in Fig. 11(c,d), 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. The equations for most interesting of these solutions are shown in Fig. 11(e).

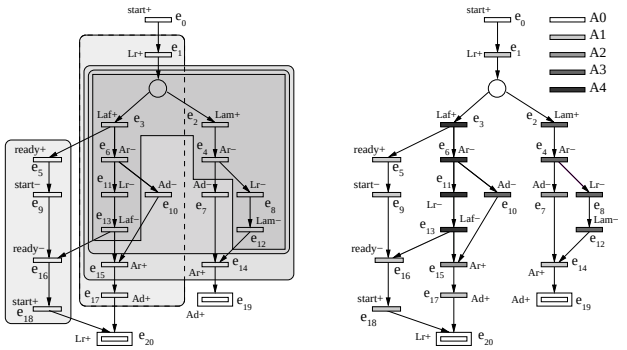
The events e_3, e_6, e_{11} and e_{13} comprise the highest peak, as each of them belongs to four cores. They can be eliminated by a forward concurrency reduction, since events e_5 and e_9 are concurrent to the events in the peak and the concurrency starts in the peak. The first four solutions in the table in Fig. 11(c) 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



(a) unfolding prefix with cores (b) height map

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

concurrency reduction				
#	causal constraint	Δω; Δ*logic*; Δ*cores*	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

(c) possible concurrency reductions

signal insertion					
#	phase 1	phase 2	Δω; Δ*logic*; Δ*cores*	lits	delay
6	$h(e_3) \overset{0}{\longrightarrow} h(e_{11})$	$\wr h(e_{16})$	1; 3; -5	16	1.2ns
7	$h(e_3)\wr$	$\wr h(e_{16})$	3; 2; -5	15	1.5ns
8	$\wr h(e_6)$	$\wr h(e_{16})$	2; 3; -5	16	1.2ns
9	$\wr h(e_{11})$	$\wr h(e_{16})$	2; 4; -5	15	1.3ns
10	$h(e_3)\wr$	$h(e_9)\wr$	3; 3; -5	18	1.5ns
11	$h(e_3)\wr$	$h(e_5) \overset{0}{\longrightarrow} h(e_{16})$	2; 4; -5	20	1.6ns

(d) possible signal insertions

$[\text{ready}] = \text{Laf}$
$[\text{Lr}] = \text{start} \cdot \text{Ad} \cdot \text{Ar} + \text{Laf} \cdot (\text{Ar} + \text{start})$
$[\text{Ar}] = \overline{\text{Lam}} \cdot \overline{\text{Laf}} \cdot (\text{Ar} + \overline{\text{Ad}})$
equations for solution 1

$[\text{ready}] = \text{start} \cdot \text{ready} + \text{Laf}$
$[\text{Lr}] = \text{start} \cdot \text{Ad} \cdot \overline{\text{ready}} \cdot \text{Ar} + \text{Laf} \cdot (\text{Ar} + \overline{\text{ready}})$
$[\text{Ar}] = \overline{\text{Lam}} \cdot \overline{\text{Laf}} \cdot (\text{Ar} + \overline{\text{Ad}})$
equations for solution 2

$[\text{ready}] = \text{Laf} + \text{csc}$
$[\text{Lr}] = \text{start} \cdot \text{Ad} \cdot \text{Ar} \cdot \overline{\text{csc}} + \text{Laf} \cdot (\overline{\text{csc}} + \text{Ar})$
$[\text{Ar}] = \overline{\text{Lam}} \cdot \overline{\text{Laf}} \cdot (\text{Ar} + \overline{\text{Ad}})$
$[\text{csc}] = \text{start} \cdot \text{csc} + \text{Laf}$
equations for solution 6

$[\text{ready}] = \text{csc}$
$[\text{Lr}] = \overline{\text{Ar}} \cdot (\text{start} \cdot \overline{\text{csc}} \cdot \text{Ad} + \text{Laf})$
$[\text{Ar}] = \overline{\text{Lam}} \cdot \overline{\text{Laf}} \cdot (\text{Ar} + \overline{\text{Ad}}) + \text{Laf} \cdot \overline{\text{csc}}$
$[\text{csc}] = \text{start} \cdot \text{csc} + \text{Laf}$
equations for solution 7

$[\text{ready}] = \text{Laf} + \text{csc}$
$[\text{Lr}] = \overline{\text{csc}} \cdot (\text{start} \cdot \text{Ar} \cdot \text{Ad} + \text{Laf})$
$[\text{Ar}] = \overline{\text{Lam}} \cdot \overline{\text{Laf}} \cdot (\text{Ar} + \overline{\text{Ad}})$
$[\text{csc}] = \text{start} \cdot \text{csc} + \text{Laf} \cdot \overline{\text{Ar}}$
equations for solution 9

(e) selected equations corresponding to valid transformations

Fig. 11. Top level of the A/D converter.

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 Fig. 11(d).

Solution 6 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 $Laf^- \rightarrow csc^+ \rightarrow Ar^- \rightarrow Lr^-$, 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 ($Laf^+ \rightarrow Ar^- \rightarrow csc^+ \rightarrow Lr^-$).

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 tradeoff 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).

B. Toggle

Toggle is one of the key elements of self-timed micropipeline controllers [19]. The STG of a toggle element is shown in Fig. 12(a). It responds to each even (respectively, odd) transition on input x with a transition on output y_1 (respectively, y_2). The STG satisfies the USC and CSC properties, but violates the normalcy property and therefore it cannot be implemented as a logic circuit build from monotonic gates. The normalcy violation is caused by contradictory hypotheses for y_1 and y_2 as shown in Fig. 12(b). The signals y_1 and y_2 are neither p-normal nor n-normal, e.g., y_1^+ is triggered by x^- (this suggest that y_1 is not p-normal) and y_1^- is triggered by $x-/1$ (this suggest that y_1 is not n-normal).

These violations can be resolved by introducing a new internal signal n_0, e.g., consistent hypotheses that y_1 and y_2 are n-normal can be made after inserting n_0^+ before y_1^- and n_0^- before y_2^+. (Note that consistent trigger-based hypotheses about n-normalcy of the newly inserted signal n_0 can also be made.) However, the verification of the hypotheses produces n-normalcy cores for both y_1 and y_2 shown separately in Fig. 12(c). These cores can be eliminated by the insertion of additional signals as illustrated in this figure.

The resulting circuit is shown in Fig. 12(e); note that none of its gates has input inverters. However, this solution

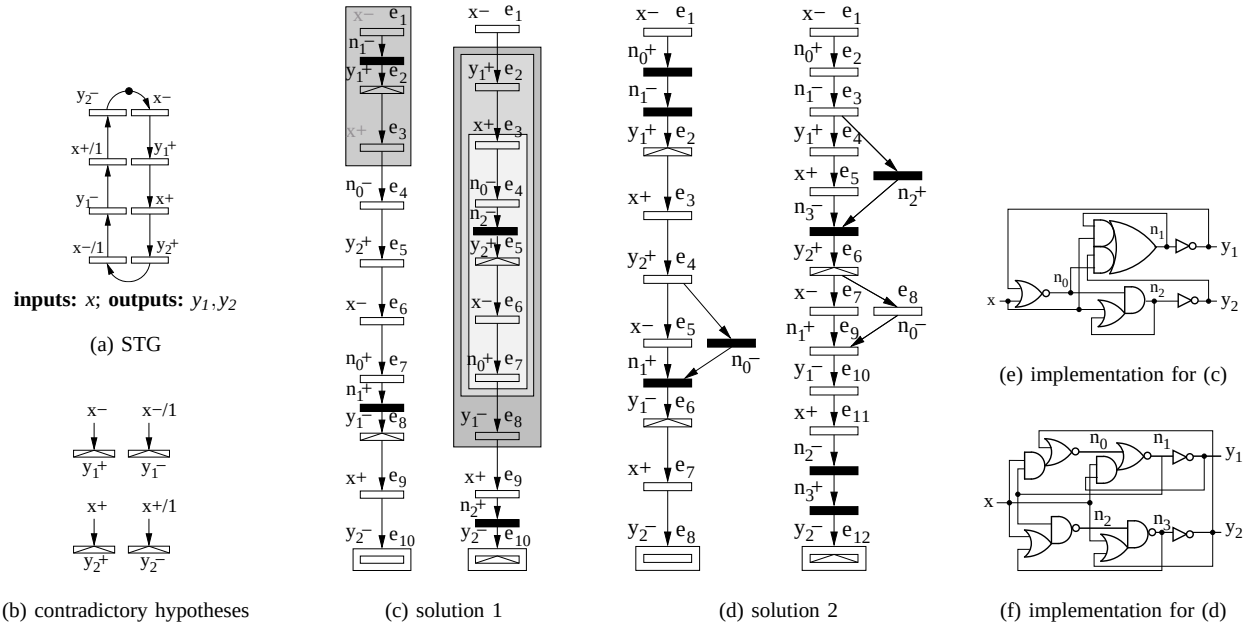


Fig. 12. Toggle element.

is asymmetric and contains large gates. A symmetric circuit with simpler gates can be obtained by using different signals to resolve the contradictory hypotheses about the normalcy types of y_1 and y_2, as illustrated in Fig. 12(d). The consistent hypotheses that y_1 is n-normal can be made after inserting n_1^- before y_1^+ and n_1^+ before y_1^- as shown in the first part of Fig. 12(d). However, the hypotheses about the normalcy type of n_1 are contradictory. To make consistent hypotheses that n_1 is n-normal, another signal, n_0, is inserted, as shown in this figure. The contradictory triggers for y_2 are resolved in a similar way, as shown in the second part of Fig. 12(d), and the resulting circuit is shown in Fig. 12(f).

The equations for the former solution contain 11 literals compared with 14 literals for the latter solution. This is due to the fact that the former uses three internal signal to resolve the normalcy violation and the latter four internal signals. However, the performance in terms of latency is better for the symmetric solution. The wort-case input-to-output delay of the asymmetric solution is 0.65ns ($x^- \to n_0^+ \to n_1^+ \to y_1^-$) compared with the worst-case delay of 0.52ns of the symmetric solution ($x^- \to n_1^- \to y_1^+$).

C. D-element

The STG of a handshake decoupling D-element [20], [21] is shown in Fig. 13(a). It controls two handshakes, where one handshake initiates the other. The first handshake waits for the other to complete. Then, the first handshake completes, and the cycle is repeated.

The initial STG has a CSC conflict, resulting in a CSC core shown in Fig. 13(b,d). The conflict occurs because the encodings before and after the execution of the second handshake ($r_2^+ \to a_2^+ \to r_2^- \to a_2^-$) are equal. Two types of insertion are considered. In the first case, a single signal insertion is applied, and in the second case, a flip-flop insertion is used. In both cases auxiliary signals are inserted either sequentially or concurrently. The concurrent insertion is performed in order to achieve lower latency, because it does not delay any output signals.

The sequential insertions of csc^+ before r_2^- and csc^- before a_1^- (where the former transformation is performed inside the core and the latter is outside the core) are illustrated in the first part of Fig. 13(b). Note that input signals must not be delayed by the internal signal and thus this is the only possible valid fully sequential insertion of *csc* (up to the signs of inserted transitions). These transformation introduce inconsistent normalcy type hypotheses for a_1 and *csc*: the signal a_1 has negative triggers for both a_1^+ and a_1^-, and the signal *csc* has positive triggers for both csc^+ and csc^-, which makes a_1 and *csc* neither p-normal nor n-normal. Similarly, reversing the polarity of the instances of *csc* introduces inconsistent normalcy type hypotheses for r_2 and *csc*.

The concurrent insertion $r_2^+ \longrightarrow\!\!\!\!\!\!\circ\; r_2^-$ adds csc^+ inside the core concurrently to a_2^+, the concurrent insertion $a_1^+ \longrightarrow\!\!\!\!\!\!\circ\; a_1^-$ adds csc^- outside the core, concurrently to r_1^-, as shown in Fig. 13(b). Note that there is no other locations to insert *csc* concurrently. In this transformation the normalcy type hypotheses for a_1 and *csc* are also inconsistent. The reversing of the polarities of the inserted signal transitions also introduces inconsistent normalcy type hypotheses for r_2 and *csc*.

In both sequential and concurrent transformations, the normalcy is violated due to the inconsistent polarity of triggers. This is reflected in the existence of input inverters in the implementation of these solutions in Fig. 13(c). The sequential solution has somewhat simpler logic (7 literals) compared with the concurrent solution (13 literals). The sequential solution has two transition delays between two adjacent input transitions, whereas the concurrent solution has only one transition delay. The maximum input-to-output delay in the sequential solution is 0.54ns ($r_1^+ \to csc^- \to a_1^-$), and in the concurrent

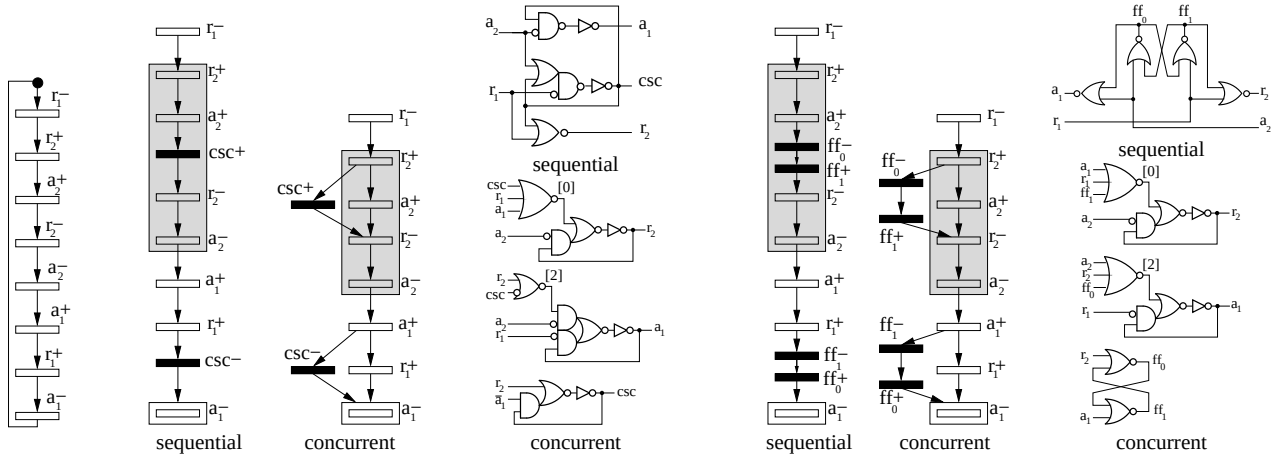


Fig. 13. D-element.

solution it is 0.65ns ($a_2^- \rightarrow a_1^+$).

The sequential and concurrent flip-flop insertions are shown in Fig. 13(d), and the corresponding implementations are shown in Fig. 13(e). Note that the sequential solution produces an n-normal STG, which is reflected in the absence of input inverters in the implementation. The concurrent flip-flop insertion yields an STG for which the normalcy type hypotheses are consistent. However, the verification of hypotheses shows that the STG is not normal, e.g., the n-normalcy core $\{a_2^+, r_2^-, a_2^-\}$ shows that r_2 is not n-normal, as the hypotheses based on its triggers suggest. (Normalcy is also violated for a_1.) Thus additional inverters are needed in the implementation, as shown in Fig. 13(e). The equations for the sequential and concurrent solutions have 8 and 14 literals, respectively. The sequential solution has three transition delays between two adjacent input transitions, whereas the concurrent solution has only one transition delay but its logic is more complex. The maximum input-to-output delay in the former is 0.43ns ($r_1^+ \rightarrow ff_1^- \rightarrow ff_0^+ \rightarrow a_1^-$), and in the latter it is 0.56ns ($r_1^- \rightarrow r_2^+$).

This example shows that the concurrent insertion, although not delaying output signals, may produce slower and larger circuits compared with the sequential insertion (which delays output signals). This is caused by complex logic, derived from an increased combination of reachable signal values leaving less room for Boolean minimisation. The sequential flip-flop insertion offers a good solution in terms of size and latency. Additionally, it consists of monotonic, simple and negative gates.

V. Conclusion and future work

A framework for the interactive resolution of a wide class of encoding conflicts in STG unfoldings has been presented. It explores a larger design space and allows the designer to exploit the area/delay tradeoff, which is crucial in synthesis of controllers. Encoding conflicts are represented by means of cores, which are sets of transitions 'causing' them. The advantage of using cores is that only those parts of STGs which cause encoding conflicts, rather than the complete list of encoding conflicts, are considered. Since the number of cores is usually much smaller than the number of encoding conflicts, this approach reduces the amount of information to be analysed.

The future work will be focused on the following issues:

- improving the cost function;
- performing the transformations directly on the unfolding prefix rather than the STG whenever possible, in order to reduce the number of runs of the unfolding algorithm.
- extending the framework by including timing assumption for the resolution of encoding conflicts, which can be derived from the transformations based on concurrency reduction.
- extending the described technique to logic decomposition and technology mapping.

References

[1] T.-A. Chu, "Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications," Ph.D. dissertation, MIT Laboratory for Computer Science, 1987.

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

[3] P. Riocreux, "Private communication," UK Asynchronous Forum, 2002.

[4] N. Starodoubtsev, S. Bystrov, M.Goncharov, I. Klotchkov, and A. Smirnov, "Towards Synthesis of Monotonic Asynchronous Circuits from Signal Transition Graphs," in *Proc. ACSD'01*. IEEE Comp. Soc. Press, 2001, pp. 179–188.

[5] A. Madalinski, A. Bystrov, V. Khomenko, and A. Yakovlev, "Visualization and Resolution of Coding Conflicts in Asynchronous Circuit Design," in *Proc. DATE'03*. IEEE Comp. Soc. Press, 2003, pp. 926–931, Full version: IEE Proceedings, Computers and Digital Techniques, vol. 150, no. 5, pp. 285–293, 2003.

[6] V. Khomenko, "Model Checking Based on Petri Net Unfolding Prefixes," Ph.D. dissertation, University of Newcastle upon Tyne, 2003.

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

[8] ——, "Detecting State Coding Conflicts in STGs Unfoldings using SAT," in *Proc. ACSD'03*. IEEE Comp. Soc. Press, 2003, pp. 51–60.

[9] ——, "Logic Synthesis for Asynchronous Circuits Based on Petri Net Unfoldings and Incremental SAT," in *Proc. ACSD'04*. IEEE Comp. Soc. Press, 2004, pp. 16–25.

[10] A. Madalinski, "CONFRES: Interactive Coding Conflict Resolver Based on Core Visulation," in *Proc. ACSD'03*. IEEE Comp. Soc. Press, 2003.
[11] A. Semenov, "Verification and Synthesis of Asynchronous Control Circuits Using Petri Net Unfolding," Ph.D. dissertation, University of Necastle upon Tyne, 1997.
[12] H. Saito, A. Kondratyev, J. Cortadella, L. Lavagno, and A. Yakovlev, "What is the cost of delay insensitivity?" in *Proc. HWPN'99*, 1999, pp. 169–189.
[13] V. Khomenko, A. Madalinski, and A. Yakovlev, "Resolution of Encoding Conflicts by Signal Insertion and Concurrency Reduction Based on STG Unfoldings," School of Computing Science, University of Newcastle upon Tyne, Tech. Rep. CS-TR-858, 2004.
[14] A. Madalinski, "Interactive Synthesis of Asynchronous Systems based on Partial Order Semantics," Ph.D. dissertation, University of Newcastle upon Tyne, 2005.
[15] A. Bardsley, "Implementing Balsa Handshake Circuits," Ph.D. dissertation, School of Computer Science, University of Manchester, 2000.
[16] J. Carmona, J.-M. Colom, J. Cortadella, and F. Garcia-Vallés, "Synthesis of Asynchronous Controllers Using Integer Linear Programming," *IEEE Transactions on CAD*, 2006, to appear.
[17] D. Sokolov, A. Bystrov, and A. Yakovlev, "STG Optimisation in the Direct Mapping of Asynchronous Circuits," in *Proc. DATE'03*. IEEE Comp. Soc. Press, 2003.
[18] D. J. Kinniment, B. Gao, A. Yakovlev, and F. Xia, "Towards Asynchronous A-D Conversion," in *Proc. ASYNC'98*. IEEE Comp. Soc. Press, 1998, pp. 206–215.
[19] I. E. Sutherland, "Micropipelines," *Communications of the ACM*, vol. 32, no. 6, pp. 720–738, 1989.
[20] A. Bystrov, D. Shang, F. Xia, and A. Yakovlev, "Self-Timed and Speed Independent Latch Circuits," in *Proc. UK Asynchronous Forum*. Department of Computing Science, University of Manchester, 1999.
[21] V. I. Varshavsky, Ed., *Self-Timed Control of Concurrent Processes: The Design of Aperiodic Logical Circuits in Computers and Discrete Systems*. Kluwer Academic Publishers, 1990.