School of Electrical, Electronic & Computer Engineering

UNIVERSITY OF
NEWCASTLE UPON TYNE

Interactive Synthesis of Asynchronous Systems based on Partial Order Semantics

Agnes Madalinski

Technical Report Series

NCL-EECE-MSD-TR-2006-112

February 2006

Contact:

a.a.madalinski@ncl.ac.uk

EPSRC supports this work via GR/M94366 (MOVIE) and GR/R16754 (BESST).

NCL-EECE-MSD-TR-2006-112

School of Electrical, Electronic & Computer Engineering,

Merz Court,

University of Newcastle upon Tyne,

Newcastle upon Tyne, NE1 7RU, UK

http://async.org.uk/

University of Newcastle upon Tyne

School of Electrical, Electronic and Computer Engineering

Interactive Synthesis of Asynchronous Systems based on Partial Order Semantics

by

A. Madalinski

PhD Thesis

May 2005

To my Mother

Contents

List of Figures **v**

List of Tables **viii**

List of Algorithms **ix**

Acknowledgements **x**

Abstract **xi**

1 Introduction **1**

1.1 Asynchronous vs. synchronous 1

1.2 Control signalling 5

1.3 Data path 7

1.4 Delay models 10

1.5 Synthesis of control logic 16

1.6 Motivation for this work 18

1.7 Main contribution of this work 20

1.8 Organisation of thesis 21

2 Formal models **23**

2.1 Petri Nets 23

2.1.1 Enabling and firing 26

2.1.2 Petri net properties 27

2.1.3 Subclasses of Petri nets 28

2.2 Signal Transition Graphs and State Graphs . . . 29
2.2.1 Signal Transition Graphs . . . 30
2.2.2 State Graphs . . . 32
2.2.3 Signal Transition Graph properties . . . 32
2.3 Branching processes . . . 35
2.3.1 Configurations and cuts . . . 39
2.3.2 Complete prefixes of PN unfoldings . . . 41
2.3.3 LPN branching processes . . . 43

3 Logic synthesis 46
3.1 Related work . . . 46
3.2 Implementation as logic circuit . . . 49
3.3 State-based logic synthesis . . . 50
3.4 Synthesis example . . . 51
3.5 State encoding problem . . . 54

4 Detection of encoding conflicts 60
4.1 Approximate state covering approach . . . 60
4.1.1 Necessary condition . . . 62
4.1.2 Refinement by partial state construction . . . 65
4.2 Implementation . . . 66
4.2.1 Collision stable conditions . . . 67
4.2.2 ON-set of a collision relation . . . 69
4.2.3 Overall procedure . . . 73
4.2.4 Reducing the number of fake conflicts . . . 76
4.3 Experimental results . . . 79
4.4 Conclusions . . . 82

5 Visualisation and resolution of encoding conflicts 84
5.1 Compact representation of conflicts . . . 85
5.2 Net transformation for resolution of conflicts . . . 92
5.2.1 Validity . . . 93

5.2.2 Concurrency reduction . . . 95
5.2.3 Signal insertion . . . 97
5.2.3.1 Transition splitting . . . 97
5.2.3.2 Concurrent insertion . . . 100
5.3 Resolution concept . . . 101
5.3.1 $\mathcal{X}$ cores elimination: a general approach . . . 102
5.3.2 $\mathcal{X}$ core elimination by signal insertion . . . 104
5.3.2.1 Single signal insertion . . . 104
5.3.2.2 Dual signal insertion (flip flop insertion) . . . 108
5.3.3 $\mathcal{X}$ core elimination by concurrency reduction . . . 111
5.3.4 Resolution constraints . . . 114
5.4 Resolution process . . . 117
5.4.1 Manual resolution . . . 118
5.4.1.1 Transformation condition . . . 119
5.4.1.2 Resolution examples . . . 120
5.4.2 Automated resolution . . . 125
5.4.3 Cost function . . . 126
5.5 Tool ConfRes: CSC conflict resolver . . . 131
5.5.1 Applied visualisation method . . . 131
5.5.2 Description . . . 133
5.5.3 Implementation . . . 136
5.6 Conclusion . . . 140

6 Interactive synthesis **141**

6.1 VME-bus controller . . . 141
6.2 Weakly synchronised pipelines . . . 145
6.3 Phase comparator . . . 147
6.4 AD converter . . . 149
6.4.1 Top level controller . . . 150
6.4.2 Scheduler . . . 154
6.5 D-element . . . 157

6.6 Handshake decoupling element 160
6.7 GCD 165
6.8 Conclusion 172

7 Conclusion **175**

7.1 Summary 175
7.2 Areas of further research 179

Bibliography **181**

List of Figures

1.1 Signalling protocols 6
1.2 Single-rail protocol 8
1.3 Four-phase dual-rail 9
1.4 Two-phase dual-rail 10
1.5 Huffman circuit model 12
1.6 Illustration of delay models 13
1.7 Micropipeline structure 15

2.1 An example: a PN and its RG, an STG, its SG and its unfolding prefix 25
2.2 Example: An illustration of a firing rule 26
2.3 Examples of subclasses of PN systems 29
2.4 A PN and one of finite and complete prefixes of its unfoldings 38

3.1 Overview of approaches for STG-based logic synthesis 47
3.2 Design flow for asynchronous control circuits 50
3.3 VME-bus controller 52
3.4 CSC refinement 52
3.5 Read cycle VME-bus controller implementation 53
3.6 Approach overview for the resolution of the CSC problem 55

4.1 An example for the approximation technique 64
4.2 Traversing a node 69
4.3 Deriving the ON-set 72
4.4 Reduction of a marked region by procedure *reduceSlice* 73

4.5 Detection of state coding conflicts 77
4.6 STG unfoldings with different initial markings 78

5.1 Visualisation of CSC conflicts 86
5.2 Visualisation examples of $\mathcal{X}$ cores 87
5.3 Visualisation of normalcy violation 91
5.4 Concurrency reduction $U \overset{n}{\dashrightarrow} t$ 96
5.5 Transition splitting 97
5.6 Concurrent insertion $v \overset{n}{\longrightarrow\!\!\!\!\!\rightarrow} w$ 100
5.7 Strategies for eliminating $\mathcal{X}$ cores 103
5.8 $\mathcal{X}$ core elimination by signal insertion 105
5.9 Example: elimination of CSC cores in sequence 105
5.10 Example: elimination of CSC cores in concurrent parts 106
5.11 Example: elimination of CSC cores in choice branches 107
5.12 Strategies for $\mathcal{X}$ core elimination 108
5.13 Strategy for flip flop insertion 109
5.14 Example: flip flop insertion 110
5.15 $\mathcal{X}$ core elimination by concurrency reduction 112
5.16 Elimination of type I CSC cores 113
5.17 Elimination of type II CSC cores 114
5.18 Visualisation of encoding conflicts 115
5.19 Intersection of CSC cores 116
5.20 N-normalcy violation for signal b 117
5.21 The resolution process of encoding conflicts 118
5.22 CSC conflict resolution 121
5.23 Normalcy conflict resolution 123
5.24 Logic decomposition $(\overline{d} + \overline{b})$: signal insertion $n_1^+ \rightarrow\!\!\!\!\wr a^-$ and $n_1^- \rightarrow\!\!\!\!\wr n_0^+$ . . 124
5.25 Overview: automated resolution process 126
5.26 Visualisation of encoding conflict: an example 132
5.27 ConfRes dependencies 133
5.28 Interaction with ConfRes: an example 135

6.1 VME-bus controller: read cycle . . . 142
6.2 Selected equations . . . 144
6.3 Weakly synchronised pipelines . . . 146
6.4 Phase comparator . . . 148
6.5 Block diagram of the AD converter . . . 149
6.6 STG transformation of the top level controller . . . 151
6.7 Valid transformations for top level controller . . . 153
6.8 STG transformation of the scheduler . . . 154
6.9 Logic equations . . . 155
6.10 Resolution attempts for the scheduler . . . 155
6.11 D-element . . . 158
6.12 Handshake decoupling element . . . 161
6.13 Handshake decoupling element: final implementation . . . 164
6.14 Implementation of the decomposed flip flop solution . . . 165
6.15 GCD . . . 167
6.16 Equations for GCD controller obtained by PETRIFY . . . 168
6.17 Resolution process of GCD controller . . . 169
6.18 Equations for GCD controller obtained by CONFRES . . . 171
6.19 Complex gates implementation of GCD controller (sequential solution) . . . 172

7.1 Applications of interactive refinement of CSC conflicts . . . 177

List of Tables

4.1 Necessary condition 80
4.2 Size of the traversed state space 81
4.3 Number of fake conflicts 81
4.4 Reduction of collisions 82

6.1 Possible single signal insertions 144
6.2 Comparison: automatic and manual refinement 172

List of Algorithms

1 Identifying collision stable conditions 68

2 Procedure to traverse an event 70

3 Procedure to traverse a condition 71

4 Constructing the ON-set of a collision relation 71

5 Setting the "first" min-cuts for c 74

6 Reducing slice 74

7 Detecting state coding conflicts by STG unfolding 75

8 Resolution of encoding conflicts 137

9 Computation of possible transformations 137

10 Computation of valid transformations (phase one) 139

Acknowledgements

This work would not have been possible without intensive collaboration with many people. I would like to express my gratitude to my supervisor, Alex Yakovlev, for introducing me to the ideas of asynchronous circuit design and for supporting me during my research.

Most of the ideas described in this thesis are a result of numerous discussions with Alex Bystrov, Victor Khomenko and, of course, my supervisor. I would also like to thank my fellow PhD students, especially Imam Kistijantoro and Danil Sokolov, who been very helpful whenever I needed advice and information about all sorts of practical questions.

In addition, I would like to thank Margie Craig for reading through the whole PhD thesis and helping me with my English.

Finally, I would like to acknowledge that this work was supported by the EPSRC projects MOVIE (grant GR/M94366) and BESST (grant GR/R16754) at the University of Newcastle upon Tyne.

Abstract

An interactive synthesis of asynchronous circuits from Signal Transition Graphs (STGs) based on partial order semantics is presented. In particular, the fundamental problem of encoding conflicts in the synthesis is tackled, using partial orders in the form of STG unfolding prefixes. They offer a compact representation of the reachable state space and have the added advantage of simple structures.

Synthesis of asynchronous circuits from STGs involves resolving state encoding conflicts by refining the STG specification. The refinement process is generally done automatically using heuristics. It often produces sub-optimal solutions, or sometimes fails to solve the problem, then requiring manual intervention by the designer. A framework for an interactive refinement process is presented, which aims to help the designer to understand the encoding problems. It is based on the visualisation of conflict cores, i.e. sets of transitions causing encoding conflicts, which are represented at the level of finite and complete prefixes of STG unfoldings. The refinement includes a number of different transformations giving the designer a larger design space as well as applying to different levels of interactivity. This framework is intended to work as an aid to the established state-based synthesis. It also contributes to an alternative synthesis based on unfolding prefixes rather than state graphs.

The proposed framework has been applied to a number of design examples to demonstrate its effectiveness. They show that the combination of the transparent synthesis together with the experience of the designer makes it possible to achieve tailor-made solutions.

Chapter 1

Introduction

Asynchronous circuits design has been an active research area since the early days of digital circuit design, but only during the last decade has there been a revival in the research on asynchronous circuits [1, 2, 83]. Up until now, asynchronous circuits have only been applied commercially as small sub-circuits, often as peripherals to controllers. Examples [24] include counters, timers, wake-up circuits, arbiters, interrupt controllers, FIFOs, bus controllers, and interfaces. The need for such asynchronous circuits stems largely from intrinsically asynchronous specifications. Emphasis is now shifting from small asynchronous sub-circuits to asynchronous VLSI circuits and systems. Asynchronous VLSI is now progressing from a fashionable academic research topic to a viable solution to a number of digital VLSI design challenges.

This chapter briefly outlines the motivation and basic concept of asynchronous circuit design. A number of works, e.g. [9, 21, 30, 75, 98], provide a more extensive introduction to, and comparison of, the most common approaches. This chapter also describes the main contribution and the organisation of this thesis.

1.1 Asynchronous vs. synchronous

The majority of the digital circuits designed and manufactured today are synchronous. In essence, they are based on two assumptions which greatly simplify their design. Firstly, all signals are assumed binary where simple Boolean logic can be used to de-

scribe and manipulate logic constructs. Secondly, all components share a common and discrete notation of time, as defined by a clock signal distributed throughout the circuit. However, asynchronous circuits are fundamentally different. They also assume binary signals, but there is no common and discrete time. Instead the circuits use handshaking between their components in order to perform the necessary synchronisation, communication and sequencing operations. This difference gives asynchronous circuits inherent properties as listed below. A further introduction to the advantages mentioned below can be found in [5].

- Power efficiency

  The power consumption of high performance synchronous circuits is related to the clock, which signal propagates to every operational block of the circuit even though many parts of the circuit are functionally idle. Thus energy is wasted on driving the clocked inputs of these gates which do not perform any useful actions. In contrast, an asynchronous circuit only consumes energy when and where it is active. Any sub-circuit is quiescent until activated. After completion of its task, it returns to a quiescent state until a next activation. However, it is not obvious to what extent this advantage is fundamentally asynchronous. Synchronous techniques such as clock gating may achieve similar benefits, but have their limitations.

- Performance

  In a synchronous circuit all the computation must be completed before the end of the current cycle, otherwise an unstable and probably incorrect signal value may be produced. To guarantee the correct operation of synchronous circuits it is mandatory to find out the propagation times through all the paths in the combinational gates, assuming any possible input pattern. Then the length of the clock cycle must be longer than the worst propagation delay. This fixed cycle time restricts the performance of the system.

  In an asynchronous circuit the next computation step can start immediately after the previous step has been completed. There is no need to wait for a transition of

the clock signal. This leads, potentially, to a fundamental performance advantage for asynchronous circuits, an advantage that increases with the variability in delays associated with these computation steps.

- Clock skew

  Reliable clock distribution is a big problem in complex VLSI chips because of the clock skew effect. This is caused by variations in wiring delays to different parts of the chip. It is assumed that the clock signal reaches the different stages of the chip simultaneously. However, as chips get more complex and logic gates reduced in size, the ratio between gate delays and wire delays changes so that latter begin to significantly affect the operation of the circuit. In addition, clock wiring can take more than half of all the wiring in a chip. By choosing an asynchronous implementation the designer escapes the clock skew problem and the associated routing problem. Although asynchronous circuits can also be subject to the greater effect of wire delays; those problems are solved at a much more local level.

- Electromagnetic compatibility (EMC)

  The clock signal is a major cause of electromagnetic radiation emissions, which are widely regarded as a health hazard or source of interference, and are becoming subject to strict legislation in many countries. EMC problems are caused by radiation from the metal tracks that connect the clocked chip to the power supply and target devices, and from the fact that on-chip switching activity tends to be concentrated towards the end of the clock cycle. These strong emissions, being harmonics of the clock frequency, may severely affect radio equipment. Due to the absence of a clock, asynchronous circuits have a much better energy spectrum than synchronous circuits. The spectrum is smoother and the peak values are lower due to irregular computation and communication patterns.

- Modularity

  In a synchronous circuit all its computational modules are driven by one basic clock (or a few clocks rationally related to each other), and hence must work at a fixed speed rate. If any functional unit is redesigned and substituted by a faster

unit no direct increase in performance will be achieved unless the clock frequency is increased in order to match the shorter delay of the newly introduced unit. The increased clock speed, however, may require the complete redesign of the whole system.

Asynchronous circuits, however, are typically designed on the basis of explicit *self-timed* protocols. Any functional unit can be correctly substituted by another unit implementing the same functionality and following the same protocol but with a different performance. Hence asynchronous circuits achieve better *plug and play* capabilities.

– Metastability

All synchronous chips interact with the outside world, e.g. via interrupt signals. This interaction is inherently asynchronous. A synchronisation failure may occur when an unstable synchronous signal is sampled by a clock pulse into a memory latch. Due to the dynamic properties of an electronic device that contains internal feedback, the latch may, with nonzero probability, hang in a metastable state (somewhere in between logical *0* and *1*) for a theoretically indefinite period of time. Although in practice this time is always bounded, it is much longer than the clock period. As a result, the metastable state may cause an unpredictable interpretation in the adjacent logic when the next clock pulse arrives. Most asynchronous circuits wait until metastability is resolved. Even though in some real time application this may still cause failure, the probability is very much lower than in synchronous systems, which must trade off reliability against speed.

On the other hand there are also some drawbacks. Primary, asynchronous circuits are more difficult to design in an ad hoc fashion than synchronous circuits. In a synchronous system, a designer can simply define the combinational logic necessary to compute the given functions, and surround it with latches. By setting the clock rate to a long enough period, worries about hazards (undesired signal transitions) and the dynamic state of the circuit are removed. In contrast, designers of asynchronous systems must pay attention to the dynamic state of the circuit. To avoid incorrect results hazards must be

removed from the circuit, or not introduced in the first place. The predominance of CAD tools orientated towards synchronous systems makes it difficult to design complex asynchronous systems. However, most circuit simulation techniques are independent of synchrony, and existing tools can be adapted for asynchronous use. Also, there are several methodologies and CAD tools developed specifically for asynchronous design. Some of them are discussed later. Another obstacle is that asynchronous design techniques are not typically taught in universities. If a circuit design company decides to use an asynchronous logic it has to train its engineering staff in the basics.

Finally, the asynchronous control logic that implements the handshaking normally represents an overhead in terms of silicon area, circuit speed, and power consumption. It is therefore pertinent to ask whether or not the investment pays off, i.e. whether the use of asynchronous techniques result in a substantial improvement in one or more of the above areas. In spite of the drawbacks, there exist several commercial asynchronous designs which benefit from the advantages listed above. Such designs include the AMULET microprocessors developed at the University of Manchester [28], a contactless smart card chip developed at Philips [35], a Viterbi decoder developed at the University of Manchester [88], a data driven multimedia processor developed at Sharp [101], and a flexible 8-bit asynchronous microprocessor developed at Epson [34].

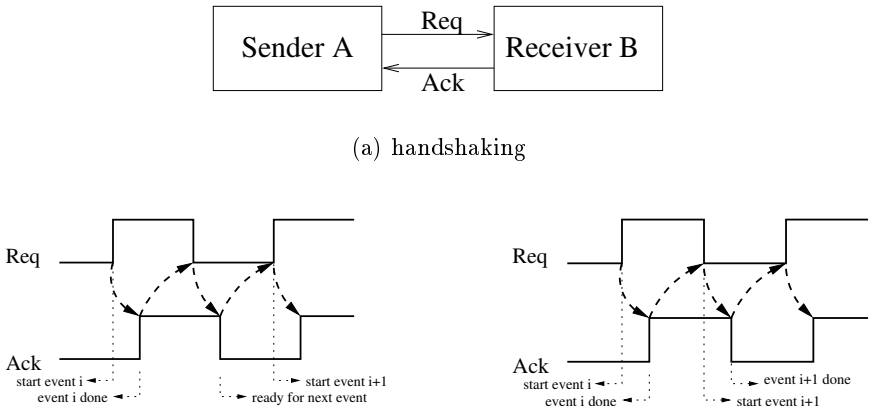
1.2 Control signalling

Asynchronous circuits do not use clocks and therefore synchronisation must be done in a different way. One of the main paradigms in asynchronous circuits is *distributed control*. This means that each unit synchronises only with those units for which the synchronisation is relevant and at the time when synchronisation takes place, regardless of the activities carried out by other units in the same circuit. For this reason, asynchronous functional units incorporate explicit signals for synchronisation that execute some *handshake* protocol with their neighbours. For example, let there be two units, a sender A and a receiver B, as shown in Figure 1.1(a). A request is sent from A to B to indicate that A is requesting some action from B. When B has either done the action or

has stored the request, it acknowledges the request by sending the acknowledge signal, which is sent from B to A. To operate in this manner, data path units commonly have two signals, *request* and *acknowledge* used for synchronisation. The request signal is an input signal used by the environment to indicate that input data are ready and that an operation is requested to the unit. The output signal, the acknowledge signal, is used by the unit to indicate that the requested operation has been completed and that the result can be read by the environment.

The purpose of an asynchronous control circuit is to interface a portion of data path to its sources and sinks of data. The controller manages all the synchronisation requirements, making sure that the data path receives its input data, performs the appropriate computations when they are valid and stable, and that the results are transferred to the receiving blocks whenever they are ready.

The most commonly used handshake protocols are the four-phase and two-phase protocol.



(a) handshaking

(b) four-phase

(c) two-phase

Figure 1.1: Signalling protocols

Four-phase protocol The four-phase protocol also called return-to-zero (RZ) is shown in Figure 1.1(b). The waveforms appear periodic for convenience but they do not need to be so in practice. The curved arrows indicate the required before/after sequence of events. There are no implicit assumptions about the delay between successive events. Note that in this protocol there are typically four transitions (two on the request and

two on the acknowledgement) required to complete a particular event transition.

Two-phase protocols The two-phase protocol, also called non-return-to-zero (NRZ), is shown in Figure 1.1(c). The waveforms are the same as for four-phase signalling with the exception that every transition on the request wire, both falling and rising, indicates a new request. The same is true for transition on the acknowledgement wire.

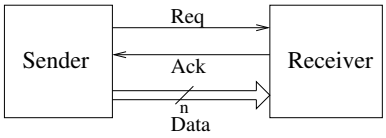
Proponents of the four-phase signalling scheme argue that typically four-phase circuits are smaller than they are for two-phase signalling, and that the time required for the falling transition on the request and on the acknowledge lines do not usually cause performance degradation. This is because falling transitions happened in parallel with other circuit operations. Two-phase proponents argue that two-phase signalling is better from both a power and a performance standpoint, since every transition represents a meaningful event and no transitions or power are consumed in the return-to-zero, because there is no resetting of the handshake link. Whilst this is true, in principle, it is also the case that most two-phase interface implementations require more logic that four-phase equivalents.

Other interface protocols, based on similar sequencing rules, exist for three or more module interfaces. A particular common design requirement is to conjoin two or more requests to provide a single outgoing request, or conversely to provide a conjunction of acknowledge signals. A commonly used asynchronous element is the *C-element* [73], which can be viewed as a protocol preserving conjunctive gate. The C-element effectively merges two requests into a single request and permits three subsystems to communicate in a protocol preserving two- or four-phase manner.

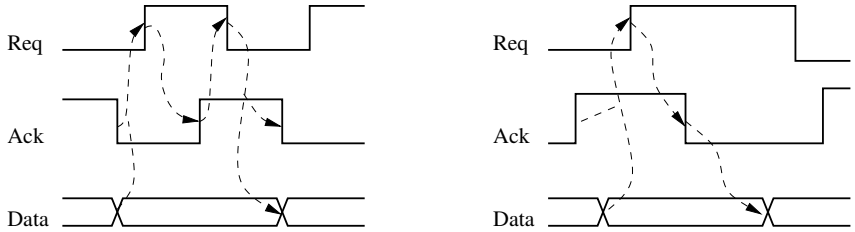
1.3 Data path

The previous section only addressed control signals. There are also several approaches to encoding data. The most common are the single- and dual-rail protocols.

Single-rail protocol The *single-rail* protocol, also referred to as *bundled-data* protocol, uses either two- or four-phase signalling to encode data. In this case, for an n-bit data value to be passed from sender to receiver, n+2 wires will be required (n bits of data, one request bit and one acknowledge bit), see Figure 1.2(a). The data signals are bundled with the request and acknowledgement. This type of data encoding contains an implied timing assumption, namely the assumption that the propagation times of control and data are either equal, or that the control propagates slower than the data signals.



(a) channel



(b) four-phase (c) two-phase

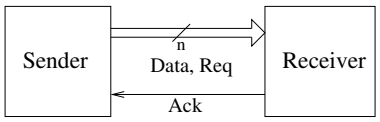
Figure 1.2: Single-rail protocol

The four-phase protocol is illustrated in Figure 1.2(b). The term four-phase refers to the number of communication actions. First, the sender issues data and sets request high, then the receiver absorbs the data and sets acknowledge high. The sender responds by taking request low, at which point data is no longer guaranteed to be valid. Finally, the receiver acknowledges this by taking acknowledge low. At this point the sender may initiate the next communication cycle.

Dual-rail protocol A common alternative to the single-rail approach is *dual-rail* encoding. The request signal is encoded into the data signal using two wires per bit of information. For an n-bit data value the sender and receiver must contain $3n$ wires:

two wires for each bit of data and the associated request, plus another bit for the acknowledge. An improvement on this protocol is possible when n-bits of data are considered to be associated in every transition, as in the case when the circuit operates on bytes or words. Then it is convenient to combine the acknowledges into a single wire. The wiring complexity is reduced to $2n + 1$ wires: $2n$ for the data and an additional acknowledge signal.

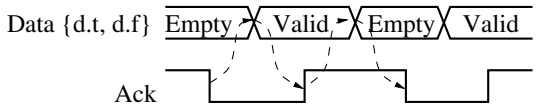
The four-phase dual-rail protocol, shown in Figure 1.3(a), is in essence a four-phase protocol using two request wires per bit of information, d; one wire $d.t$ is used for signalling a logic *true*, and another $d.f$ is used for signalling a logic *false*. When observing a *1*-bit channel one will see a sequence of four-phase handshakes where the participating *request* signal in any handshake cycle can be either $d.t$ or $d.f$. This protocol is very robust, because two parties can communicate reliably regardless of delays in the wires connecting the two parties, thus the protocol is delay-insensitive.



(a) channel

	d.t	d.f
Empty	0	0
Valid 0	0	1
Valid 1	1	0
Not used	1	1

(b) encoding

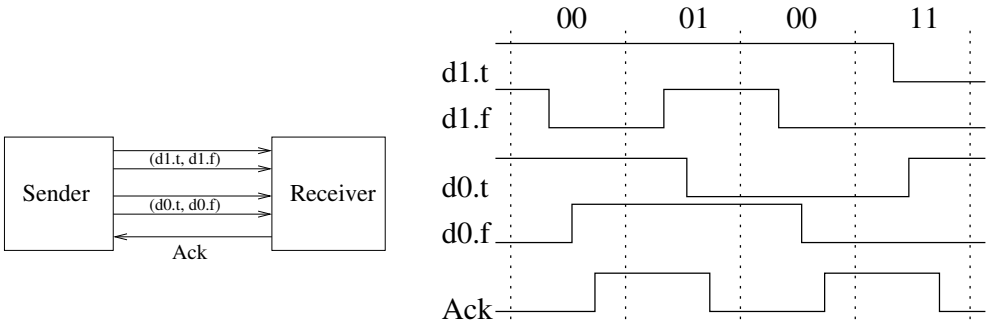


(c) waveforms

Figure 1.3: Four-phase dual-rail

The encoding is presented in Figure 1.3(b). It has four codewords: two valid codewords (representing logic *true* and logic *false*), one idle and one illegal codeword. If the sender issues a valid codeword, then the receiver absorbs the codeword and sets acknowledge high. The sender responds by issuing the empty codeword, which is then acknowledged by the receiver, by taking the acknowledge low. At this point the sender may initiate the next communication cycle. This process is illustrated in Figure 1.3(c). An abstract view is a data stream of valid codewords separated by empty ones.

The two-phase dual-rail protocol also uses two wires per bit, but the information is encoded as events (transitions). On an n-bit channel a new codeword is received when exactly one wire in each of the n wire pairs has made a transition. There is no empty value. A valid message is acknowledged and followed by another message that is acknowledged. Figure 1.4 shows the signal waveforms on a 2-bit channel using two-phase dual-rail protocol.



(a) channel (b) waveforms

Figure 1.4: Two-phase dual-rail

There exist other communication protocols such as 1-of-n encodings used in control logic and higher-radix data encodings. If the focus is on communication rather then computation, m-of-n encodings may be of relevance.

1.4 Delay models

Asynchronous design methodologies are classified by the delay models which they assume on gates, wires and feedback elements in the circuit. There are two fundamental models of delay, the *pure delay* model and the *inertial delay* model. A pure delay model can delay the propagation of a waveform, but does not otherwise alter it. An inertial delay can alter the shape of a waveform by attenuating short glitches. More formally, an inertial delay has a threshold period. Pulses of a duration less then the threshold period are filtered out.

Delays are also characterised by their timing models. In a *fixed delay* model, a delay is assumed to have a fixed value. In a *bounded delay* model, a delay may have any value

in a given time interval. In *unbounded delay* model, a delay may take on any finite value.

An entire circuit's behaviour can be modelled on the basis of its component model. In a *gate-level* model each gate and primitive component in the circuit has a corresponding delay. In a *complex-gate* model, an entire sub-network of gates is modelled by a single delay, that is, the network is assumed to behave as a single operator, with no internal delays. Wires between gates are also modelled by delays. A *circuit model* is thus defined in terms of the delay models for the individual wires and components. Typically, the functionality of a gate is modelled by an instantaneous operator with an attached delay.

Given the circuit model, it is also important to characterise the interaction of the circuit with its environment. The circuit and its environment together form a closed system, called *complete circuit*. If the environment is allowed to respond to a circuit's outputs without any timing constraints, the two interact in *input/output mode*. Otherwise, environmental timing constraints are assumed. The most common example is *fundamental mode* where the environment must wait for a circuit to stabilise before responding to circuit outputs.

This section introduces widely used models, and briefly reviews the existing design methodologies for each model.

Huffman circuits The most obvious model to use for asynchronous circuits is the same model used for synchronous circuits. Specifically, it is assumed that the delay in all circuit elements and wires is known, or at least bounded. Huffman circuits are designed using a traditional asynchronous state machine approach. As illustrated in Figure 1.5, an asynchronous state machine has primary inputs, primary outputs, and feed-back state variables. The state is stored in the feedback loop and thus may need delay elements along the feedback path to prevent state changes from occurring too rapidly.

The design of Huffman circuits begins with a specification given in a flow table [102] which can be derived using an asynchronous state machine. The goal of the synthesis procedure is to divide a correct circuit netlist which has been optimised according to

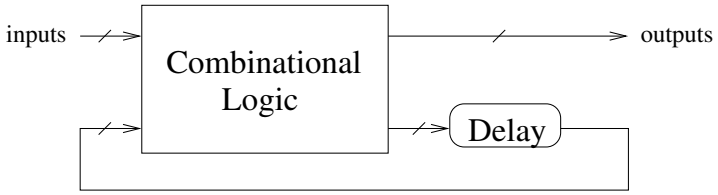
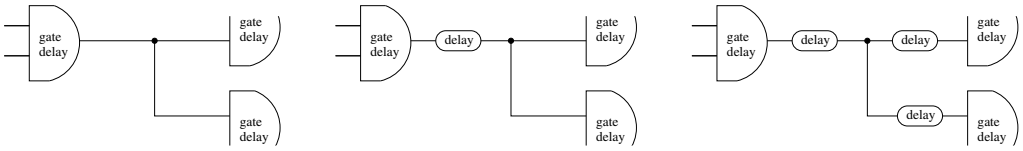


Figure 1.5: Huffman circuit model

design criteria such as area, speed, or power. The approach taken for the synthesis of synchronous state machines is to derive the synthesis problem into three steps. The first step is *state minimisation*, in which compatible states are merged to produce a simple flow table. The second step is *state assignment* in which a binary encoding is assigned to each state. The third step is *logic minimisation* in which an optimised netlist is derived from an encoded flow table. The design of Huffman circuits can follow the same three step process, but each step must be modified to produce correct circuits under an asynchronous timing model.

Huffman circuits are typically designed using the *bounded gate* delay and *wire delay* model. With this model, circuits are guaranteed to work, regardless of gate and wire delays as long as a bound on the delays is known. In order to design correct Huffman circuits, it is also necessary to put some constraints on the behaviour of the environment, namely when inputs are allowed to change. A number of different restrictions on inputs have been proposed, each resulting in variations of the synthesis procedure. The first is *single-input change (SIC)*, which states that only one input is allowed to change at a time. In other words, each input change must be separated by a minimum time interval. If the minimum time interval is set to be the maximum delay for the circuit to stabilise, the restriction is called *single-input change fundamental mode*. This is quite restrictive, though, so another approach is to allow *multiple-input changes (MIC)*. Again, if input changes are allowed only after the circuit stabilises, this mode of operation is called *multiple-input change fundamental mode*. An extended MIC model, referred to as *burst mode* [77], allows multiple inputs to change at any time as long as the input changes are grouped together in bursts.

Muller circuits Muller circuits [73] are designed under the *unbounded gate delay* model. Under this model, circuits are guaranteed to work regardless of gate delays, assuming that wire delays are negligible. This means that whenever a signal changes values, all gates it is connected to will see that change immediately. Such a model is called *speed-independent* (SI)[73] and is illustrated in Figure 1.6(a). A similar model, where only certain forks are *isochronic forks* is called *quasi-delay insensitive* (QDI)[67]. This model is depicted in Figure 1.6(b). Isochronic forks [6] are forking wires where the difference in delays between the branches is negligible.



(a) SI (b) QDI (c) DI

Figure 1.6: Illustration of delay models

Similar to the SI model is the *self-timed* model [92]. It contains a group of self-timed *elements*. Each element contains an *equipotential region*, where wires have negligible or well-bounded delay. An element itself may be an SI circuit, or a circuit whose correct operation relies on the use of local timing assumptions. However, no timing assumptions are made on the communication between regions, i.e. , communication between regions is delay-insensitive.

Muller circuit design requires explicit knowledge of the behaviour allowed by the environment. It does not, however, put any restriction on the speed of the environment. The design of Muller circuits requires a somewhat different approach as compared with traditional sequential state machine design. Most synthesis methods for Muller circuits translate the higher-lever specification into a state graph. Next, the state graph is examined to determine if a circuit can be generated using only the specified input and output signals. If two states are found that have the same value of inputs and outputs but lead through an output transition to different next states, no circuits can be produced directly. This ambiguity is known as an encoding conflict. In this case,

either the protocol must be changed or new internal state signals must be added to the design. The method of determining the necessary state variables is quite different from that used for Huffman circuits. Logic is derived using modified versions of logic minimisation procedures. The modifications needed are based upon the technology that is being used for implementation. Finally, the design must be mapped to gates in a given gate library. This last step requires a substantially modified technology mapping procedure as compared with traditional state machine synthesis methods.

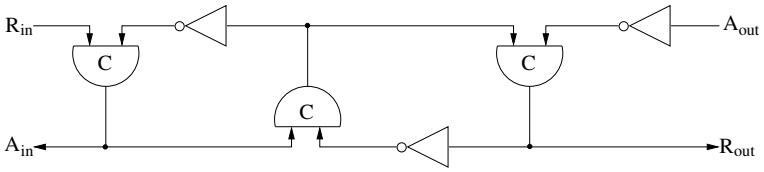
Delay-insensitive circuits A *delay-insensitive* (DI) circuit assumes that delays in both gates and wires are unbounded, Figure 1.6(c). This delay model is most realistic and robust with respect to manufacturing processes and environmental variations. If a signal fails to propagate at a particular point then the circuit will stop functioning rather than producing a spurious result.

The class of DI circuits is built out of simple gates and thus its operation is quite limited. Only very few circuits can be designed to be completely DI. Therefore, DI implementation is usually obtained from modules whose behaviour is considered to be DI on their interfaces.

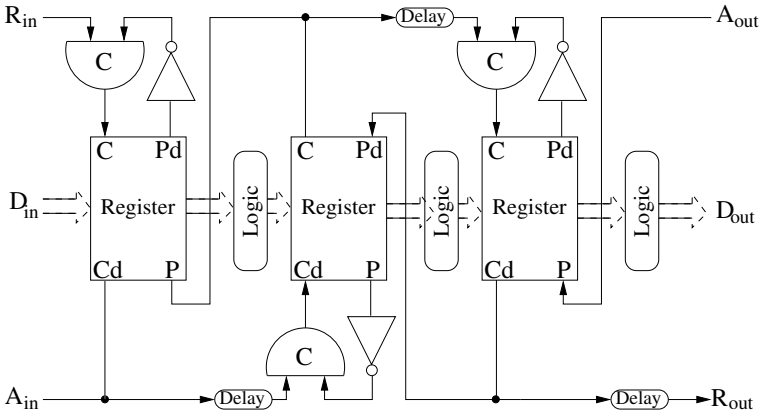
Several DI methodologies have been proposed. They are usually obtained from a specification in a high-level programming language such as Communicating Sequential Processes (CSP) [32] and Tangram [7]. The transformation from the specification to implementation, composed of common gates, is driven by the syntax and structure of the specification itself.

Micropipelines Micropipelines [100] are an efficient implementation of asynchronous pipelined modules. They were introduced as an alternative to synchronous *elastic* pipeline design, i.e. pipelines in which the amount of data contained can vary. However, they proved to be a very efficient and fast implementation of arithmetic units, and have been used in micropipelined systems such as the AMULET microprocessor [28].

The methodology does not fit precisely in the delay model classification, as it uses a bundled data communication protocol moderated by a delay-insensitive control circuit [30].



(a) control



(b) computation

Figure 1.7: Micropipeline structure

The basic implementation structure for a micropipeline is the control first-in first-out queue (FIFO) shown in Figure 1.7(a), where the gates labelled C are C-elements, which are elements whose output is *1* when all inputs are *1*, *0* when all inputs are *0*, and hold their state otherwise. The FIFO stores transitions sent to it trough R_{in}, shifts them to the right, and eventually outputs them through R_{out}.

The simple transition FIFO can be used as the basis for a complete computation pipeline as shown in Figure 1.7(b). The register output C_d is a delayed version of input C, and output P_d is a delayed version of input P. Thus, the transition FIFO in Figure 1.7(a) is embedded in Figure 1.7(b), with delays added. The registers are initially active, passing data directly from data inputs to data outputs. When a transition occurs on the C (capture) wire, data is no longer allowed to pass and the current values of the outputs are statically maintained. Then, once a transition occurs on P (pass) input, data is again allowed to pass from input to output, and the cycle repeats. The logic blocks between registers perform computation on the data stored in a micropipeline.

Since these blocks slow down the data moving through them, the accompanying control transition must also be delayed. This is done by adding delay elements.

1.5 Synthesis of control logic

There exist a number of approaches for the specification and synthesis of asynchronous circuits. They are based on state machines, Petri Nets (PNs) and high-level descriptions languages. State machines are the most traditional approach, where specifications are described by a flow table [102], and the synthesis is performed similar as in synchronous systems. The main characteristic of such an approach is that it produces a sequential model out of a possibly concurrent specification, where concurrency is represented as a set of possible interleaving resulting in a problem with the size of the specification. In order to specify concurrent systems PN [86] have been introduced. Rather than characterising system states, these describe partially ordered sequences of events. Methods for synthesis based on PN can be divided in two categories. The first category comprises techniques of direct mapping of PN constructs into logic, and the second category performs explicit logic synthesis from interpreted PNs. In high-level descriptions languages the system is specified in a similar way to conventional programming languages. Most of these languages are based on Communicating Sequential Processes (CSP) [32], which allow concurrent system to be described in a abstract way by using channels as the primary communication mechanism between sub-systems. The processes are transformed into low-level process and mapped directly to a circuit.

Aspect of PN based synthesis is within the scope of this work, therefore the problem of the synthesis of control circuits from PN[1] is outlined. PNs provide a simple graphical description of the system with the representation of concurrency and choice. In order to use PNs to model asynchronous circuits, it is necessary to relate transitions to events on signal wires. There have been several variants of PNs that accomplish this, including M-nets [91], I-nets [72], and change diagrams [46]. The most common are Signal Transition Graphs (STGs)[12]. They are a particular type of labelled Petri nets, where transitions are associated with the changes in the values of binary variables. These variables can,

[1] See next chapter for a formal definition of the PN theory.

for example, be associated with wires, when modelling interfaces between blocks, or with input, output and internal signals in a control circuit. STGs can be extracted from a high-level Hardware Description Language (HDL) and timing diagrams, respectively, which are popular amongst hardware designers.

A state-based synthesis [14][2] can be applied to design asynchronous control circuits from STGs. 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 output signals via their next-state functions obtained from the state graph. The state-based synthesis sufferers from problems of state space explosion and state assignment. The explicit representation of concurrency results in the state explosion problem. The state assignment problem arises when semantically different reachable states of an STG have the same binary encoding. If this occurs, the system is said to violate the *Complete State Coding* (CSC) property. Enforcing CSC is one of the most difficult problems in the synthesis of asynchronous circuits from STGs.

While the state-based approach is relatively simple and well-studied, the issue of computational complexity for highly concurrent STGs is quite serious due to the state space explosion problem. This puts practical bounds on the size of control circuits that can be synthesised using such techniques. In order to alleviate this problem, Petri net analysis techniques based on causal partial order semantics, in the form of Petri net unfoldings, are applied to circuit synthesis.

A *finite and complete unfolding prefix* of an STG is a finite acyclic net which implicitly represents all the reachable states of its STG together with transitions enabled at those states. Intuitively, it can be obtained through "unfolding" the STG until it eventually starts to repeat itself and can be truncated without loss of information, yielding a finite and complete prefix. Efficient algorithms exist for building such prefixes. 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.

The unfolding-based synthesis avoids the construction of reachability graph of an

[2]See Chapter 3 for a more formal definition of the state-based synthesis.

STG and instead use only structural information from its finite and complete unfolding prefix. These informations are used in [93] to derive approximated Boolean covers and recently in [42] they are used to build efficient algorithm based on the Incremental Boolean Satisfiability (SAT).

1.6 Motivation for this work

The explicit logic synthesis method based on state space exploration is implemented in the PETRIFY tool [18]. It performs the process automatically, after first constructing the reachability graph (in the form of a BDD [8]) of the initial STG specification and then, applying the theory of regions [19] to derive Boolean equations. An example of its use was the design of many circuits for the AMULET-3 microprocessor [28]. Since popularity of this tool is steadily growing, it is likely that STGs and PNs will increasingly be seen as an intermediate or back-end notation for the design of controllers.

During the synthesis the state assignment problem requires the refinement of the specification. PETRIFY explores the underlying theory of regions to resolve the CSC problem and relies on a set of optimisation heuristics. Therefore, it can often produce sub-optimal solutions or sometimes fail to solve the problem in certain cases, e.g. when a controller specification is defined in a compact way using a small number of signals. Such specifications often have CSC conflicts that are classified as irreducible by PETRIFY. Therefore, manual design may be required for finding good synthesis solutions, particularly in constructing interface controllers, where the quality of the solution is critical for the system's performance.

According to a practising designer [87], a synthesis tool should offer a way for the designer 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 by choosing how to refine the specification and where in the specification to perform transformations. State graphs distort the relationship of causality, concurrency and conflict and they are known to be large in size. PETRIFY offers a way to exhibit encoding conflicts by highlighting states in conflicts. However, all states which are in

conflict are highlighted in the same way. The extraction of information from this model is a difficult task for human perception. Therefore, a visualisation method is needed to help the designer in understanding the causes of encoding conflicts. Moreover, a method is needed which facilitates an interactive refinement of an STG with CSC conflicts.

Alternative synthesis approaches avoid the construction of the reachable state space. Such approaches include techniques based on either structural analysis of STGs or partial order techniques. The structural approach, e.g. in [11] performs graph-based transformations on the STG and deals with the approximated state space by means of linear algebraic representations. The main benefit of using structural methods is the ability to deal with large and highly concurrent specifications, that cannot be tackled by state-based methods. On the other hand, structural methods are usually conservative and approximate, and can only be exact when the behaviour of the specifications is restricted in some sense. The unfolding-based method represents the state space in the form of true concurrency (or partial order) semantics provided by PN unfoldings. The unfolding-based approach in [93] demonstrates clear superiority in terms of memory and time efficiency for some examples. However, the approximation based approach cannot precisely ascertain whether the STG has a CSC conflict without an expensive refinement process. Therefore, an efficient method for detection and resolution of CSC conflict in STG unfolding prefixes is required.

The problem of interactive refinement of CSC conflicts in the well established state-based synthesis can be tackled by employing STG unfolding prefixes. They are well-suited for both the visualisation of STG's behaviour and alleviating the state space explosion problem due to their compact representation of the state space in the form of true concurrency semantics. At the same time the CSC problem in the unfolding-based synthesis can be addressed resulting in an complete synthesis approach which avoids the construction of the state space.

1.7 Main contribution of this work

The main objective of this work is the introduction of transparency and interactivity in the synthesis of asynchronous systems from Signal Transition Graphs (STGs). The quality of the design of systems such as asynchronous interface controllers or synchronisers, which act as "glue logic" between different processes, affects performance. Automated synthesis usually offers little or no feedback to the designer making it difficult to intervene. Better synthesis solutions are obtained by involving human knowledge in the process. For the above reasons, the partial order approach in the form of STG unfolding prefixes has been applied. The approach was thus chosen for two reasons: (a) it offers compact representation of the state space with simpler structure and (b) it contributes to an unfolding-based design cycle, which does not involve building the entire state space. The work tackles the fundamental problem of state encoding conflicts in the synthesis of asynchronous circuits, and contributes to the following:

- **Detection of encoding conflicts**

  The approximation based approach proposed by Kondratyev et al. [47] for detection of encoding conflicts in STG unfolding prefixes has been extended by improving the detection and refinement algorithms. Its implementation was examined for efficiency and effectiveness. The experimental results show that the number of "fake" encoding conflicts reported due to over-approximation in the unfolding, which are not real state conflicts, is proportional to the number of states. The conflicts found by this approach must be refined by building their partial state space. This approach is therefore inefficient for STGs where concurrency dominates due to the high number of fake conflicts.

- **Visualisation of encoding conflicts**

  A new visualisation technique has been developed for presenting the information about encoding conflicts to the designer in a compact and easy to comprehend form. It is based on *conflict cores,* which show the causes of encoding conflicts, and on STG unfolding prefixes, which retain the structural properties of the initial specification whilst offering a simple model.

– **Resolution of encoding conflicts**

A resolution approach based on the concept of conflict cores has been developed to offer the designer an interactive resolution procedure, by visualising the conflict cores, their superpositions, and constraints on transformations.

The visualisation and resolution of encoding conflicts uses an alternative method for encoding conflict detection, which avoids the impracticality of the approximation based approach. The alternative method was developed by Khomenko et al. [40] and has proved to be very efficient in finding encoding conflicts in STG unfolding prefixes.

The visualisation and resolution of several types of encoding conflicts was undertaken. Depending on the type of conflicts eliminated an STG can be made implementable, or type or complexity of the derived functions can be altered. The resolution process employs several alternative transformations offering a wide range of synthesis solutions. These can be applied at different levels of interactivity. A tool has been developed offering an interactive resolution for complete state encoding, which can be applied in conjunction with already existing logic derivation procedures (PETRIFY etc.). The work also contributes to the idea of an alternative synthesis as a whole, based on partial order. It provides a method for resolving the complete state encoding problem, and thus completes the basic design cycle for this synthesis.

1.8 Organisation of thesis

This thesis is organised as follows:

Chapter 1 Introduction briefly outlines the area of asynchronous circuit design, the scope of the thesis, and presents the contribution of this work.

Chapter 2 Formal models presents basic definitions concerning Petri Nets, Signal Transition Graphs and net unfoldings which are used for the specification and verification of asynchronous circuits.

Chapter 3 Logic synthesis reviews existing approaches to logic synthesis and the problem of complete state encoding. Furthermore, it shows with an example the

design flow of an established state-based logic synthesis.

Chapter 4 Detection of encoding conflicts presents an approximation-based approached to identify encoding conflicts at the level of STG unfolding prefixes.

Chapter 5 Visualisation and Resolution of encoding conflicts proposes a new technique to visualise encoding conflicts by means of conflict cores at the level of STG unfolding prefixes. Furthermore, it also presents a resolution procedure based on the concept of cores which can be applied interactively or automated.

Chapter 6 Interactive synthesis discusses severals synthesis cases to demonstrate the advantages of the proposed interactive resolution of encoding conflicts based on core visualisation.

Chapter 7 Conclusion contains the summary of the thesis and presents directions of future work.

Chapter 2

Formal models

In this chapter the formal models used for the specification and verification of asynchronous circuits are presented. First, the Petri net (PN) model is introduced followed by one of its variants, the Signal Transition Graph (STG), which is used to model asynchronous circuits. Then, state-based and unfolding-based models of an STG are introduced, viz. the state graph (SG) and unfolding prefix, from which an asynchronous circuit can be derived.

2.1 Petri Nets

Petri nets (PNs) are a graphical and mathematical model applicable to many systems. They are used to describe and study information processing systems that are characterised as being concurrent, asynchronous, distributed, parallel and/or non-deterministic. As a graphical tool, PNs can be used as a visual communication aid similar to flow charts, block diagrams, and networks. In addition, tokens are used in these nets to simulate the dynamic and concurrent activities of systems. As a mathematical tool, it is possible to set up state equations, algebraic equations, and other mathematical models governing the behaviour of systems. Since the introduction of PNs in the early sixties in [84] they have been proposed for a very wide variety of applications. One such application is asynchronous circuit design.

This section introduces the basic concept of Petri nets system theory. The formal

definitions and notations are based on the work introduced in [17, 74, 86, 79].

Definition 2.1. Ordinary net

An *ordinary net* is a triple $N = (P, T, F)$, where

- P is a finite set of places,
- T is a finite set of transitions ($T \cap P = \emptyset$), and
- $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs (flow relation). $\Diamond$

All places and transitions are said to be *elements* of N. A net is *finite* if the set of elements is finite.

Definition 2.2. Pre-set and post-set

For an element x of $P \cup T$, its *pre-set*, denoted by ${}^\bullet x$, is defined by ${}^\bullet x = \{y \in P \cup T \mid (y, x) \in F\}$ and its *post-set*, denoted by $x^\bullet$, is defined by $x^\bullet = \{y \in P \cup T \mid (x, y) \in F\}$. $\Diamond$

Informally, the pre-set of a transition (place) gives all its input places (transitions), while its post-set corresponds to its output places (transitions). It is assumed that ${}^\bullet t \neq \emptyset \neq t^\bullet$ for every $t \in T$.

The states of a Petri net are defined by its markings. A *marking* of a net N is a mapping $M : P \to \mathbb{N}$ where $\mathbb{N} = \{0, 1, 2, ...\}$. A place p is marked by a marking M if $M(p) > 0$. The set of all markings of N is denoted by $\mathcal{M}(N)$.

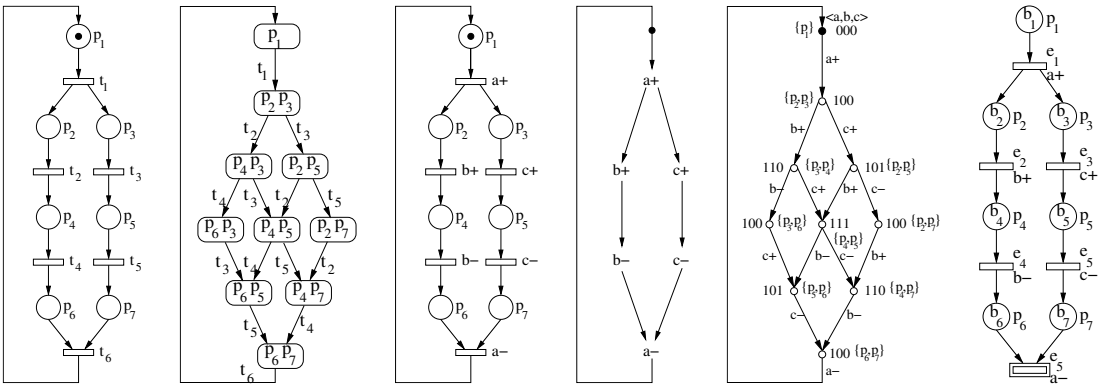
Definition 2.3. Petri net (PN)

A *PN* is a net system defined by a pair $\Sigma = (N, M_o)$, where

- N is a net, and
- $M_0 \in \mathcal{M}(N)$ is the initial marking of the net. $\Diamond$

A PN is a directed bipartite graph with two types of nodes, where arcs represent elements of the flow relation. In a graphical representation places are drawn as circles, transitions as bars or boxes, and markings as tokens in places. Transitions represent events in a

system and places represent placeholders for the conditions for the events to occur or for the resources that are needed. Each place can be viewed as a local state. All marked places taken together form the global state of the system. An example of a PN with the initial marking $M_0 = \{p_1\}$ is illustrated in Figure 2.1(a).



(a) PN (b) RG (c) STG (d) short-hand STG (e) SG (f) un-fold-ing prefix

Figure 2.1: An example: a PN and its RG, an STG, its SG and its unfolding prefix

It is often necessary to label transitions of a PN by symbols from some alphabet, e.g. by the names of signal transitions in a circuit. The labelling function need not to be injective, i.e. labelling can be partial (not all transitions are labelled).

Definition 2.4. Labelled Petri net (LPN)
A *labelled Petri net (LPN)* is a tuple $\Upsilon = (\Sigma, \mathcal{I}, \mathcal{O}, \ell)$, where

- Σ is a Petri net,
- $\mathcal{I} \cap \mathcal{O} = \emptyset$ are respectively finite sets of *inputs* (controlled by the environment) and *outputs* (controlled by the system), and
- $\ell : T \to \mathcal{I} \cup \mathcal{O} \cup \{\tau\}$ is a labelling function, where $\tau \notin \mathcal{I} \cup \mathcal{O}$ is a *silent action.* $\Diamond$

In this notion, τ's denote internal transitions which are not observable by the environment. An example of a labelled PN, where transitions are interpreted as signals of a circuits, is depicted in Figure 2.1(c).

2.1.1 Enabling and firing

The dynamic behaviour of a PN is defined as a *token game*, changing markings according to the *enabling* and *firing* rules for the transitions. A transition t in a PN system $\Sigma = (N, M_0)$ is enabled at a marking M if each place in its pre-set is marked at M.

Definition 2.5. Enabled transition

A transition $t \in T$ is *enabled* at marking M, denoted by $M[t\rangle$, iff

$$\forall p \in {}^\bullet t : M(p) > 0. \qquad \Diamond$$

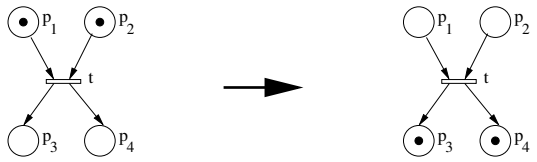
Once a transition t is enabled at marking M it may fire reaching a new marking M'. In a PN system a token is consumed from each place in the pre-set of t, while a token is added to every place in the post-set of t.

Definition 2.6. Transition firing

A transition $t \in T$ enabled at a marking M fires, reaching a new marking M', denoted by $M[t\rangle\, M'$ or $M \xrightarrow{t} M'$. The new marking M' is given by:

$$\forall p \in P : M'(p) = \begin{cases} M(p) - 1 & \text{if } p \in {}^\bullet t \setminus t^\bullet, \\ M(p) + 1 & \text{if } p \in t^\bullet \setminus {}^\bullet t, \\ M(p) & \text{otherwise.} \end{cases} \qquad \Diamond$$

An example of a transition firing rule is shown in Figure 2.2. According to the enabling and firing rule, the transition t is enabled and ready to fire (2.2(a)). When a transition fires it takes the tokens from the input places p_1 and p_2 and distributes the tokens to output places p_3 and p_4 (2.2(b)).



(a) transition enabled (b) firing complete

Figure 2.2: Example: An illustration of a firing rule

A (possibly empty) sequence of transitions including all intermediate transitions which have fired between two markings is called a firing sequence.

Definition 2.7. Firing sequence

Let $\sigma = t_1, ..., t_k \in T$ be a sequence of transitions. σ is a *firing sequence* from a marking M_1, denoted by $M_1[\sigma\rangle\, M_{k+1}$ or $M_1 \xrightarrow{\sigma} M_{k+1}$, iff a set of markings $M_2, ..., M_{k+1}$ exists such that: $M_i \xrightarrow{t_i} M_{i+1}$ for $1 \leq i \leq k$. ◊

A set of markings reachable from the initial marking M_0, denoted by $[M_0\rangle$, is called the *reachability set* of a net system. It can be represented as a graph, called *reachability graph* of the net, with nodes labelled with markings and arcs labelled with transitions.

Definition 2.8. Reachability graph (RG)

Let $\Sigma = (N, M_0)$ be a net system. Its reachability graph is a labelled directed graph $RG(N, M_0) = ([M_0\rangle\, , E, l)$, where

- $[M_0\rangle$ is the set of reachable markings,
- $E = ([M_0\rangle \times [M_0\rangle)$ is the set of arcs, and
- $l : E \rightarrow T$ is a labelling function such that $((M, M') \in E \wedge l(M, M') = t) \Leftrightarrow M \xrightarrow{t} M'$ ◊

The reachability graph of the PN system in Figure 2.1(a) is shown in Figure 2.1(b), where the nodes are labelled with the markings and the arcs are labelled with the transitions.

2.1.2 Petri net properties

In this subsection some simple but quite important behavioural properties of PN systems called boundedness, safeness, deadlock and reversibility are introduced.

Definition 2.9. Boundedness

Let $\Sigma = (N, M_0)$ be a net system.

- $p \in P$ is called *k-bounded* ($k \in \mathbb{N}$) iff $\forall M \in [M_0\rangle : M(p) \leq k$.
- Σ is k-bounded ($k \in \mathbb{N}$) iff $\forall p \in P : p$ is k-bounded. ◊

Boundedness is related to the places of the net and determines a bound to the number of tokens that a place may have at any given marking.

Definition 2.10. Safeness

A net system Σ is called *safe* iff it is 1-bounded. $\Diamond$

The places in a safe system can be interpreted as Boolean variables, i.e. a place p holds at a given marking M if $M(p) = 1$, or it does not hold any token $M(p) = 0$.

Definition 2.11. Deadlock

A marking is called a *deadlock* if it does not enable any transitions. $\Diamond$

A deadlock represents a state of a system from which no further progress can be made. Presence of deadlocks is typically regarded as an error in a system which operates in cycles. A net system Σ is *deadlock-free* if none of its reachable markings is a deadlock.

Definition 2.12. Home marking

A marking $M' \in [M_0\rangle$ is a home marking iff $\forall M' \in [M_0\rangle : M' \in [M\rangle$. $\Diamond$

The initial marking is a home marking if it can be reached from any other reachable marking.

2.1.3 Subclasses of Petri nets

Petri nets are classified according to their structural properties. Important subclasses of PNs are state machine, marked graph, and free-choice nets.

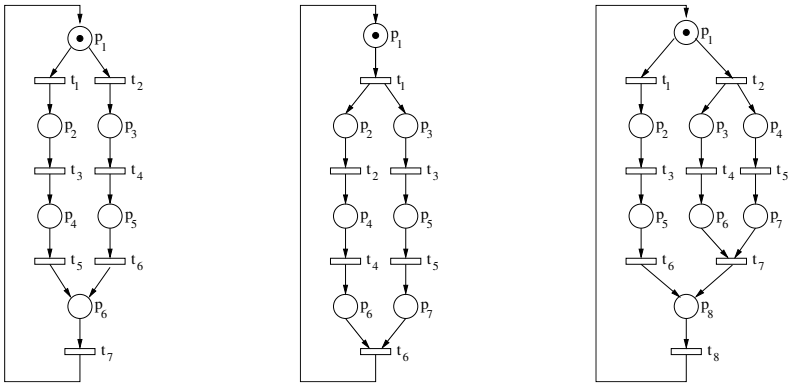
Definition 2.13. State Machine (SM) net

An SM is a Petri net Σ such that $\forall t_i \in T :\mid {}^\bullet t_i \mid = 1$ and $\mid t_i^\bullet \mid = 1$. $\Diamond$

In other words, every transition in an SM has only one input and only one output place. An example of an SM is illustrated in 2.3(a). State machines represent the structure of non-deterministic sequential systems.

Definition 2.14. Marked graph (MG) net

A MG is a Petri net Σ such that $\forall p_i \in P :\mid {}^\bullet p_i \mid = 1$ and $\mid p_i^\bullet \mid = 1$. $\Diamond$



(a) SM (b) MG (c) FC

Figure 2.3: Examples of subclasses of PN systems

A MG is a net in which each place has only one input and only one output transition. An example of a MG is shown in 2.3(b). Marked graphs represent the structure of deterministic concurrent systems.

Definition 2.15. Free-choice (FC) net

An FC net is a Petri net Σ such that for any $p_i \in P$ with the following is true: $\forall t_i \in p_i^\bullet :\mid {}^\bullet t_i \mid = 1$. $\Diamond$

Informally, if any two transitions t_1 and t_2 in a FC share the same input place p then p is the unique input place of both t_1 and t_2. An example of an FC is depicted in 2.3(c). This subclass allows to model both non-determinism and concurrency but restricts their interplay. The former is necessary for modelling choice made by the environment whereas the latter is essential for asynchronous behaviour modelling.

2.2 Signal Transition Graphs and State Graphs

This section introduces formally the specification language, the Signal Transition Graph (STG), and its semantical model, the State Graph (SG), together with their important properties.

2.2.1 Signal Transition Graphs

The Signal Transition Graph (STG) model was introduced independently in [12] and [89], to formally model both the circuit and the environment. The STG can be considered as a formalisation of the widely used timing diagram, because it describes the causality relations between transitions on the input and output signals of a specified circuit. It also allows the explicit description of data-dependent choices between various possible behaviours. STGs are interpreted Petri nets, and their close relationship to Petri nets provides a powerful theoretical background for the specification and verification of asynchronous circuits.

In the same way as an STG is an interpreted PN with transitions associated with binary signals, a State Graph (SG) is the corresponding binary interpretation of an RG in which the events are interpreted as signal transitions.

This section formally defines the event-based STG model and the corresponding state-based SG model. The notations and definitions are based on the one introduced in [14, 37].

Definition 2.16. Signal Transition Graph (STG)
An *STG* is a triple $\Gamma = (\Sigma, Z, \lambda)$, where

- $\Sigma = (N, M_0)$ is a net system based on $N = (P, T, F)$
- $Z = Z_I \cup Z_O$ is a finite set of binary signals divided into input (Z_I) and output or non-input (Z_O) signals, which generates a finite alphabet $Z^{\pm} = Z \times \{+, -\}$ of signal transitions, and
- $\lambda : T \rightarrow Z^{\pm} \cup \{\tau\}$ is a labelling function where $\tau \notin Z^{\pm}$ is a label indicating a *dummy* transition, which does not change the value of any signal. $\Diamond$

An STG is a labelled Petri net, whose transitions are interpreted as value changes on signals of a circuit. An example of an STG is presented in Figure 2.1(c). The labelling function λ does not have to be 1-to-1 (some signal transitions may occur several times in the net), and it may be extended to a particular function, in order to allow some transitions to be dummy ones, that is, to denote silent events that do not change the

state of the circuit. The signal transition labels are of the form z^+ or z^-, and denote the transitions of signals $z \in Z$ from 0 to 1 (rising edge), or from 1 to 0 (falling edge), respectively. Signal transitions are associated with the actions which change the value of a particular signal. The notation $z^\pm$ is used to denote a transition of signal z if there is no particular interest in its direction. Signals are divided into input and output signals (the latter may also include internal signals). Input signals are assumed to be generated by the environment, whereas output signals are produced by the logic gates of the circuit.

An STG inherits the operational semantics of its underlying net system Σ, including the notations of transition enabling and execution, and firing sequence. Likewise, STGs also inherit the various structural (marked graph, free-choice, etc.) and behavioural properties (boundedness, liveness, etc.).

For graphical representation of STGs a short-hand notation is often used, where a transition can be connected to another transition if the place between those transitions has one incoming and one outgoing arc as illustrated in Figure 2.1(d). When multiple transitions have the same label, superscripts are often used to distinguish them. In drawings, indexes separated by a slash, e.g. $a+/1$, are also used.

The initial marking of Σ is associated with a binary vector $v^0 = (v_1^0, ..., v_{|Z|}^0) \in \{0,1\}^{|Z|}$, where v_i^0 corresponds to the initial value of a signal $z_i \in Z$. The sequence of transitions σ is associated with an integer *signal change vector* $v^\sigma = (v_1^\sigma, ..., v_{|Z|}^\sigma) \in \mathbb{Z}^{|Z|}$, so that each v_i^σ is the difference between the numbers of the occurrences of z_i^+ and z_i^--labelled transitions in σ. A state of an STG Γ is a pair (M, v), where M is a marking of Σ and $v \in \mathbb{Z}^{|Z|}$ is a binary vector. The set of possible states of Γ is denoted by $\mathcal{S}(\Gamma) = \mathcal{M}(N) \times \mathbb{Z}^{|\mathbb{Z}|}$, where $\mathcal{M}(N)$ is the set of possible markings of the net underlying Γ. The transition relation on $\mathcal{S}(\Gamma) \times T \times \mathcal{S}(\Gamma)$ is defined as $(M, v) \xrightarrow{t} (M', v')$ iff $M \xrightarrow{t} M'$ and $v' = v + v^t$.

A finite firing sequence of transitions $\sigma = t_1, ..., t_i$ is denoted by $s \xrightarrow{\sigma} s'$ if there are states $s_1, ..., s_{i+1}$ such that $s_1 = s$, $s_{i+1} = s'$, and $s_j \xrightarrow{t_j} s_{j+1}$, for all $j \in \{1, ..., i\}$. The notation $s \xrightarrow{\sigma}$ is used if the identity of s' is irrelevant, to denote $s \xrightarrow{\sigma} s'$ for some $s' \in \mathcal{S}(\Gamma)$. In these definitions, σ is allowed to be not only a sequence of transitions,

but also a sequence of elements of $Z^{\pm} \cup \{\tau\}$, such that $s \xrightarrow{\sigma} s'$ means that $s \xrightarrow{\sigma'} s'$ for some sequences of transitions σ' such that $\lambda(\sigma') = \sigma$.

2.2.2 State Graphs

A State Graph (SG) is the corresponding binary interpretation of the STG. It corresponds to an RG of the underlying PN of its STG.

Definition 2.17. State Graph (SG)
A *state graph* of an STG is a quadruple $SG = (S, A, s_o, Code)$, where

- S is the set of reachable states,
- A is the set of arcs restricted by the transition relation $S \times T \times S$,
- $s_0 = (M_0, v^0)$ is the initial state, and
- $Code : S \rightarrow \mathbb{Z}^{|\mathbb{Z}|}$ is the state assignment function, which is defined as $Code((M, v)) = v$. ◊

The SG of the STG in Figure 2.1(d) is shown in Figure 2.1(e). Each state corresponds to a marking of the STG and is assigned a binary vector, and each arc corresponds to a firing of a signal transition. The initial state, depicted as a solid dot, corresponds to the marking $\{p_1\}$ with the binary vector 000 (the signal order is $\langle a, b, c \rangle$).

2.2.3 Signal Transition Graph properties

The STG model must satisfy several important properties in order to be implemented as an asynchronous circuit. These properties are formally introduced here.

Definition 2.18. Output signal persistency
An STG is called *output signal persistent* iff no non-input signal transition $z_i^{\pm}$ exited at any reachable marking can be disabled by a transition of another signal $z_j^{\pm}$. ◊

Persistency means that if a circuit signal is enabled it has to fire independently from the firing of other signals. However, persistency should be distinguished between input

and non-input signals. For inputs, which are controlled by the environment, it is possible to have non-deterministic choice. A non-deterministic choice is modelled by one input transition disabled by another input transition. For non-input signals, which are produced by circuits gates, signal transition disabling may lead to a hazard.

Definition 2.19. Consistency
An STG Γ is *consistent* if, for every reachable state $s \in S$, every finite execution sequence σ of Σ starting at the initial state have the same encoding $Code(s) = v^0 + v^\sigma \in \{0,1\}^{|Z|}$.

This property guaranties that, for every signal $z \in Z$, the STG satisfies the following conditions:

- the first occurrence of z in the labelling of any firing sequence of Γ starting from M_0 always has the same sign (either rising or falling), and
- the rising and falling labels z alternate in any firing sequence of Γ. $\Diamond$

At the level of the STG model the states are represented by pairs, marking and binary code, however, at the level of the circuit only their binary codes are represented. Thus it may be possible that two states of an STG have equal binary codes but have different markings and are semantically different (they generate different behaviour in terms of firing transition sequences). These states are indistinguishable at the circuit level. For example, in the SG in Figure 2.1(e) there are several states which have the same encoding but have different markings, e.g. the states encoded with 101 have different marking, $\{p_2, p_5\}$ and $\{p_5, p_6\}$.

Definition 2.20. Unique state coding (USC)
Two distinct states s and s' of SG are in *USC conflict* if $Code(s) = Code(s')$. An STG Γ satisfies the USC property if no two states of SG are in USC conflict. $\Diamond$

As shown in [12], the USC condition is sufficient for deriving Boolean equations from the SG without ambiguity, by simply using the binary codes of the states. It is however not a necessary condition because the problem of logic synthesis consists in deriving Boolean

next-state function for each output signal $z \in Z_O$, which requires the conditions for enabling output signal transitions to be defined without ambiguity by the encoding of each reachable state. This is captured by the CSC property.

Definition 2.21. Complete state coding (CSC)
Two distinct states s and s' of SG are in *CSC conflict* if $Code(s) = Code(s')$ and $Out(s) \neq Out(s')$, where $Out(s)$ is the set of enabled output signals at a state s, defined as $Out(s) = \{z \in Z_O | s \xrightarrow{\tau * z^{\pm}}\}$. An STG Γ satisfies the CSC property if no two states of SG are in CSC conflict. $\Diamond$

Logic synthesis derives for each signal $z \in Z_O$ a Boolean *next-state function* Nxt_z defined for every reachable state s as follows: $Nxt_z(s) = 0$ if $Code_z(s) = 0$ and no z^+-labelled transition is enabled at s, or $Code_z(s) = 1$ and a z^--labelled transition is enabled at s; and $Nxt_z(s) = 1$ if $Code_z(s) = 1$ and no z^--labelled transition is enabled at s, or $Code_z(s) = 0$ a z^+-labelled transition is enable at s. Moreover, the value of this function must be determined without ambiguity by the encoding of each reachable state, i.e. $Nxt_z(s) = F_z(Code(s))$ for some function $F_z : \{0,1\}^Z \rightarrow \{0,1\}$. F_z will eventually be implemented as a logic gate.

Definition 2.22. Complete state coding for z w.r.t. X (CSC^z_X)
Let s and s' be two distinct states SG, $z \in Z_O$ and $X \subseteq Z$. The states s and s' are in an CSC^z_X *conflict* if $Code_x(s) = Code_x(s')$ for all $x \in X$ and $Nxt_z(s) \neq Nxt_z(s')$. An STG Γ satisfies the CSC property for z (CSC^z) if no two states of SG are in CSC^z_Z conflict. $\Diamond$

Consequently, an STG Γ satisfies the CSC property if it satisfies the CSC^z property for each $z \in Z_O$. X is a support for $z \in Z_O$ if no two states of Γ are in CSC^z_X conflict. In such a case the value of Nxt_z at each state s of SG is determined without ambiguity by the encoding of s restricted to X. A support X of $z \in Z_O$ is *minimal* if no set $Y \subset X$ is a support of z. In general, a signal can have several distinct minimal supports.

The property of *normalcy* [99] is a necessary condition for STGs to be implemented as logic circuits which are built from monotonic gates.

Definition 2.23. Positive normalcy (p-normalcy)

An STG Γ satisfies the *p-normalcy* condition for output signal $z \in Z_O$ if for every pair of reachable states s' and s'', $Code(s') \leq Code(s'')$ implies $Nxt_z(s') \leq Nxt_z(s'')$. $\Diamond$

Similarly, the negative normalcy is defined.

Definition 2.24. Negative normalcy (n-normalcy)

An STG Γ satisfies the *n-normalcy* condition for output signal $z \in Z_O$ if for every pair of reachable states s' and s'', $Code(s') \leq Code(s'')$ implies $Nxt_z(s') \geq Nxt_z(s'')$.

Finally, the normalcy condition is defined, which implies CSC [99].

Definition 2.25. Normalcy

An STG Γ is *normal* if it is either p-normal or n-normal for each non-input signal. $\Diamond$

Note, that for any two states having the same encoding $Code(s') = Code(s'')$ the equivalence $Nxt_z(s') = Nxt_z(s'')$ follows from either definition 2.23 or 2.24. Thus, a normal STG for z automatically satisfies the CSC condition for z.

2.3 Branching processes

The concept of net unfolding is a well known *partial order semantic* first introduced in [76] and later described in more detail in [25] under the name of branching processes. In contrast to the interleaving semantic, the partial order semantic considers a single execution as a partially ordered set of events. The partial order semantic does not distinguish among total order executions that are *equivalent* up to reordering of independent events, thereby resulting in a more abstract and faithful representation of concurrency.

Unfoldings are usually infinite nets. However, it is possible to construct a finite initial part of the unfolding containing as much information as the unfolding itself. Several techniques for truncating unfoldings have been introduced [26, 27, 44, 70].

This section formally defines the concept of net unfoldings and their finite and complete prefixes. The definitions and notations are mainly based on those described

in [27, 37]. First, the notation for multiset is introduced. A *multiset* over a set X is a function $\mu : X \to \mathbb{N} = \{0, 1, 2, \ldots\}$. Note that any subset of X may be viewed (through its characteristic function) as a multiset over X. The notation $\{\!|h(x) \mid x \in \mu|\!\}$, or, alternatively, $h\{\!|\mu|\!\}$, where μ is a multiset over X and $h : X \to Y$ is a function, will be used to denote the multiset μ' over Y such that $\mu'(y) = \sum_{x \in X \wedge h(x)=y} \mu(x)$\$.

Definition 2.26. Structural conflict relation

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

In other words, y and y' are in structural conflict if two paths exist leading to y and y' and starting at the same place and immediately diverging (although later on they can converge again).

The unfolding of a net system is an occurrence net, a particularly simple net without cycles. The unfolding of a net system is behaviourally equivalent to it.

Definition 2.27. Occurrence net

An *occurrence net* is a net $ON = (B, E, G)$, where

- B is a set of conditions (places),
- E is a set of events (transitions), and
- G is a flow relation.

such that:

- $| {}^\bullet b | \leq 1$ for every $b \in B$,
- ON is acyclic, i.e. $\preceq$ is a partial order,
- ON is finitely preceded, i.e. for every $x \in B \cup E$, the set of elements $y \in B \cup E$ such that (y, x) belongs to the transitive closure of F is finite, and

– no event $e \in E$ is in self-conflict. ◊

Definition 2.28. Causal relation

The *causal relation*, denoted by $\prec$, is the irreflexive transitive closure of G. ◊

Two node in x and y are in $x \prec y$ if the net contains a path with at least one arc leading from x to y.

Definition 2.29. Concurrency relation

Two nodes $x, y \in B \cup E$ are *concurrent*, denoted by $x \parallel y$, if neither $x \# y$ nor $x \prec y$ nor $y \prec x$ holds. ◊

$Min(ON)$ denotes the set of minimal elements of $B \cup E$ with respect to the causal relation, i.e. the elements that have an empty preset. Since only those nets are considered in which every transition has a non-empty preset, the elements of $Min(ON)$ are conditions. An example of an occurrence net of a PN in Figure 2.4(a) is shown in Figure 2.4(b), where the following relationships holds: $e_1 \prec e_6$, $e_4 \# e_5$ due to the choice at b_1 and $e_6 \parallel e_7$.

The labelled occurrence nets obtained from net systems by unfolding are called branching processes, and have the following formal definition.

Definition 2.30. Branching Process

A *branching process* of a net system $\Sigma = (P, T, F, m_0)$ is a labelled occurrence net $\beta = (ON, h) = (B, E, G, h)$, where the labelling function h satisfies the following properties:

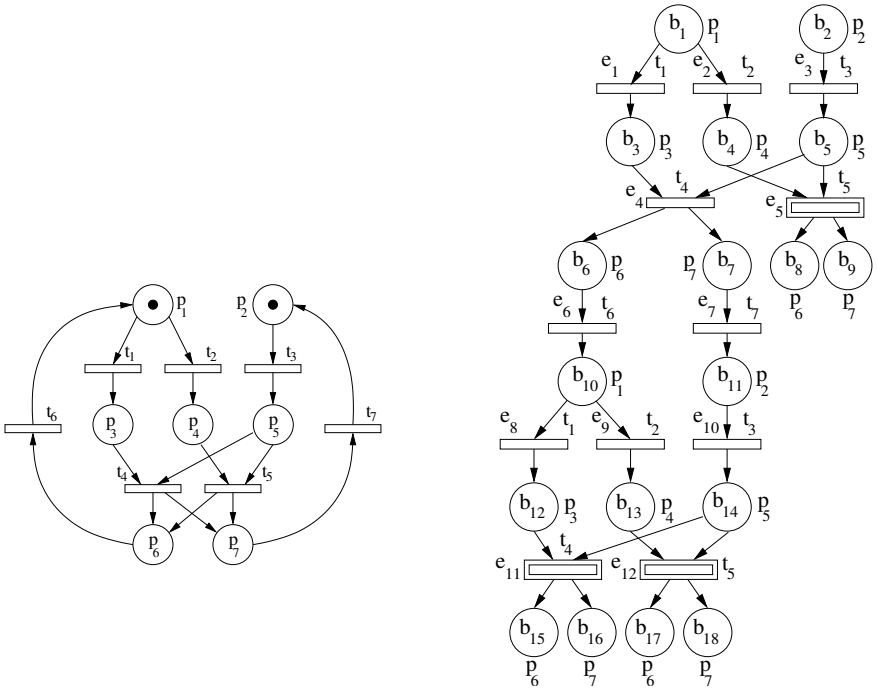
– conditions are mapped to places and events to transitions: $h(B) \subseteq P$ and $h(E) \subseteq T$,

– transitions environments are preserved: for each $e \in E, h\{|\ {}^\bullet e\ |\} = {}^\bullet h(e)$ and $h\{|\ e^\bullet\ |\} = h(e)^\bullet$,

– the branching process starts at the initial marking: $h(|\ Min(ON)\ |\} = m_0$, and

– β does not duplicate the transitions in Σ: for all $e_1, e_2 \in E$, if ${}^\bullet e_1 = {}^\bullet e_2$ and $h(e_1) = h(e_2)$ then $e_1 = e_2$. ◊

The labelling function h is represented as labels of nodes in the branching process in Figure 2.4(b). Branching processes differ on "how much they unfold". It is natural to introduce a prefix relation formalising the idea that "a branching process unfolds less than another".



(a) PN (b) finite complete prefix

Figure 2.4: A PN and one of finite and complete prefixes of its unfoldings

Definition 2.31. Prefix

A branching process $\beta' = (ON', h')$ of a net system Σ is a *prefix* of a branching process $\beta = (ON, h)$, denoted by $\beta \sqsubseteq \beta'$, if $ON' = (B', E', G')$ is a subnet of $ON = (B, E, G)$ containing all minimal elements and such that:

- if a condition b belongs to ON', then its input event $e \in {}^\bullet b$ in ON also belongs to ON' (if it exists),
- if an event e belongs to ON', then its input and output conditions ${}^\bullet e \cup e^\bullet$ in ON also belong to ON', and
- h' is the restriction of h to $B' \cup E'$. $\Diamond$

It is shown in [25] that a net system has a unique maximal branching process with respect to the prefix relation. This process is unique up to isomorphism, i.e. up to the renaming of conditions and events. This is the branching process that "unfolds as much as possible". It is called the *unfolding* Unf_{Σ}^{max} of the system. The unfolding of the PN system in Figure 2.4 is infinite.

A branching process has a (virtual) *initial event*, denoted by $\perp$, which has the post-set $Min(ON)$, empty preset, and no label. The initial event will be assumed to exist, without drawing it in a figure.

Definition 2.32. Process

A branching process is called a *process* π if for every its condition $b \in B$, $|b^\bullet| \leq 1$. $\Diamond$

Processes are a partial order analog of traces. The main difference between the processes and traces is that in the former the events are ordered only partially, and thus one process can correspond to several traces, which can be obtained from it as the linearisations of the corresponding partial order. A Petri net generates a set of processes much like it generates a language.

A process can be represented as a (perhaps infinite) labelled acyclic net, with places having at most one incoming and one outgoing arc. (And a branching process can be considered as overlayed processes.) A process is *maximal* if it is maximal w.r.t. $\sqsubseteq$, i.e. if it cannot be extended by new events. A maximal process is either infinite (though not every infinite process is maximal) or leads to a deadlock.

If π is a process and $E' \subseteq E$ is a set of events of the unfolding not belonging to π such that the events from π and E' together with their incident conditions induce a process, then this process will be denoted by $\pi \oplus E'$. Moreover, if π is finite and $U \subseteq T$, $\#_U\pi$ will denote the number of events of π with labels in U; furthermore, if $t \in T$ then $\#_t\pi = \#_{\{t\}}\pi$.

2.3.1 Configurations and cuts

In this subsection the most important theoretical notions regarding occurrence net, configurations and cuts, are introduced.

Definition 2.33. Configuration

A *configuration* of an occurrence net ON is a set of events $C \subseteq E$ such that for all $e, f \in C$, $\neg(e \# f)$ and, for every $e \in C$, $f \prec e$ implies $f \in C$. Since the assumption of the initial event $\perp$ it is additionally required that $\perp \in C$. $\Diamond$

Intuitively, a configuration is a partial order execution, i.e. an execution where the order of firing of some of its transitions is not important. It is crucial that all events of a configuration are closed under the causality relation (for every event in a configuration all its causal predecessors are in it, too) and are not in conflict. For example in Figure 2.4(b) the configuration $C = \{e_1, e_3, e_4\}$ corresponds to two totally ordered executions, $e_3 e_1 e_4$ and $e_1 e_3 e_4$. In Figure 2.4(b) C is a configuration, whereas $\{e_1, e_2, e_3\}$ and $\{e_4, e_7\}$ are not. The former includes events in conflict, $e_1 \# e_2$, while the latter does not include $e_1 \prec e_4$.

Definition 2.34. Local configuration

For every event $e \in E$, the configuration $[e] = \{f | f \preceq e\}$ is called the *local configuration* of e. $\Diamond$

The set of *causal predecessors* of e is denoted by $\langle e \rangle = [e] \setminus \{e\}$. The notation $C \oplus E'$ denotes the fact that $C \cup E'$ is a configuration and $C \cap E' = \emptyset$ for a set of events E'. Such an E' is a *suffix* of C, and $C \oplus E'$ is an *extension* of C.

The set of all finite (local) configurations of a branching process β is denoted by $\mathcal{C}^{\beta}_{fin}(\mathcal{C}^{\beta}_{loc})$, and the superscription β is dropped in the case $\beta = Unf^{max}_{\Sigma}$. The set of *triggers* of an event $e \in E$ is defined as $trg(e) = \max_{\prec}([e] \setminus \{e\})$. For example in Figure 2.4(b) $trg(e_4) = \{e_1, e_3\}$.

A marking of a net system is represented in a branching process as a cut.

Definition 2.35. Cut

A *cut* is a maximal (w.r.t. $\subseteq$) set of conditions B' such that $b || b'$, for all distinct $b, b' \in B'$. $\Diamond$

In other words, a cut is a maximal set of mutually concurrent conditions which is reached from $Min(ON)$. Finite configurations and cuts are tightly related. Let C be

a finite configuration of a branching process β. Then $Cut(C) = \bigcup_{e \in C} e^\bullet \setminus \bigcup_{e \in C} {}^\bullet e$ is a cut. In particular, given a configuration C the set of places $h(Cut(C))$ represent a reachable marking of Σ, which is denoted by $Mark(C)$. Loosely speaking, $Mark(C)$ is the marking reached by firing the configuration C. A marking M of Σ is *represented* in β if there is $C \in \mathcal{C}^\beta_{fin}$ such that $M = Mark(C)$. Every marking represented in β is reached in the original net system Σ, and every reachable marking of Σ is represented in the unfolding of Σ. For example in Figure 2.4(b), the cut $\{b_6, b_7\}$ corresponds to the configuration $C = \{e_1, e_3, e_4\}$, and the corresponding reachable marking of Σ is $\{p_6, p_7\}$.

2.3.2 Complete prefixes of PN unfoldings

There exist different methods of truncating PN unfoldings. The differences are related to the kind of information about the original unfolding which are to be preserved in the prefix, as well as to the choice between using either only local or all finite configurations. The former can improve the running time of an algorithm, and the latter can result in a smaller prefix.

A more general notion of completeness for branching processes was proposed in [37]. In [37] the entire set-up was generalised so that it is applicable to different methods of truncating unfoldings and, at the same time, it allows the expression of completeness with respect to properties other than marking reachability.

In order to cope with different variants of the technique for truncating unfoldings the cutting context was introduced [37], which generalised the whole set-up using an abstract parametric model. The first parameter determines the information which is intended to be preserved in a complete prefix. (In the standard case, this is the set of reachable markings.) The main idea behind it is to speak about finite complete configurations of Unf^{max}_Σ rather than reachable markings of Σ. Formally, the information to be preserved corresponds to the equivalence classes of some equivalence relation $\sim$ on $\mathcal{C}_{fin}$. The other parameters are more technical, they specify the circumstances under which an event can be designed as a cut-off event.

Definition 2.36. Cutting context

A *cutting context* is a triple $\Theta = (\approx, \lhd, \{\mathcal{C}_e\}_{e \in E})$, where

- $\approx$ is an equivalence relation on $\mathcal{C}_{fin}$,
- $\lhd$ is called an *adequate* order, which is a strict well-founded partial order on $\mathcal{C}_{fin}$ refining $\subset$, i.e. $C \subset C'$ implies $C \lhd C'$,
- $\approx$ and $\lhd$ are preserved by the *finite extensions*, i.e. for every pair of configurations $C \sim C'$, and for every suffix E *of* C there exists a finite suffix E' of C', such that
  - $C' \oplus E' \sim C \oplus E$, and
  - if $C' \lhd C$ then $C' \oplus E' \lhd C \oplus E$.
- $\{C_e\}_{e \in E}$ is a family of subsets of $\mathcal{C}_{fin}$. $\Diamond$

The main idea behind the adequate order is to specify which configurations are preserved in the complete prefix. The last parameter is needed to specify the set of configurations used later to decide whether an event can be designed as a cut-off event. For example, $\mathcal{C}_e$ may contain all finite configurations of Unf^{max}_{Σ} or , as is usually the case in practice, only the local ones. A cutting context Θ is *dense* (*saturated*) if $\mathcal{C}_e \supseteq \mathcal{C}_{loc}$ ($\mathcal{C}_e = \mathcal{C}_{fin}$), for all $e \in E$.

In practice, several cases of the adequate order $\lhd$ and the equivalence $\approx$ have been shown in the literature to be of interest. For example the adequate order in the original McMillan's algorithm [69] is $C \lhd_m C'$ if $|C| < |C'|$ and in [27] is $C \lhd_{sl} C'$ if $h\{|C|\} \ll_{sl} h\{|C'|\}$, where $\ll$ is an arbitrary total order on transitions of the original net system. The most widely used equivalence relation (see [26, 27, 31, 70]) is $C \approx_{mar} C'$ if $Mark(C) = Mark(C')$. Note that the equivalence classes of $\approx_{mar}$ correspond to the reachable marking of Σ. The equivalence for STG unfoldings used in [93] is $C \approx_{code} C'$ if $Mark(C) = Mark(C')$ and $Code(C) = Code(C')$.

Definition 2.37. Completeness

A branching process β is *complete* w.r.t. a set E_{cut} of events of Unf^{max}_{Σ} if the following holds:

– if $C \in \mathcal{C}_{fin}$, then there is $C' \in \mathcal{C}^{\beta}_{fin}$ such that $C' \cap E_{cut} = \emptyset$ and $C \sim C'$;

– if $C \in \mathcal{C}^{\beta}_{fin}$ such that $C \cap E_{cut} = \emptyset$, and e is an event such that $C \oplus \{e\} \in \mathcal{C}_{fin}$, then $C \oplus \{e\} \in \mathcal{C}^{\beta}_{fin}$.

A branching process β is complete if it is complete w.r.t. some set E_{cut}. $\Diamond$

Note that, in general, β remains complete after removing all events e for which $\langle e \rangle \cap E_{cut} \neq \emptyset$, i.e. without affecting the completeness a complete prefix can be truncated so that the events from E_{cut} (also referred as *cut-off* events) will be either maximal events for the prefix or not in the prefix at all. Note also that the last definition depends only on the equivalence and not on other components of the cutting context.

The branching process shown in Figure 2.4(b) is complete with respect to the set $E_{cut} = \{e_5, e_{11}, e_{12}\}$. The cut-off events of a finite and complete prefix are drawn as double boxes.

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

2.3.3 LPN branching processes

Finally, the definition of the LPN branching process and the STG branching process (where STG is a special kind of LPN) is presented.

Definition 2.38. LPN branching process
A *branching process* of an LPN $\Upsilon = (\Sigma, \mathcal{I}, \mathcal{O}, \ell)$ is a branching process of Σ augmented with an additional labelling of its events, $(\ell \circ h) : E \rightarrow \mathcal{I} \cup \mathcal{O} \cup \{\tau\}$. $\Diamond$

Processes of an LPN are defined in a similar way. If $\pi = (B, E, G, h)$ is a process of an LPN $\Upsilon = (\Sigma, \mathcal{I}, \mathcal{O}, \ell)$ then the *abstraction of* π w.r.t. ℓ is the labelled partially-ordered set (with the labels in $\mathcal{I} \cup \mathcal{O}$) $\mathrm{abs}_{\ell}(\pi) = (E', \prec', \ell')$ where: $E' = \{e \in E \mid \ell(h(e)) \neq \tau\}$; $\prec'$ is the restriction of $\prec$ to $E' \times E'$; and $\ell' : E' \rightarrow \mathcal{I} \cup \mathcal{O}$ is such that for all $e \in E'$, $\ell'(e) = \ell(h(e))$.

If ℓ is obvious from the context abs(π) will be written instead of abs$_\ell(\pi)$.

Definition 2.39. Input-proper LPN

An LPN is *input-proper* if no input event in its unfolding is triggered by an internal event, i.e. if for every event e in the unfolding such that $\ell(h(e)) \in \mathcal{I}$, and for every event $f \in trg(e)$, $\ell(h(f)) \neq \tau$. $\Diamond$

Finally, the STG branching process is presented. Note that an STG Γ can be considered as a special case of an LPN with the same underlying Petri net, $\mathcal{I} = Z_I$, $\mathcal{O} = Z_O$ and ℓ defined as

$$\ell(t) = \begin{cases} \tau & \text{if } \lambda(t) = z^{\pm} \wedge z \in Z_\tau \\ z & \text{if } \lambda(t) = z^{\pm} \wedge z \notin Z_\tau \ . \end{cases}$$

Definition 2.40. STG branching process

A *STG branching process* is a triple $\beta_\Gamma = (\Sigma, Z, \lambda)$, where

- $\Sigma = (N, M_0)$ is a net system,
- Z is a finite set of binary signals, and
- λ is a labelling function of events, $\lambda \circ h : E \rightarrow Z^{\pm} \cup \{\tau\}$. $\Diamond$

With any finite set of events $E' \subseteq E$, an integer *signal change vector* $v^{E'} = (v_1^{E'}, ..., v_k^{E'}) \in \mathbb{Z}^k$, such that each $v_i^{E'}$ is the difference between the number of z_i^+ and z_i^--labelled events in E'. It can be proven that $v^{E'} = v^\sigma$, where σ is an arbitrary linearision of $h\{|E'|\}$.

The function *Code* is extended to finite configurations of the branching process of Γ through $Code(C) = v^0 + v^C$. Note that $Code(C) = v^0 + v^\sigma$, for any linearision σ of $h\{|C|\}$, i.e. this definition is consistent with the definition of *Code* for *SG* in Definition 2.17 in the sense that the pair $(Mark(C), Code(C))$ is a state of *SG*, and for any state s of *SG* there exists a configuration C in the unfolding of Γ such that $s = (Mark(C), Code(C))$.

It is important to note that, since the states of an STG do not necessarily correspond to its reachable markings, some of the states may be not represented in the prefix built using the cutting context in [26, 27]. It was suggested in [93] to restrict the cut-off

criterion by requiring that not only should the final markings of two configurations be equal, but also their codes. This can be formalised by choosing the equivalence relation $\approx_{code}$ rather than $\approx_{mar}$ in the cutting context (see section 2.3.2).

In this thesis it is assumed that Θ is a dense cutting context with the equivalence relation $\approx_{code}$, $\Gamma = (\Sigma, Z, \lambda)$ is a consistent and bounded STG, and $\Gamma^{\Theta} = (B, E, G, \mathcal{M}_{in})$ is a safe net system built from a finite and complete prefix $Pref_{\Gamma} = (B, E, G, h)$ of the unfolding of Γ, where $\mathcal{M}_{in}$ is the canonical initial marking of Γ^{Θ} which places a single token in each of the minimal conditions and no token elsewhere. The assumption of Θ is not problematic because most existing algorithm use this dense cutting context. The finite and complete unfolding prefix of the STG in Figure 2.1(c) is illustrated in Figure 2.1(f).

Chapter 3

Logic synthesis

A method for designing asynchronous control circuits, i.e. those circuits that synchronise the operations performed by the functional units of data-path through handshake protocols, is described in this chapter. The design is based on logic synthesis from Signal Transition Graphs (STGs). The STG model offers important advantages to the asynchronous controller and interface design. STGs are very similar to Timing Diagrams, which can be seen as a conventional pragmatic design notation, and they are based on Petri nets, which offer a formally sound theory.

In the previous chapters the motivation for, and the concept of, asynchronous design was introduced and the main formalism of STGs was defined. In this chapter the synthesis process of an asynchronous circuit is described. The process involves generating a network of gates that correctly implement the specified behaviour. After a review of related work, the conditions for the implementation of speed-independent (SI) circuit are presented. Then, the process of synthesis of speed-independent circuits is described, and illustrated by an example. Finally, the problems and the approaches for ensuring complete state encoding is addressed.

3.1 Related work

There exist a variety of approaches to the synthesis of SI circuits from STG specifications. An overview is presented in Figure 3.1. These approaches can be classified into

two major groups, viz. state-based and event-based approaches. The first approach constructs an SG and then computes a consistent and complete binary encoding (if it exists) in order to synthesise the circuit from the encoding. This approach is well studied (see e.g. [14, 48, 53, 104]). This method is used in tools such SIS [96], ASSASSIN [54] and PETRIFY [18]. To gain efficiency, PETRIFY uses symbolic techniques, such as Binary Decision Diagrams (BDDs) [48, 82], to represent the state space. An obvious practical limitation of the state-base synthesis approach is the potential combinatorial growth of the number of reachable states. To cope with the state explosion problem Chu [12] suggested a method for decomposing an STG into several smaller ones. However, this method can only deal with very restrictive net classes, e.g. each transition label is allowed only once and structural conflict can only occur between input signals. A similar decomposition method is described in [3], where only MG with only output signals are considered. In [106] a method based on [12] was proposed, which can be applied to more general class of PN. In particular, the nets is not restricted to live and safe free-choice nets or to MG, and it is not required that reach label occurs only ones. The decomposition is done by partitioning the set of output signals and generating components that produce these outputs.

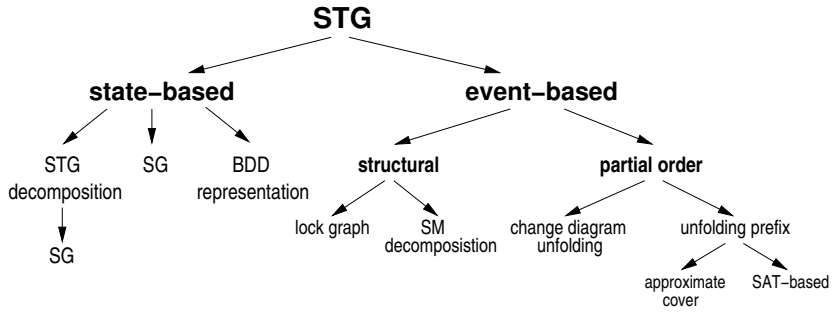


Figure 3.1: Overview of approaches for STG-based logic synthesis

The event-based approach avoids the construction of the full reachable state space. It includes techniques either based on structural analysis of STGs [80, 78, 81, 111] or partial order techniques [46, 41, 71, 95, 94, 108, 109]. The earliest structural approach was based on analysis of coupledness classes [108] and [109]. The structural method proposed in [111] uses structural information of the STG by means of lock relations between signals to synthesise a circuit. It extends the lock graph framework presented

in [55, 103], and avoids the construction of the state space. However, both coupledness and lock relation techniques were restricted to a sub-class of PN, marked graphs. The method in [78, 80, 81] uses SM decomposition of STGs to obtain concurrent relations between signal transitions. Using these relations, this method finds an approximate implementation avoiding the exploration of the state space. Although it demonstrated impressive results, it is restricted to FC specifications.

Partial order techniques have been used in [71] to derive logic functions from PN unfoldings. However, this work is based on restoring the state space from the partial order model and is therefore also prone to state explosion. The work in [46] uses change diagram unfoldings to synthesise SI circuits. However, the specification is restricted by the absence of choice. This work was a significant step in the development of the approach in [95, 94]. The synthesis tool Punt [93] inherits this approach, which uses STG unfoldings to derive logic implementations. This approach is based on the idea of finding approximated Boolean covers from structural information, i.e. from the conditions and events of the unfolding, as opposed to the use of exact Boolean covers for markings and excitation regions extracted from the reachability graph. The results demonstrated clear superiority in terms of memory and time efficiency for some examples. The main shortcoming of this work was that its approximation and refinement strategy was straightforward and could not cope well with *don't care* state subsets, i.e. sets of states which would have been unreachable if the exact reachability analysis was applied. As a result it cannot be precisely ascertained whether the STG has a CSC conflict without an expensive refinement process.

The approach in [41] uses an efficient algorithm based on the Incremental Boolean Satisfiability (SAT) for logic synthesis. It only uses the information about causality and structural conflicts between the events involved in a finite and complete prefix of an STG unfolding, thus avoiding the construction of its reachability graphs. Experimental results show that this technique leads not only to huge memory saving when compared with state-based methods, but also to significant speedups in many cases.

It can be seen that there are several approaches proposed for the synthesis of SI, some of which are restricted by the form of the specification. The traditional state-base

approach is well defined and supported by tools such as PETRIFY, but suffers from state space explosion. The partial order synthesis approach offers an alternative to the state-based approach. The results in [41, 95, 94] are encouraging and, together with [40, 60], the detection and resolution process of state conflicts (see chapter 5), form a complete design flow for complex gate synthesis of asynchronous circuits based on the complete and finite prefix of STG unfoldings.

3.2 Implementation as logic circuit

The aim of the synthesis methodology is to derive a SI circuit that realises the specified behaviour. Speed independence is a property that guarantees correct behaviour with the assumption that all gates have an unbounded delay and all wires have a negligible delay. A specification must fulfil certain conditions to be implementable as an SI circuit. The properties that guarantee the existence of hazard-free implementation for an STG are the following:

- boundedness
- consistency
- complete state coding (CSC)
- output persistency

Boundedness of an STG guarantees that the final circuit has a finite size. The consistency property of an STG ensures that the rising and falling transitions for each signal alternate in all possible runs of the specification. The CSC condition guarantees that there are no two different states with the same signal encoding and different behaviour of the non-input signals. Whilst the previous three properties aim to guarantee the existence of a logic implementation of the specified behaviour, the output persistency property of an STG aims to ensure the robustness of the implementation under any delay of the gates of the circuit. The condition for the implementation of an STG can be formulated as follows.

Proposition 3.1. *[12] An STG is implementable as an SI circuit iff it is bounded, consistent, output-persistent and satisfies CSC.*

3.3 State-based logic synthesis

The objective of logic synthesis is to derive a gate netlist that implements the behaviour defined by the specification. The design flow for the synthesis of asynchronous control circuits is illustrated in Figure 3.2. The states are derived and encoded from the STG specification. If the state encoding is not complete, the specification must be modified by, for example, introducing additional internal signals or reducing concurrency. From the completely encoded state space the behaviour of each non-input signal is determined by calculating the next value expected at each state. The state space is partitioned for each signal in ON-set, OFF-set and DC-set. The ON-set for a signal z is the set of all states in which z is either excited to rise or stable high. The OFF-set is the set of all states in which z is either excited to fall or stable low. The DC-set is the set of all unreachable states. Once the next-state functions have been derived Boolean minimisation can be applied to obtain a logic equation that implements the behaviour of signals. In this step it is crucial to make efficient use of the DC-set derived from those binary codes not corresponding to any states of the specification. Each Boolean function can be implemented as one *atomic complex gate*. This requires that all internal delays within each gate are negligible and do not produce any spurious behaviour observable from the outside. Complex gates are assumed to be implemented as complementary pull-up and pull-down networks in static CMOS. They are adapted to the available technology, possibly based on standard cell or library generators. Other implementation strategies use generalised C-elements [67, 68] or standard C-elements [4, 50].

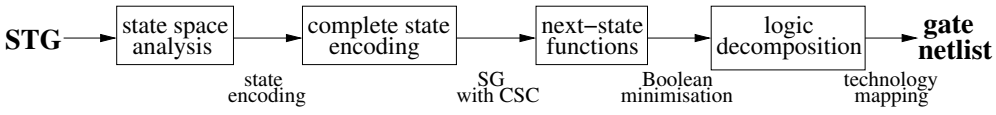


Figure 3.2: Design flow for asynchronous control circuits

The implementation of the next-state functions can be too complex to be mapped into one gate available in the library. Furthermore, the solution can require the use

of gates which are not typically present in standard libraries. Complex Boolean functions can be decomposed into elementary gates from a given library. However, the logic decomposition in asynchronous design is difficult without introducing hazards. The decomposition of a gate into smaller gates must preserve not only the functional correctness of a circuit but also speed independence, i.e. hazard freedom under unbounded gate delays. Several approaches have been proposed for the decomposition of complex gates. For a brief review see [16, 49].

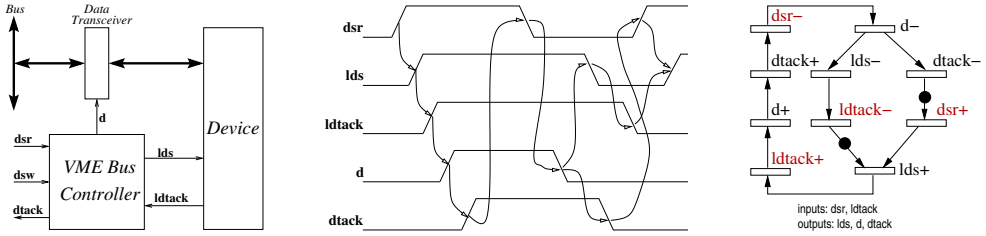
The assumption that the components of a system can have arbitrary delays may be too conservative in practice and can lead to inefficient circuits. The knowledge of the timing behaviour of some components can be used to simplify the functionality of the system to derive simpler and faster circuits. Synthesis under relative timing assumptions is described in detail in [14].

Analysis and verification are used at different stages of the synthesis process. These involve checking that the conditions are sufficient for implementation of an STG by a logic circuit. Other properties of the specification can be of interest as well, e.g. absence of deadlocks, fairness in serving requests, etc. If necessary, the initial STG is modified to make it implementable as an SI circuit.

3.4 Synthesis example

In this section the synthesis process is described on a simple example based using a VME-bus controller [17, 109]. The steps of the design process are supported by the synthesis tool PETRIFY. A VME-bus controller reads from a device to a bus and writes from the bus into the device. Its block diagram is illustrated in Figure 3.3(a). According to its functionality, the controller has three sets of handshake signals: those interacting with the bus (*dsr*, *dsw* and *dtack*), those interacting with the device connected to the bus (*lds* and *ldtack*), and that controlling the transceiver that connects the bus with the device (*d*).

The controller behaves as follows. A request can be received to read from the device or to write into the device by the signals *dsr* or *dsw*, respectively. In the case of the



(a) interface

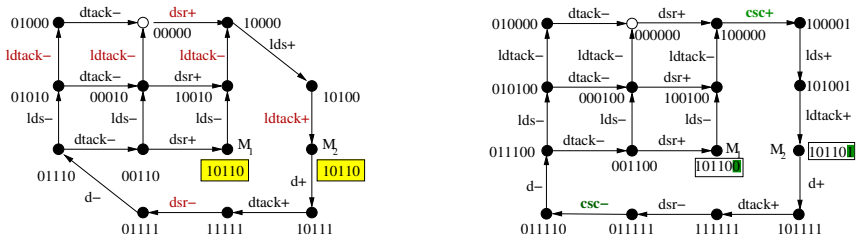
(b) timing diagram for the read cycle

(c) STG for the read cycle

Figure 3.3: VME-bus controller

read cycle being activated, a request to read is made through the signal *lds*. When the device has the data ready, and this is acknowledged by *ldack*, the controller opens the transceiver to transfer data to the bus. This is done using signal *d*. In the write cycle, data is first transferred to the device. Next, the write request is made by the signal *lds*. Once the device acknowledges the reception of data by the signal *ldtack*, the transceiver is closed in order to isolate the device from the bus. Each transaction is completed by a four-phase protocol for all interface signals, seeking for maximum parallelism between the bus and the device operations.

The behaviour of the read cycle is described as a timing diagram in Figure 3.3(b) and its corresponding STG 3.3(c). The behaviour of the input signals (*dsr*, *dsw* and *ldtack*, depicted in red) is determined by the environment, and the behaviour of the output signals (*d*, *dtack* and *lds*) is determined by the system, and is the one that must be implemented by the circuit.



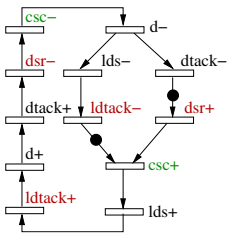
(a) SG with CSC conflict

(b) SG satisfying CSC

inputs: *dsr*, *ldtack*; **outputs:** *lds*, *d*, *dtack*; **internal:** *csc*
signal order: *dsr*, *dtack*, *lds*, *ldtack*, *d*, (*csc*)

Figure 3.4: CSC refinement

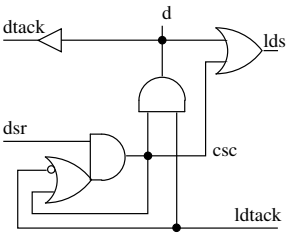
In order to derive Boolean equations from the specification, it is necessary to calculate the encoding of all the states of the system. The SG of the read cycle specified in 3.3(c) is illustrated in Figure 3.4(a). The states correspond to markings of the corresponding STG and are associated with a binary vector. The SG in Figure 3.4(a) has an ambiguous state encoding. Two states have different markings, M_1 and M_2, but have the same binary encoding, 10110. Moreover, they enable different sets of output signals, $Out(M_1) = \{lds\} \neq Out(M_2) = \{d\}$, thus a CSC conflict exists. This means that, e.g. the value of the next-state Boolean function for signal *lds* $F_{lds}(1, 0, 1, 1, 0)$ is ill-defined. It should evaluate to 0 according to the state M_1, and 1 according to M_2, hence *lds* is not implementable as a logic gate. A similar problem occurs for the signal *d*. To cope with this, an additional signal helping to resolve this CSC conflict is added to the SG in such a way, that the resulting STG preserves the properties for implementation. This process is done automatically by the tool PETRIFY. The resulting SG is depicted in 3.4(b), where the states previously in CSC conflict are made distinct by the newly inserted signal *csc*. The insertion can be performed manually if the designer is not satisfied with the automatic resolution. However, the designer must first understand what causes the conflicts in order to eliminate them. This process is difficult, especially if it is performed on the SG, because of its large size and tendency to obscure the causal relationship and concurrency. Guaranteeing CSC is one of the most difficult problems in the synthesis of asynchronous circuits. The next chapters are devoted to the detection, visualisation, and resolution of state coding conflicts. The next section gives a brief overview of the methods used to solve this problem.



(a) STG with CSC

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

(b) equations



(c) implementation

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

Figure 3.5: Read cycle VME-bus controller implementation

Once the CSC property is established, for each non-input signal the next-state functions are derived from the SG, using Boolean minimisation. It is crucial to make use of the don't care conditions derived from those binary codes not corresponding to any state of the specification. From the refined read cycle of the VME-bus shown in Figure 3.5(a) the equations are obtained (see Figure 3.5(b)). The next-state functions are implemented with only one atomic complex gate per signal, i.e. the internal delay within each gate is negligible. A possible hazard free gate implementation is shown in Figure 3.5(c). Note that the signal *dtack* is merely implemented as a buffer. It is not enough to implement it with a wire because the specification indicates that the transitions on *dtack* must always occur after the transitions on *d*. Thus, the buffer introduces the required delay to enforce the specified causality. If the logic functions are too complex or are not available in a library, they can be decomposed.

3.5 State encoding problem

The CSC requirement, as already noted, is one of the main problems in the synthesis of asynchronous circuits. The CSC condition requires that the binary encoding of every state of an STG is unambiguous with respect to the internal behaviour of a system, i.e. every pair of different states that are assigned the same binary code enable exactly the same set of non-input signals. A specification which does not satisfy the CSC property can be transformed in such a way that its behaviour remains externally equivalent to the original specification. This was the case in the example presented in the previous section, where an additional internal signal was inserted to resolve the conflict. Another technique to eliminate CSC is to reduce concurrency between the events in the specification.

Figure 3.6 shows several approaches for the resolution of the state encoding problem. These are divided into state-based and event based approaches, which are described as follows:

- State-based approach

  This approach constructs the reachability graph of the initial STG specification

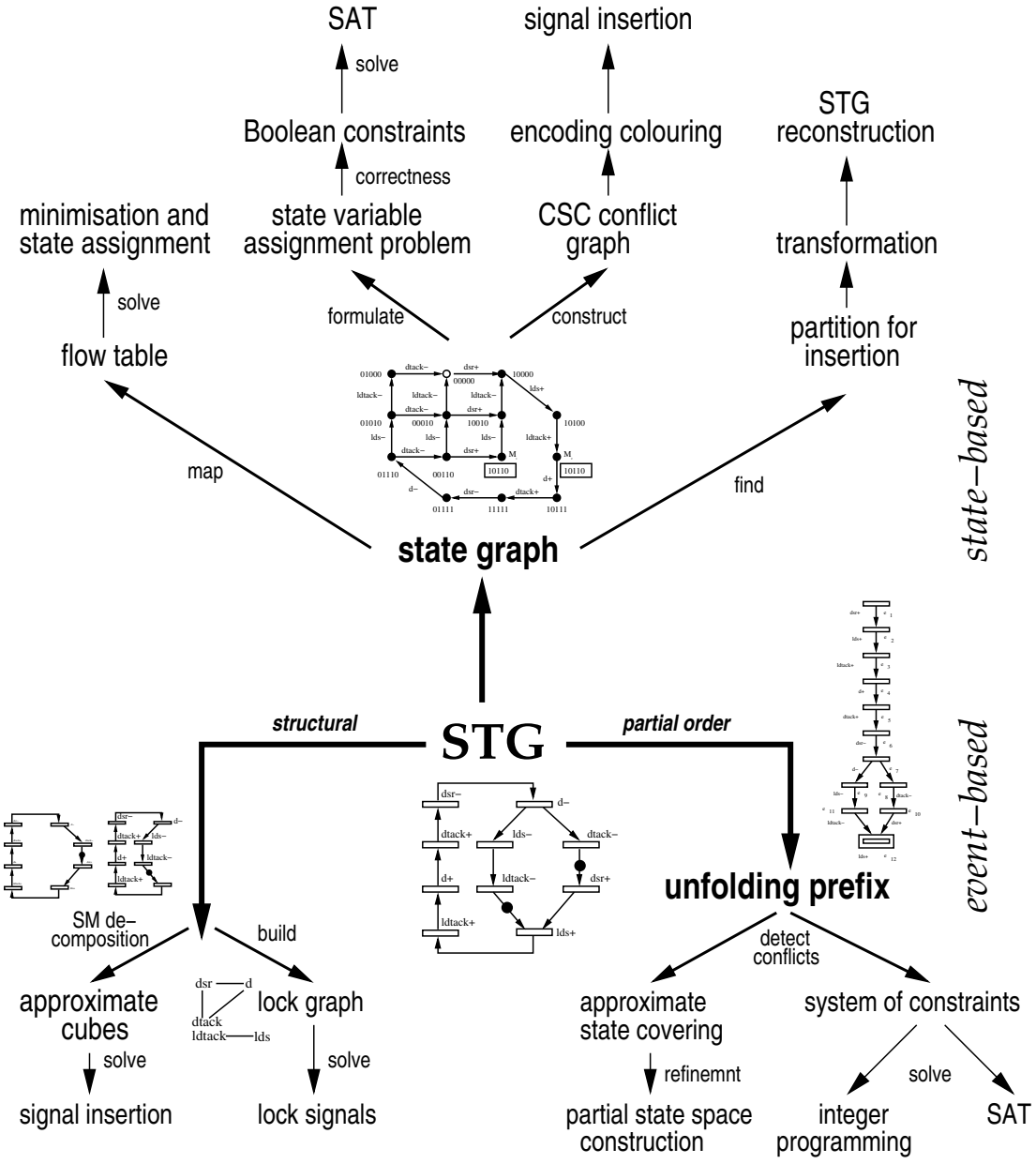


Figure 3.6: Approach overview for the resolution of the CSC problem

to resolve the state encoding problem. Whilst this approach is convenient (due to the nature of the problem) it suffers from the state explosion problem. To gain efficiency BDD-based techniques are used to represent the state graph.

- flow table

  An approach was proposed in [52] to solve the state coding problem at the SG level by mapping an initial SG into a flow table synthesis problem. The state coding problem is then solved by using flow table minimisation and state assignment methods. This approach is restricted to live and safe free-choice STGs and cannot process large SGs due to limitations of the classical state assignment problem.

- generalised assignment

  A general framework for state assignment is presented in [104]. The CSC problem is formulated as a search for a state variable assignment on the SG. The correctness conditions for such assignments are formulated as a set of Boolean constraints. The solution can be found using a Boolean satisfiability solver. Unfortunately, this approach only allows relatively small specifications (hundred of states) to be handled because the computational complexity of this method is doubly exponential to the number of signals in the SG. Although [29] presented a method to improve effectiveness by means of the preliminary decomposition of the satisfiability problem. The decomposition may produce sub-optimal solutions due to the loss of information incurred during the partitioning process. Moreover, the net contraction procedure used to decompose the problem has never been formally defined for non-free-choice STGs.

- CSC conflict graph

  In [46] another method based on state signal insertions at the SG level has been proposed. The problem is solved by constructing a CSC conflict graph between the excitation regions of an SG, a set of states in which a signal is enabled to change its value, and then the graph is coloured with binary

encoded colours. Each part of this code corresponds to a new state signal. After that, new state signals are inserted into the SG using the excitation regions of the original or previously inserted signals. The main drawback of this approach is its limitation to STGs without choices. This limitation was overcome in [110] which suggested partitioning the state space into blocks with no internal CSC conflicts and then insert new signals based on excitation regions of these blocks. However, restricting an insertion to excitation regions significantly limits the capabilities of this method.

- combination of region and SIP-set

  The method in [13, 19] is probably the most efficient and generally published so far. It is based on a combination of two fundamental concepts. One is the notation of regions of states in an SG. Regions correspond to places in the associated STG. The second concept is a speed-independence preserving (SIP) set, which is strongly related to the implementation of the specification in logic. The method is well suited for symbolic BDD representation of the main objects in the insertion procedure. This feature expands the capabilities for the state-base approach, and allows the resolution of CSC conflict for an SG with hundreds of thousands of states.

– Event-based approach

  Several event-based approaches have also been proposed, which do not generate the state space, and thus avoid the state explosion. However, many of them are limited to some sub-nets of PN or have expensive refinement procedures.

  - Structural approach

    This approach uses structural information derived directly from the STG without doing state traversal. They avoid state explosion and therefore can process large specification if some additional constraints on the structure of the STG are given. Such constraints limit the design space and do not produce solutions for some practical specifications.

    * coupledness and lock relations

The techniques in [103, 111] are based on lock relations among signals (similar relations where called coupledness in [108, 109]) and treat the state encoding problem wholly in the STG domain. However, their STG descriptions are restricting to MG having only one falling and rising transition for each signal. Furthermore, the more constrained USC property is considered. The work in [56] extends the lock relation framework to test the CSC condition (in [56] the term realisable state coding is used) and is applied to free-choice STG but shows an exponential worst case complexity.

  * approximation-based

    The approach in [79] uses structural STG analysis for the state encoding of FC STGs. It approximates the binary codes of the markings in the STGs by a set of cubes, and a set of SM-components to describe the order in relation between the transitions. A signal insertion algorithm is used to resolve CSC conflicts, which uses the set of SM-components to guarantee the consistency of the transformed STG.

- Partial order approach

  This approach uses the finite and complete unfolding prefixes of STGs to detect CSC conflicts. It allows not only finding the states which are in conflict, but also to derive execution paths leading to them without performing a reachability analysis. This provides a basis for the framework for resolution of encoding conflict described in chapter 5, which uses the set of pairs of configurations representing encoding conflicts as an input.

  * approximated state covering

    In [47] an approximation-based approach for identifying CSC conflicts was presented. A necessary condition for CSC conflicts to exist exploits partial coding information about conditions, which is made available from the computation of a maximal tree in an STG unfolding prefix. Whilst this condition coincides with real conflicts in many cases, and is computationally efficient (quadratic in the size of the unfolding), it

may report a so-called fake conflict. A refinement technique aimed at resolving such situations, at the expense of extra computational cost, is applied. This technique limits the search to these parts of the unfolding that may potentially exhibit a fake conflict. Those parts need explicit state traversing, which is of exponential complexity. The extension of the approach in [47] is described in the next chapter, which is based on the results developed in [62, 63].

* system of constraints

  The techniques in [39, 40] also use STG unfolding prefixes for the detection of CSC conflicts. These techniques are based on integer programming and SAT, respectively. The notation of CSC conflict is characterised in terms of a system of constraints derived from the information about causality and structural conflicts between the events involved in a finite and complete prefix of its STG unfolding. Those techniques achieve significant speedups compared with state-based methods, and provide a basis for the framework for the resolution of encoding conflicts, which is presented in chapter 5.

Chapter 4

Detection of encoding conflicts

State coding conflict detection is a fundamental part of the synthesis of asynchronous concurrent systems from their specifications as STGs. There exist several techniques for the detection of CSC conflicts as described in section 3.5. The work in [47] proposed a method for the detection of encoding conflicts in STGs, which is intended to work within a synthesis framework based on STG unfoldings. This method proposes algorithms in a very sketched form without an implementation, experiments and analysis of the practicality of the method. Since this method was on the surface promising it has been studied and implemented to determine its practicality. It has been implemented as a software tool ICU using refined algorithms. A necessary condition detects state coding conflicts by using an approximate state covering approach. Being computationally efficient, this algorithm may generate false alarms. Thus a refinement technique is applied based on partial construction of the state space with extra computational cost.

4.1 Approximate state covering approach

The extension of the approach in [47], which identifies state coding conflicts at the level of STG unfolding prefixes is presented. The detection of encoding conflicts is divided into two stages. The first stage exploits the necessary conditions for state coding conflicts such as partial coding information, represented as cover approximations of conditions, and information from the computation of maximal trees. While these

conditions are computationally efficient (quadratic in the size of the unfolding prefix) they may hide non-existent "fake" conflicts. At the second stage sufficient conditions are used, which are based on partial construction of the state space. The state space construction is computationally expensive, though it is done only for small parts of the unfolding prefix identified in the first stage. The work presented in this chapter is based on the results developed in [62, 63].

Before defining the necessary conditions for the detection of encoding conflicts, the notation of cover approximations and slices is presented. The detection of coding conflicts in an STG unfolding prefix requires the concept of Boolean cover approximations for individual instances of places (conditions).

Definition 4.1. Cover approximation

Let $b \in B$ be an arbitrary condition and $e = {}^\bullet b$ be the unique (due to the non-convergent nature of unfolding prefixes with respect to conditions) predecessor event, and let $Code([e])$ be the binary code of the final state of the local configuration of e. The *cover approximation of a condition* b is the cube $Cover^{approx}(b) = c[1]c[2]...c[n]$, where $n = \mid Z \mid$ is the number of signals in the STG and $\forall i : c[i] \in \{0, 1, -\}$, and is computed as follows:

- $c[i] =$ '-', if $\exists z_i^{\pm}$ such that $z_i^{\pm} \parallel b$, and
- $c[i] = Code([e])[i]$, otherwise. $\Diamond$

The cover approximation of a condition b is obtained from the binary code assigned to its preceding event e. Any marking including b can only be reached from $[e]$ by firing events concurrent to b. The literals corresponding to signals whose instances are concurrent to b are replaced by '-'. The approximated covers defined above are cubes derived only by knowing local configurations of events and concurrency relations between conditions and events. All information can be derived in polynomial time from the unfolding.

To represent a mutually connected set of states of the SG the notation slice is introduced.

Definition 4.2. Slice

A *slice* of an STG unfolding is a set of cuts $\mathcal{S} = \langle {}^\bullet\mathcal{S}, \{\mathcal{S}^\bullet\}\rangle$ defined by a cut ${}^\bullet\mathcal{S}$ called min-cut and a set of cuts $\{\mathcal{S}^\bullet\}$ called max-cuts, which satisfy the following conditions:

- min-max correspondence: for every max-cut $\mathcal{S}^\bullet$: ${}^\bullet\mathcal{S} \prec \mathcal{S}^\bullet$ (the min-cut is backward reachable from any max-cut),
- structural conflict of max-cuts: all max-cuts $\mathcal{S}^\bullet$ are in structural conflict,
- containment: if a cut $cut \in \mathcal{S}$, then there is a max-cut such that ${}^\bullet\mathcal{S} \prec cut \prec \mathcal{S}^\bullet$ (every cut of a slice is "squeezed" between a min and some max-cuts), and
- closure: if a cut cut such that ${}^\bullet S \prec cut \prec \mathcal{S}^\bullet \in \{\mathcal{S}^\bullet\}$ then $cut \in \mathcal{S}$ (there are no "gaps" in a cut). ◊

The first two conditions guarantee well-formedness of the slice border, and the remaining two conditions guarantee containment and contiguity of a slice. A special case of a slice is a *marked region* for a condition $b \in B$, which is the set of cuts to which b belongs, denoted as $MR(b)$.

4.1.1 Necessary condition

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

Definition 4.3. Collision
The conditions b_1 and b_2 are said to be in *collision* in an STG unfolding if their cover approximations intersect, i.e. $Cover^{approx}(b_1) \cap Cover^{approx}(b_2) \neq 0$. ◊

There are three sources of collision between a pair of place instances:

1. The marked regions of conditions b_1 and b_2 contain only cuts that are mapped to the same marking of the original STG, i.e. there is no state coding conflict.
2. The exact Boolean covers of the marked regions of a given pair of conditions in collision do not contain the same binary codes but the cover approximations

$Cover^{approx}(b_1)$ and $Cover^{approx}(b_2)$ intersect due to an overestimation. Thus this collision does not correspond to a state coding conflict.

3. In the marked regions of b_1 and b_2 there are two cuts that are mapped to two different markings with the same binary encoding. This results in a state coding conflict and may or may not be a CSC conflict depending on whether these markings enable different sets of output signals.

The idea of detecting state coding conflicts by approximate techniques uses collisions, which can be easily analysed, instead of actual coding conflicts. However, this can be overly conservative because only the case 3 is of interest, while 1 and 2 must be excluded. To make this analysis less conservative, as many *fake* collisions as possible need to be identified.

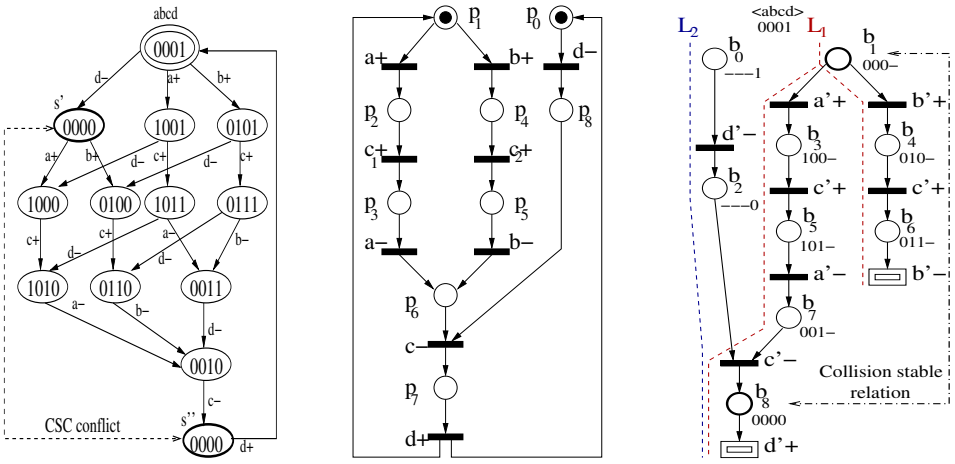
Definition 4.4. Fake collision
A collision between conditions b_1 and b_2 is called *fake* if no cut in the marked region of b_1 is in a state coding conflict with cuts from the marked region of b_2. ◊

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

Definition 4.5. Collision stable conditions
A condition b of an STG unfolding prefix is called *collision stable* if every maximal tree passing through b contains another condition b' which is in collision with b. ◊

In [47] if an original STG has a state coding conflict then its unfolding contains a pair b_1, b_2 of collision stable conditions. This means that if an STG has coding conflicts, then there are at least two conditions in the STG unfolding each of which is in collision with another condition in every maximal tree. This fact is used as a characteristic

property of a state coding conflict in terms of cover approximation. This property is necessary but not sufficient. The STG with no state coding conflicts may have stable collision conditions. This can happen due to an overestimation of cover approximations and reflects the conservative nature of this approach.



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

Figure 4.1: An example for the approximation technique

Consider the STG and its SG represented in Figure 4.1(b) and (a), respectively. The SG shows a CSC conflict between the states s' and s'' (output signal d is not enabled in the first state but is enabled in the second). Figure 4.1(c) shows the unfolding prefix of the STG. The cover approximations are presented next to their conditions. Note that the events of the unfolding prefix are not explicitly labelled, instead their corresponding transitions are used which are referred to by adding one or several primes. Condition b_1 is concurrent to d'^{-} and is ordered with the signals of a, b and c, hence the cover approximation is 000-. Finding collision stable conditions requires the consideration of maximal trees. There are two maximal trees L_1 and L_2 depicted as dashed lines. In the maximal tree L_1 conditions b_1 and b_8 are in collision. The only maximal tree passing through b_1 is L_1 thus b_1 is a collision stable condition. The condition b_8 belongs to two maximal trees. It is in collision with b_1 in L_1 and with b_2 in L_2, hence b_8 is collision stable as well. This pair of collision stable conditions suggests that the STG may not satisfy the CSC property.

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

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

A particular case of an STG in which the initial marking can be transferred without changing the specification is an STG where the initial marking is a home marking. In [33] a procedure of finding a "good" initial marking in an STG was proposed which suggests the use of a *stable basic marking* as the initial one. In an unfolding a stable basic marking is a marking by which, in an unfolding construction, a cut-off is made (the final state of local configuration of a cut-off event). The initial marking of the unfolding can be transferred to this marking in order to reduce its size. A new unfolding is constructed using the new initial marking.

4.1.2 Refinement by partial state construction

A straightforward way to check whether a collision corresponds to a real state coding conflict is to construct all states corresponding to a cube, which is the intersection of the covers of given conditions in collision, in the marked regions of conditions belonging to the collision. If conflicting states are found, the nature of the conflict (i.e. USC/ CSC

conflict) can be determined by comparison of the transitions enabled in these states.

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

The task of detecting coding conflicts from a pair of collision stable conditions b_1 and b_2 consists of the following steps:

1. Let $Cover^{approx}(b_1)$ and $Cover^{approx}(b_2)$ be the covers approximating b_1 and b_2; $c = Cover^{approx}(b_1) \cap Cover^{approx}(b_2)$ and $c \neq 0$.

2. Find the intersection of the ON-set of c with the marked regions of b_1 and b_2 denoted by $ON(b_1)$ and $ON(b_2)$.

3. Construct the binary states of $ON(b_1)$ and $ON(b_2)$.

4. Check for state coding conflicts using the binary states of $ON(b_1)$ and $ON(b_2)$.

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

4.2 Implementation

This section presents the implementation of the approach described in the previous section. The original algorithms [47] have been refined and are described here in detail. The STG unfolding prefix (based on covering all reachable states of the STG) and the cover approximations are obtained by the synthesis tool PUNT [93]. The procedure which identifies collision stable conditions and the procedure which constructs the ON-set are presented first, followed by the overall procedure and further refinements.

4.2.1 Collision stable conditions

The necessary condition requires the refinement of collisions by collision stable conditions. The algorithm proposed in [47] uses information extracted from all possible trees, without enumerating all of them. This algorithm looks for one maximal tree where a condition is free from collisions. If such a tree exists, the given condition is not a collision stable condition. In this algorithm, all conditions which are concurrent with the given condition and all conditions with which the given condition is in collision are removed. Other nodes that cannot be included in any other maximal tree because of the removal of nodes in the previous step are also removed. If the given condition has not been deleted, then it is not a collision stable condition.

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

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

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

Algorithm 1 Identifying collision stable conditions

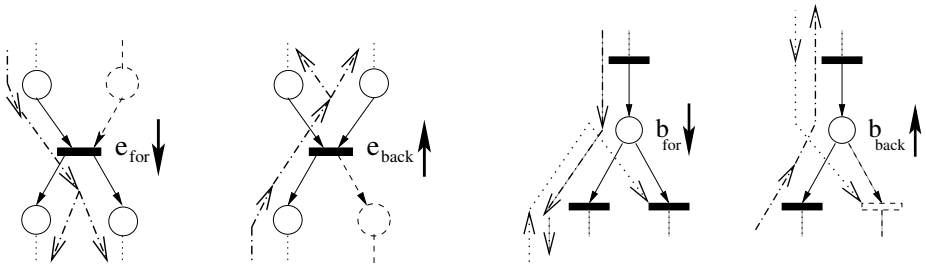
```
isCollisionStableCondition(b_in)
{
   forall (node ∈ E ∪ B )   //reset node flags in unfolding prefix
      setFlag(node,0)
   if (traverseEvent(bwd, e_pre = e : •b_in = {e}) == true)   //start traversing e_pre backward
      return true
   forall (e ∈ b_in•)
   {
      if (traverseEvent(fwd, e) == true)   //start traversing e forward
      {
         setFlag(e,2)
         return true
      }
   }
   return false
}
```

The procedures for traversing an event and a condition are presented in Algorithms 2 and 3. Each procedure requires two input parameters. One parameter is the node to traverse and the other parameter is the traversal direction, forwards or backwards. The procedures return "true" if a collision is found, otherwise "false". The rules for traversing a node in the unfolding prefix are described below.

Traversing an event Several maximal trees can pass through an event e. The given condition b_{in} is collision stable if a collision exists with b_{in} in every tree preceding or succeeding e. In the case of forward traversal in the Algorithm 2 only the successors of e are considered. For example the event e_{for} in Figure 4.2(a) is traversed forwards as follows. The post-set of e_{for} is checked for the existence of collisions in such a way that either $e_{for}{}^{\bullet}$ are in collision with b_{in} or there is a collision with b_{in} in the tree succeeding $e_{for}{}^{\bullet}$. This is done by using node flags and the results which were obtained from procedure *traverseCondition*. If the succeeding nodes of e_{for} have already been traversed they contain information about the existence of collisions. This ensures that these nodes are only traversed once.

The backwards traversal is performed in a similar way. The predecessors of, e.g., the event e_{back} in Figure 4.2(a) are checked for the existence of collisions.

Traversing a condition A condition b traversed by the procedure *traverseCondition* (Algorithm 3) can either be in collision with b_{in} or in collision with conditions in the trees preceding or succeeding b. If b and b_{in} are not in collision the procedure checks



(a) event (b) condition

Figure 4.2: Traversing a node

if collisions exist in the preceding and succeeding conditions. Consider the forward traversal of the condition b_{for} in the Figure 4.2(b). The procedure tries to find a condition in a collision with b_{in} in at least one of the choice branches of b_{for} using the procedure *traverseEvent*. At least one collision is necessary because the succeeding trees of b_{for} belong to a maximal tree passing through b_{for}.

The backwards traversal is illustrated in the right part of Figure 4.2(b). If b_{back} is not in collision with b_{in} then the predecessors of b_{back} are scanned for the existence of collisions. Note that in the unfolding prefix a condition has always only one direct predecessor. If this fails, the forward traversal is applied to the successors of b_{back}, where only the choice branches which have not been visited are considered.

4.2.2 ON-set of a collision relation

The algorithm in [47] calculates the ON-set as follows. It first finds all min-cuts of the given cube c that are reachable from the initial marking without passing through another min-cut. For all these min-cuts it constructs the corresponding ON-slices by calculating the matching set of max-cuts. This is done by restricting the unfolding as follows. The part of the unfolding preceding or in conflict with the given min-cut is irrelevant, since all max-cuts from the matching set succeed their min-cut. The nodes that succeed events that reset the cube c are removed from the unfolding. However, the unfolding may have other min-cuts of c that succeed cuts from the set of the obtained min-cuts, because the cube c can be set and reset several times. To identify those the procedure of finding the ON-set is iterated.

Algorithm 2 Procedure to traverse an event

```
traverseEvent(direction, e)
{
   if (getFlag(e) == 2)   //check if conflict exists
      return true
   setFlag(e,1)   //tag flag of e that it is visited
   if (direction == fwd)
   {   //traverse forwards
      if (e is cut-off event)   //forward traversal possible
         return false
      forall (b ∈ e•)   //check if all succeeding trees contain a collision
      {
         if ((getFlag(b) ≠ 1) and (traverseCondition(fwd, b)))
            setFlag(b, 2)
         else
            return false
      }
      return true
   }
   else
   {   //traverse backwards
      if (e is initial transition)   //backward traversal possible
         return false
      forall (b ∈ •e)   //check if all preceding trees contain a collision
      {
         if ((getFlag(b) ≠ 1) and (traverseCondition(bwd, b)))
            setFlag(b, 2)
         else
            return false
      }
      return true
   }
}
```

The algorithm used here is based on the original [47] as described above. The main difference is that the marked regions of conditions involved in a collision are identified first. Then the restriction is applied only to the slices corresponding to these marked regions, using the cube c similar to the original algorithm. This reduction results in a better performance.

Consider a condition b. From the definition of marked regions follows that all conditions which are concurrent to b, and b itself, belong to the marked region of b. The final state of local configuration of ${}^\bullet b$ (single event) corresponds to the first reachable marking after firing ${}^\bullet b$. Hence this marking is the min-cut of this slice.

The procedure *setONset* is presented in Algorithm 4. Its first step is to find the marked regions of the pair of collision stable conditions involved in a collision. This is done by setting the nodes corresponding to the slice by the function *setMR*. The second step transfers the min-cuts of the marked regions to the cuts where cube c evaluates to 1. These cuts represent the "first" min-cuts of the ON-set. During this process the nodes which do not belong to the ON-set are removed, and the min-cuts are set by the

Algorithm 3 Procedure to traverse a condition

```
traverseCondition(direction, b)
{
   if ((getFlag(b) == 2) or (C(b) ∩ C(b_in) ≠ 0))   //check if conflict exists
      return true
   setFlag(b, 1)   //tag flag of b that it is visited
   if (direction == fwd)   //traverse forwards
   {
      forall (e ∈ b•)   //check if there is a collision in e
      {
         if (getFlag(e) ≠ 1) and (traverseEvent(fwd, e))
         {
            setFlag(e,2)
            return true
         }
      }
   }
   else
   {   //traverse backwards
      e_pre = e : •b = {e}   //in the unfolding a condition has at most one incoming arc
      if (getFlag(e_pre) ≠ 1) and (traverseEvent(bwd, e_pre))   //check for a collision in e_pre
      {
         setFlag(e_pre,2)
         return true
      }   //traverse forwards
      forall (e ∈ b• )   //check if there is a collision in e
      {
         if ((getFlag(e) ≠ 1) and (traverseEvent(fwd, e)))
         {
            setFlag(e,2)
            return true
         }
      }
   return false
   }
}
```

Algorithm 4 Constructing the ON-set of a collision relation

```
setONset(b_1, b_2, Mincuts)
{
   c = Cover^approx(b_1) ∩ Cover^approx(b_2) ≠ 0   //set cube
   forall (node ∈ E ∪ B )   //reset node flags in unfolding prefix
      setFlag(node,0)
   setMR(b_1, b_2)   //set the marked region of b_1 and b_2
   //set first reachable min-cuts of the ON-set to Mincuts
   e_1 = e : •b_1 = {e}   //in the unfolding a condition has at most one incoming arc
   e_2 = e : •b_2 = {e}
   setMincuts(Mark([e_1]), c, Mincuts)
   setMincuts(Mark([e_2]), c, Mincuts)
   //remove nodes which do not belong to the ON-set from the remaining parts of MR(b_1, b_2)
   cuts = Mincuts
   forall (b ∈ cuts)
      reduceSlice(b, c, cut_i, Mincuts)
}
```

procedure *setMincuts*. In the last step, events which force c to reset and their post-sets are removed. In the case where cube c is set again, the min-cuts are determined and the procedure *reduceSlice* is repeated. The construction of the ON-set is schematically illustrated in Figure 4.3.

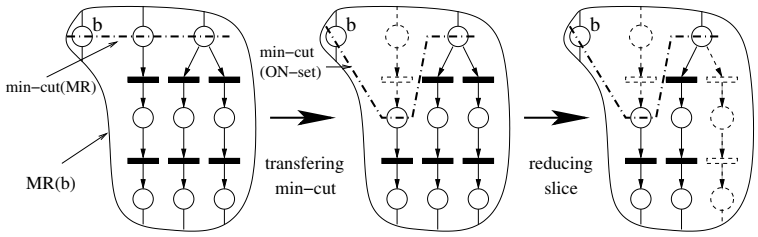


Figure 4.3: Deriving the ON-set

Algorithms 5 and 6 present the procedure *setMincuts* and *reduceSlice* in detail. The procedure *setMincuts* transfers a given min-cut of the marked region *cut* to min-cuts of the ON-set Cut_{ON-set}. This procedure checks if signals differ from the cube c in the current cut *cut*. If this is the case, the events which differ from c are determined and are removed from the marked region using flags. Then the procedure is called recursively after the *nextCut* (the marking after firing the given event from *cut*) has been set by the function *setCut*. In the event of c evaluating to 1, the current cut is included in Cut_{ON-set} (*Mincuts*). In addition if a new max-cut is found the nodes which are in conflict with e are removed. This can be done because from this new max-cut the nodes which are in structural conflict with the removed nodes cannot be reached from the new min-cut. They belong to a different ON-set. Note that this procedure only finds min-cuts of the ON-set which are reached from the min-cut of the marked region.

The procedure *reduceSlice* removes events which force c to reset, and the post-sets of these events. The search of those events is started from the min-cuts of the ON-set Cut_{ON-set} obtained in the previous step and is applied to the remaining parts of marked regions MR. This procedure checks events which are reached from a given cut *cut*, to determine whether they reset c. If such a event is found, it and its post-set is removed from MR and the procedure is called recursively. The case where c is set and reset several times is determined as follows. If the cube c evaluates to 1 in *cut*, and

cut cannot be reached from Cut_{ON-set} due to the removal of events which precede *cut*, then *cut* is a min-cut of the ON-set.

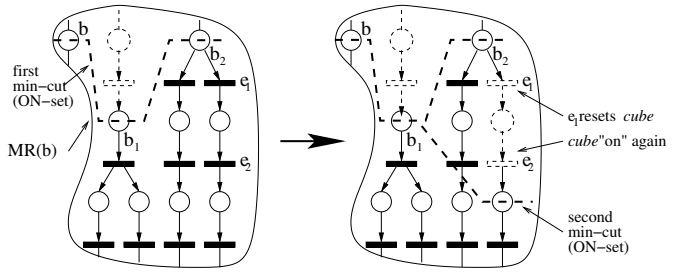


Figure 4.4: Reduction of a marked region by procedure *reduceSlice*

Figure 4.4 illustrates the reduction of a marked region by the procedure *reduceSlice*. The search of events which reset c starts from the min-cut $cut_{min} = \{b, b_1, b_2\}$ of the ON-set corresponding to the marked region $MR(b)$. Suppose event e_1 resets c, resulting in the removal of e_1 and its post-set from the marked region. The detecting process is repeated from the marking $cut_1 = \{b, b_1, e_1^\bullet\}$. Imagine that the event e_2 sets c "on" again. Thus the cut $cut_2 = \{b, b_1, e_2^\bullet\}$ is a min-cut of the ON-set because the successors of e_2 cannot be reached from cut_{min}. The event e_2 is also removed because, until this event has fired, c is not turned "on". The remaining marked region $MR(b)$ is the ON-set of b.

4.2.3 Overall procedure

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

The original algorithm [47] for the necessary condition constructs a $B \times B$ collision matrix, which is then refined by the collision stable conditions. The collisions between conditions which are not collision stable, and any condition which is not concurrent to those conditions, are also removed from the collision matrix. The approach used here

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

```
setMincuts(cut, c, Mincuts)
{
   if (c evaluates to 0 in cut)
   {   //find signals which differ from cube
      forall (b ∈ cut )
      {
         forall (e ∈ b• )
         {
            if ((e differs from c) and (getFlag(e) == 1))
            {
               setFlag(e,0))   //remove e from MR
               if (e is cut-off event)
                  return
               nextCut = setCut(e, cut)
               setMincuts(nextCut, c, Mincuts)
            }
         }
      }
   }
   else if (cut ∉ Mincuts)   //cut is a min-cut of ON-set
   {
      if (e#x : x ∈ E ∪ B)   //nodes in conflict with e
         removeConflictingNodes   //remove conflicting nodes from MR
      add(cut, Mincuts)
   }
}
```

Algorithm 6 Reducing slice

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

Algorithm 7 Detecting state coding conflicts by STG unfolding

```
detectingConflicts(Γ, cover approximations)
{
   forall (b_i ∈ B )
   {
      forall (b_j ∈ B where i > j)
      {
         //necessary condition
         if ((Cover^approx(b_i) ∩ Cover^approx(b_j) ≠ 0) and (isCollisionStableCondition(b_i))
            and (isCollisionStableCondition(b_j)))
         {
            Mincuts = ∅
            setONset(b_i, b_j, Mincuts)   //setting the ON-set
            states = traverseONset(Mincuts)   //constructing the states belonging to the ON-set
            if (states in conflict)   //check of existence of conflicts
               store in data base
            else
               fake conflict
         }
      }
   }
}
```

constructs the collision stable relations "on the fly" in such a way that, if a collision between conditions b_1 and b_2 exists, these conditions must also be collision stable conditions. The information as to whether a condition is collision stable or not is stored in order to prevent repeated checking.

The constructed states which belong to the ON-set are stored as Binary Decision Diagrams (BDDs). The package BuDDy [57] provides main functions for manipulating BDDs. The existence of state coding conflicts is subsequently determined and the output is stored.

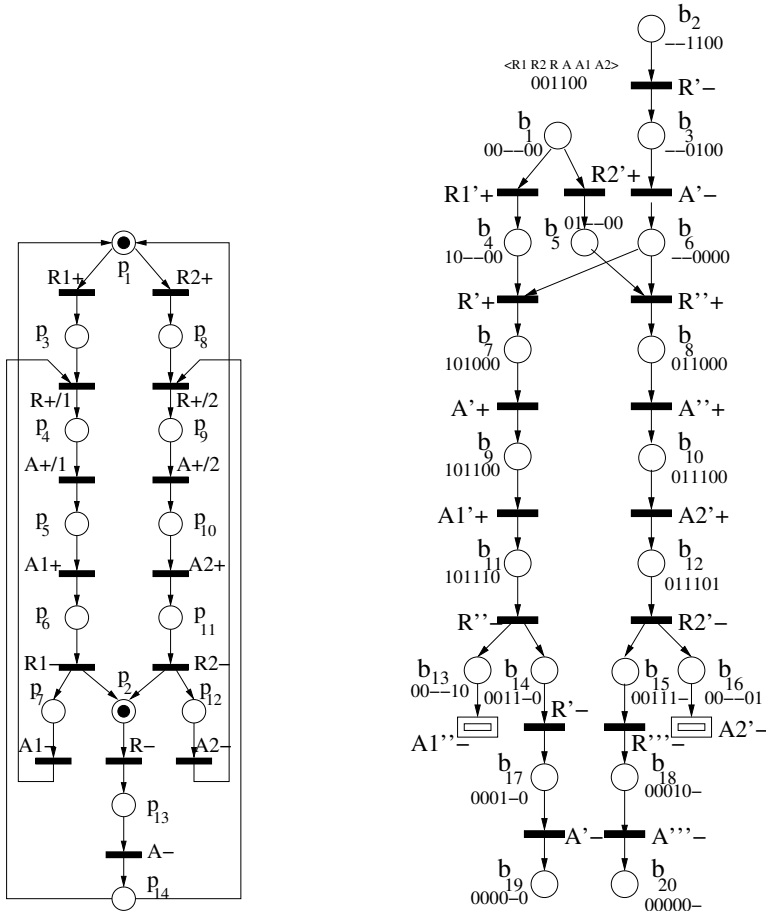
The application of the overall procedure to detect state coding conflicts is illustrated in the example in Figure 4.5. Consider the collision between b_2 and b_9 in the STG unfolding in Figure 4.5(c). This collision must first be refined by the collision stable conditions in order to detect state coding conflicts. The procedure *isCollisionStableCondition* is applied to b_2 and b_9. The condition b_2 is traversed backwards first. Because its predecessor is the initial event, no advance from it is possible, thus b_2 is traversed forward to its direct successor. From this node the node b_3 is visited; then the traversal continued to A'^- and to b_6 because the cover approximations of b_2 and b_3 do not intersect. The condition b_6 is a choice. The traversal is continued in one of the choice branches first, say to R'^+, b_7, A'^+ and b_9. The condition b_9 is in collision with b_2, thus b_2 is a collision stable condition. The condition b_9 is traversed backwards to A'^+, b_7

and R'^{+}. The event R'^{+} has two direct predecessors b_4 and b_6. These conditions, or their preceding conditions, must be in collision with b_9. This is the case because b_4 as well as b_2, which is a predecessor of b_6, are in collision with b_9, thus b_9 is also a collision stable condition. The intersection of the cover approximations of the conditions b_2 and b_9 in the collision stable relation gives a cube $c = --1100 \cap 101100 = 101100$.

The ON-set derived for this collision stable relation is depicted in Figure 4.5(d). The marked region of a condition is identified by finding its concurrent nodes. The marked region $MR(b_2)$ includes the following nodes $\{\{b_1, b_2, b_4b_5\}, \{R1'^{+}, R2'^{+}\}\}$. The min-cut of this slice is the final state of the local configuration of $b_2^{\bullet}$, which is the initial marking $\{p_2p_1\}$ corresponding to $\{b_2b_1\}$. This is the first slice in which the ON-set of the cube c is constructed. In the initial marking, c evaluates to 0. Event $R1'^{+}$ differentiates the binary state of $\{b_2b_1\}$ from c. Hence the min-cut of $MR(b_2)$ is transferred immediately after the firing of $R1'^{+}$ to $\{b_2b_4\}$, which has a binary state of 101100. In this binary state, c evaluates to 1 and therefore is a min-cut of the ON-set. The nodes $R2'^{+}$ and b_5, which are in conflict in this slice with $R1'^{+}$ can be removed, because they belong to another slice. The remaining part of the slice belongs to the ON-set $ON(b_2) = \{b_2b_4\}$ with the corresponding binary state 101100. The marked region of $MR(b_9)$ includes only b_9 (b_9 is not concurrent to other nodes), resulting in the ON-set $ON(b_9) = ON\{b_9\}$ with the binary state 101100. It is easy to conclude that the collision between b_2 and b_9 indeed corresponds to a state coding conflict by checking the binary states corresponding to $ON(b_2)$ and $ON(b_9)$.

4.2.4 Reducing the number of fake conflicts

Reducing the size of the STG unfolding by transferring the initial marking can result in fewer fake collisions between conditions which are mapped onto the same place in the original STG (multiple instances). Consider the example in Figure 4.6. The first unfolding prefix is constructed from the STG using p_1 as initial marking and the second unfolding using p_5, p_6 as initial marking. It can be seen that the second unfolding contains two places (p_5, p_6) in the original STG which have multiple instances in the unfolding, resulting in additional collisions. The condition b_2 (p_5) is in collision either



(a) STG

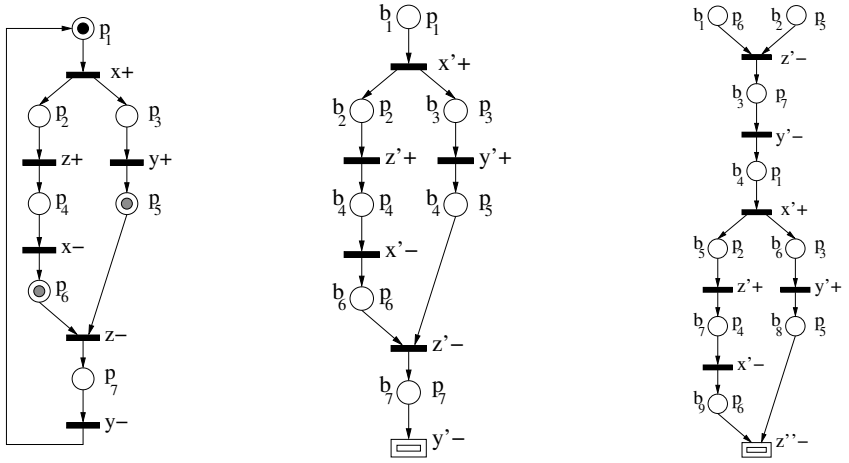
(b) prefix

(c) stable collisions

(d) ON-set

Figure 4.5: Detection of state coding conflicts

with b_8 or b_9 (p_5 or p_6), and b_1 (p_6) is in collision either with b_8 or b_9 (p_5 or p_6). A way to reduce the size of this unfolding is to choose an appropriate initial marking. The final state of local configuration of the cut-off event z''^- is used as a new initial marking. A new unfolding is constructed from the new initial marking $\{p_7\}$ resulting in a smaller unfolding.



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

Figure 4.6: STG unfoldings with different initial markings

Another way of reducing collisions caused by multiple instances, without having to construct a new unfolding, is to consider only the "live" part of the unfolding. Mapping the post-set of the cut-off events into their images results in a "folded" unfolding. The part of the unfolding which does not belong to the folded net can be taken out of the process of detection of state coding conflicts, because this part includes markings which are already contained in the remaining net. For example, in the unfolding in the Figure 4.6(c) the post-set of the cut-off event z''^- can be mapped to b_3 and the nodes b_1, b_2 and z'^- are excluded from the detection process. The marking $\{b_1, b_2\}$ corresponds to the marking $\{p_5, p_6\}$ in the original STG and can be found in the remaining part of the unfolding as marking $\{b_8, b_9\}$.

The process of finding the images of the post-set of cut-off events requires finding the first image of the final state of local configuration of the cut-off events in the unfolding. The predecessors of such an image can only be discarded if its preceding sub-graph is

isomorphic to the one preceding the finite state of local configuration of the given cut-off event.

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

4.3 Experimental results

The approach described above has been applied to a set of benchmarks. A wide spectrum of benchmarks including SMs, MGs, FC nets and arbiters have been chosen to explore the efficiency of this approach. The experimental results of the necessary conditions are presented, and then these are followed by the results of the refinements by partial state construction and the reduction of the size of the unfolding.

The results of the detection of state coding conflicts by the necessary condition are shown in Table 4.1. The number of collision relations is in most cases significantly higher than the the number of collision stable relations. This reflects the conservative estimate of the state space via cover approximations. Benchmarks based on SMs, e.g. *alloc-outbound*, *rpdft* and *seq4*, have the same number of collisions and stable collisions. This happens because their cover approximations contain no "don't cares".

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

In Table 4.2 the size of the traversed state space is illustrated. In the columns labelled "MR" and "ON-set", the traversed state space relating to every collision corresponds to the marked regions and to the ON-sets, respectively. It can be observed from these percentages (to the total reachable state space) that the size of the traversed state space for each possible conflict is smaller than the entire state space. Furthermore, the number of states corresponding to the ON-set for each collision is in many cases

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

Table 4.1: Necessary condition

smaller than the number of states corresponding to the marked regions. This reflects the reduction of the marked regions by cubes.

In the Table 4.3 the relation of collisions and state coding conflicts is presented. It can be seen that the number of fake conflicts is high in several cases. Benchmarks based on, or dominated by, SM have a small number of fake conflicts, whereas highly concurrent nets have a large number of fake conflicts. This happens because cover approximations contain many "don't cares", resulting in false alarms.

In Table 4.4 the relation between collisions and the size of the unfolding was examined for those benchmarks where it is possible to minimise the size of the unfolding. This table shows the reduction of collision relations obtained by transferring the initial marking (transf.) and by using the folding technique (fold.). It can be observed that, using both methods, the majority of collision stable relations are reduced in these benchmarks, but the reduction obtained by transferring the initial marking is greater.

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

Table 4.2: Size of the traversed state space

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

Table 4.3: Number of fake conflicts

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

Table 4.4: Reduction of collisions

4.4 Conclusions

The approach to detect state coding conflicts by STG unfolding based on [47] has been implemented as a software tool [63]. This approach uses cover approximation of conditions and the information of maximal trees to estimate the state space, resulting in a necessary condition for state coding conflict to exists. Whilst this condition is computationally efficient it may hide the so-called "fake" conflicts. Thus, a refinement technique is applied to resolve such situations at the expense of extra computational costs. This technique limits the search to those parts of the unfolding that may potentially exhibit a fake conflict. Those parts need explicit state traversing, which may be exponentially expensive.

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

examined and this shows a high reduction of collisions in several benchmarks.

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

In general, due to the over-approximation this approach is inefficient in detecting CSC conflict in STGs, where concurrency dominates. Since STGs usually exhibit a lot of concurrency this approach is impractical. However, during this time an unfolding based approach in [40] had been proposed. It has proved to be efficient in identifying encoding conflicts in STG unfolding prefixes and therefore is used for the visualisation and resolution of encoding conflicts in the next chapter.

Chapter 5

Visualisation and resolution of encoding conflicts

The synthesis of asynchronous circuits from STGs involves the resolution of state encoding conflicts by means of refining the STG specification. The refinement process is generally done automatically using heuristics. It can often produce sub-optimal solutions or sometimes fail to solve the problem. Thus a manual intervention by the designer may be required. According to a practising designer [87], 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, e.g. by choosing where to insert new signals in the specification.

A visualisation method is presented here, which enables the designer to comprehend the cause of encoding conflicts. It works on the level of the complete and finite prefix of the STG unfolding and avoids the explicit enumeration of encoding conflicts. The encoding conflicts are visualised as *cores*, i.e. sets of transitions "causing" one or more of conflicts.

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. 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 trans-

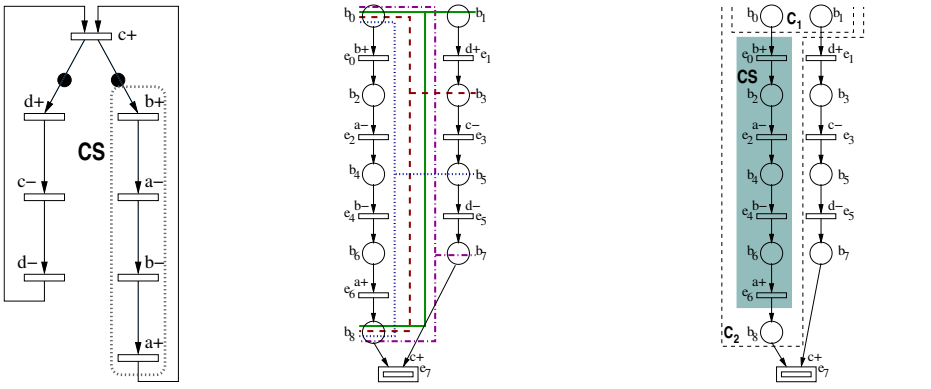
formation. The method can also work in a completely automatic or semi-automatic manner, making it possible for the designer to see what is happening and intervene at any stage during the encoding conflict resolution process. The proposed method is implemented as a tool ConfRes for the resolution of CSC conflicts. The work presented in this chapter has been published in [43, 59, 60, 61, 64, 65, 66].

5.1 Compact representation of conflicts

In this section the visualisation of encoding conflicts in asynchronous circuits design is presented, which is later used to resolve state coding conflicts. The first step in the resolution process is to identify the cause of encoding conflicts. Figure 5.1(a) shows an STG with a CSC problem. The representation of all possible conflicts is not efficient, as illustrated the unfolding prefix shown in the Figure 5.1(b), where the conflict pairs $\langle\{b_0, b_1\}, \{b_8, b_1\}\rangle$, $\langle\{b_0, b_3\}, \{b_8, b_3\}\rangle$, $\langle\{b_0, b_5\}, \{b_8, b_5\}\rangle$ and $\langle\{b_0, b_7\}, \{b_8, b_7\}\rangle$, which correspond to pairs of states in CSC conflict, are shown. It can be seen that even a small number of conflicts are difficult to depict. Visualising only the essential parts of the unfolding involved in conflicts (Figure 5.1(c)) offers a more elegant solution, which avoids the explicit representation as pairs of states (cuts) of conflicts. The visualisation is based on conflict sets, which are represented at the level of STG unfolding prefixes. Generally speaking, a conflict set is a set of transitions contributing to a conflict. In the example in Figure 5.1(c) the conflict set $\mathcal{CS} = \{b+, a-, b-, a+\}$ contains both the falling and rising transitions, making the states before and after executing $\mathcal{CS}$ identical. This type of conflict set, which corresponds to CSC conflicts, is also known as a *complementary set* in [12].

Since every element in an STG unfolding prefix is an instance of an element in the original STG, the conflict sets can be easily mapped from STGs to their unfolding prefixes and vice versa. The conflict set $\{e_0, e_2, e_4, e_6\}$ can be mapped from the unfolding prefix (Figure 5.1(c)) to the corresponding transitions in $\mathcal{CS}$.

Encoding conflicts can be classified in p-normalcy, n-normalcy, USC, CSC or CSC^z_X conflicts. Let two states be in an $\mathcal{X}$ *conflict*, where $\mathcal{X}$ is either p-normalcy, n-normalcy,



(a) STG

(b) an attempt to depict all CSC-conflicts

(c) prefix with CSC-conflict pair

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

Figure 5.1: Visualisation of CSC conflicts

USC, CSC or CSC^z_X. 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 Figure 5.1(c) the CSC conflict pair $\langle C_1, C_2\rangle$ leads to a CSC conflict.

The set of all $\mathcal{X}$ conflict pairs may be quite large. In the case involving the following "propagation" effect: if C_1 and C_2 can be expanded by the same event e, then $\langle C_1 \cup \{e\}, C_2 \cup \{e\}\rangle$ is also an $\mathcal{X}$ conflict pair unless it does not correspond to an $\mathcal{X}$ conflict. For example, if $\mathcal{X}$ is a CSC conflict then the expanded configurations correspond to a conflict pair unless these two configurations enable the same set of output signals. In Figure 5.1(c)$\langle\{\emptyset\}, \{e_0, e_2, e_4, e_6\}\rangle$ is a CSC conflict pair, and adding, e.g., the event e_1 to both these configurations leads to a new CSC conflict pair $\langle\{e_1\}, \{e_0, e_1, e_2, e_4, e_6\}\rangle$. Therefore, it is desirable to reduce the number of pairs which need to be 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 [37] and [41], where the fact that $C = C_1 \cap C_2$ is a configuration is also shown.

Proposition 5.1. *Let $\langle C_1, C_2\rangle$ be a concurrent $\mathcal{X}$ conflict pair. Then $C = C_1 \cap C_2$ is such that either $\langle C, C_1\rangle$ or $\langle C, C_2\rangle$ is a $\mathcal{X}$ 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 $e' \# e''$ (i.e. configurations C_1 and C_2 are in structural conflict).

An example of a type I CSC conflict is illustrated in Figure 5.1(c) and an example of a type II p-normalcy conflict for signal c is illustrated in Figure 5.2(a). Note that for simplicity the implicit conditions (place instances) of the unfolding prefix are not represented here.

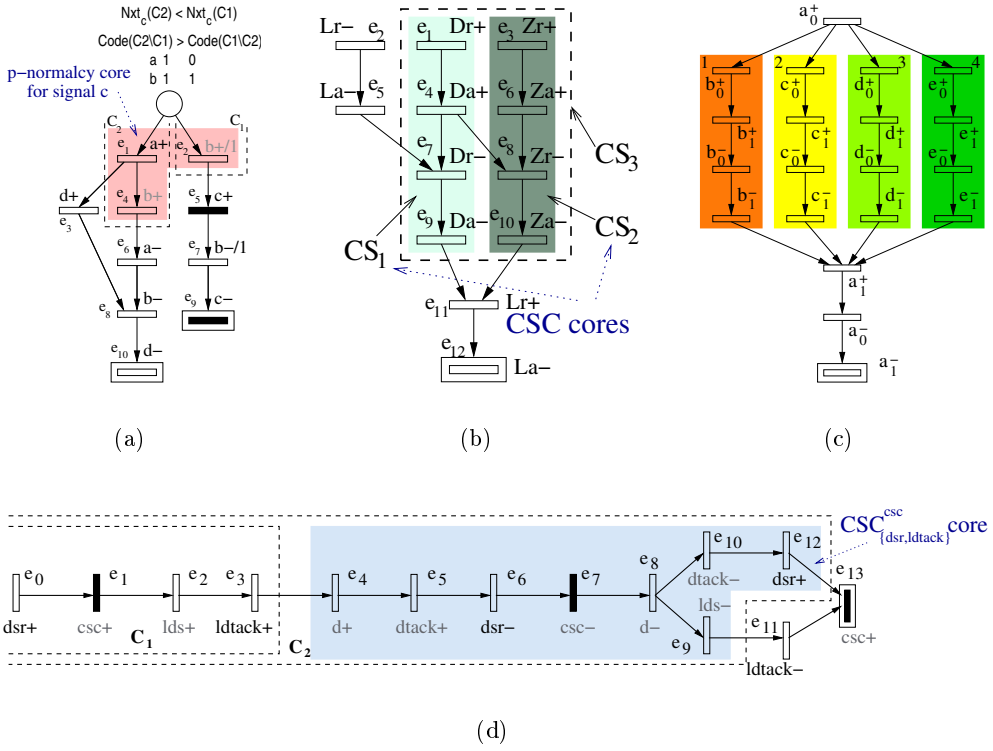


Figure 5.2: Visualisation examples of $\mathcal{X}$ cores

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

Definition 5.1. $\mathcal{X}$ core

Let $\langle C_1, C_2 \rangle$ be a $\mathcal{X}$ 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 a $\mathcal{X}$ *core* if it cannot be

represented as the union of several disjoint $\mathcal{X}$ conflict sets. A $\mathcal{X}$ conflict set is of type I/II if the corresponding $\mathcal{X}$ conflict pair is of type I/II, respectively. $\Diamond$

For example, the CSC conflict sets in Figure 5.2(b) $\mathcal{CS}_1$ 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 Figure 5.1(c) is $\{e_0, e_2, e_4, e_6\}$. Note that for a $\mathcal{X}$ 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 Figure 5.2(a) corresponds 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 a $\mathcal{X}$ 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. The encoding of such a partition is $Code(C_1 \setminus C_2) = Code(C_1) - Code(C_2)$, where $C_2 \subset C_1$. If C_1 and C_2 correspond to type II conflict pairs the encoding of such partitions is $Code(C_1 \setminus C_2) = Code(C_1) - Code(C_1 \cap C_2)$.

An important property of $\mathcal{X}$ conflicts is described below for the corresponding $\mathcal{X}$ conflicts:

Normalcy conflict: $Code(C_1 \setminus C_2)$ R $Code(C_2 \setminus C_1)$ and if $\mathcal{CS}$ is of type I then $Code(\mathcal{CS})$ R $\mathbf{0}$, where R $= \{\leq, \geq\}$. An example of a type II p-normalcy core for signal c is presented in Figure 5.2(a). Note that if the encodings are equal then $\mathcal{CS}$ corresponds to a CSC conflict. To resolve conflicts caused by $\mathcal{CS}$ the encodings must be made unordered. This can be done by extending the encoding in $\mathcal{CS}$ with an additional bit (e.g. in form of an auxiliary signal transition with an appropriate polarity), which would make the encodings not comparable.

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. Note that a USC/CSC conflict set $\mathcal{CS}$ is also known as a *complementary set* [12].

$\mathbf{CSC}^{z}_{X}$ conflict: $Code_x(C_1 \setminus C_2) = Code_x(C_2 \setminus C_1)$ for each $x \in X$, and if $\mathcal{CS}$ is of type I then for each $x \in X$, the difference between the numbers of x^+- and x^--labelled events in $\mathcal{CS}$ is zero. An example of a type I $\mathrm{CSC}^{csc}_{\{dsr,ldtack\}}$ is shown in Figure 5.2(d), where $Code_{\{dsr,ldtack\}}(\mathcal{CS})$ is 1 for both *dsr* and *ldtack* (in the core *dsr* changes twice and *ldtack* does not change). Thus, $\mathcal{CS}$ can be eliminated by introducing an additional bit into $\mathcal{CS}$ by, for example, extending the support X with a signal whose transition is in $\mathcal{CS}$, e.g. in Figure 5.2(d) the support can be extended by *csc*.

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 Figure 5.2(c) has four concurrent branches with a CSC conflict 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 Figure 5.2(c). Note that there are 15 CSC conflict sets, which can be obtained by uniting these cores.

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 easily be mapped from the prefix to the STG. It can be seen that $\mathcal{X}$ cores are crucial for the resolution of $\mathcal{X}$ conflicts. Eliminating these would result in also eliminating the $\mathcal{X}$ conflict sets which are completely composed of $\mathcal{X}$ cores. For example, the unfolding prefix in Figure 5.2(b) has several CSC conflicts caused by three CSC conflict sets $\mathcal{CS}_1$, $\mathcal{CS}_2$ and $\mathcal{CS}_3 = \mathcal{CS}_1 \cup \mathcal{CS}_2$. Eliminating $\mathcal{CS}_1$ and $\mathcal{CS}_2$ would result in the elimination of $\mathcal{CS}_3$.

The $\mathcal{X}$ conflicts can be visualised by $\mathcal{X}$ cores with some additional information depending on $\mathcal{X}$ as follows:

- Normalcy conflicts:

  This type of conflict is visualised by normalcy cores showing the signal involved. However, it is possible to use heuristics based on triggers to determine whether a non-input signal is normal or not. Thus, showing the triggers involved would be

another type of visualisation. The visualisation of normalcy is described in detail below.

- USC/CSC conflicts:

  These types of conflicts are visualised by USC/CSC cores.

- CSC_X^z conflicts:

  To visualise this type of conflict, CSC_X^z cores are used, where the events in the prefix which do not correspond to the signals in X are faded out. Additionally, the events corresponding to z are drawn as solid bars. An example is presented in Figure 5.2(d).

Normalcy conflict visualisation In [39, 40] a hypothesis is made when checking for normalcy for each output and internal signal. The hypothesis is based on the triggers of the events labelled by such a signal. If a z-labelled event has triggers with the same (opposite) sign as the event itself then a hypothesis is made that z is p-normal (n-normal).

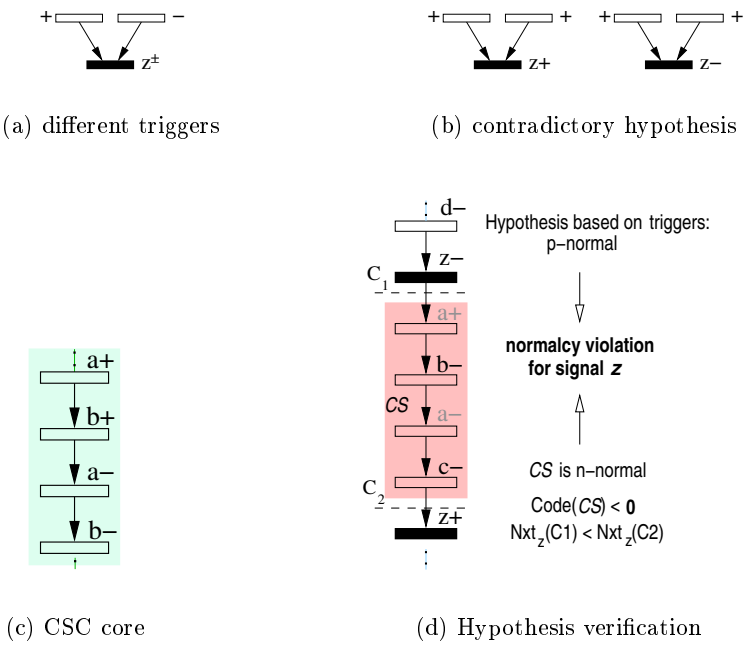
With the exception of certain special cases (e.g. when some signal is set at the beginning and can never be reset), for each signal $z \in Z_O$ there is an event labelled by $z^{\pm}$ containing a set of triggers for which at least one hypothesis can be made about the normalcy type. However, it is sometimes 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, the normalcy violation can be caused by the following factors:

1. Different triggers (Figure 5.3(a)):

   If a transition instant has triggers with different signs, then the corresponding signal is neither p-normal nor n-normal.

2. Contradictory hypothesis (Figure 5.3(b)):

   The transitions of a signal have contradictory triggers, so the signal is neither n-normal nor p-normal. This is the case if, e.g. a signal transition has triggers with the same sign (which means that the signal cannot be n-normal) whilst another



(a) different triggers

(b) contradictory hypothesis

(c) CSC core

(d) Hypothesis verification

Figure 5.3: Visualisation of normalcy violation

instance of this signal has triggers with the opposite sign (i.e. the signal cannot be p-normal) and the opposite transition of this signal has triggers with the opposite sign (i.e. the signal cannot be p-normal).

3. CSC conflict (Figure 5.3(c)):

   There is a CSC conflict.

4. Hypothesis verification (Figure 5.3(d)):

   If none of the above holds a hypothesis is made about the normalcy type of each signal, based on the signs of the triggers, which has to be disproved by definition.

These factors can be visualised by either highlighting events, CSC cores, or normalcy cores. The violations caused by different triggers and contradictory hypothesis, respectively, are visualised by highlighting the events whose triggers violate normalcy w.r.t to their corresponding signals (see Figure 5.3(a) and (b)). A CSC violation is visualised by a CSC core as shown in Figure 5.3(c). The hypothesis verification is visualised by a normalcy core, where the transition instances whose binary value do not change in the core are faded out. Additionally, events corresponding to signals for which the normalcy core is built are drawn as solid bars. For example, in the p-normal core in

Figure 5.3(d) the binary value of a is the same before and after the core, and thus the remaining transitions in the core indicate that the encoding is smaller after the core but $Nxt_z(C_1) < Nxt_z(C_2)$. This core disproves the hypothesis that z is p-normal, which is based on the signs of the triggers of z.

5.2 Net transformation for resolution of conflicts

A notion of validity that is used to justify STG transformations to solve encoding conflicts is presented. However, this notion is much more general and is also of independent interest: it is formulated for labelled Petri nets (of which STGs being a special case) and transformations preserving the alphabet of the system.

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 [85]. 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 [90] can increase the concurrency of inputs, which in turn *extends* the language). For the sake of generality, arbitrary transformations (not necessarily concurrency reductions or signal insertions) are discussed.

Intuitively, there are four aspects to a valid transformation:

I/O interface preservation The transformation must preserve the interface between the circuit and the environment. In particular, no input transition can be "delayed" by newly inserted signals or ordering constraints.

Conformation Bounds the behaviour from above, i.e. requires that the transformation introduces no "wrong" behaviour. Note that certain extensions of behaviour are valid, e.g. two inputs in sequence may be accepted concurrently [22, 90], extending the language.

Liveness Bounds the behaviour from below, i.e. requires that no "interesting" behaviour is completely eliminated by the transformation.

Technical restrictions It might happen that a valid transformation is still unacceptable because the STG becomes unimplementable or because of some other technical restriction. For example, one usually requires the transformation to preserve the speed-independence of the STG [15, 14].

In this section a bisimulation-style validity notion is introduced, which takes the liveness into account and allows to justify both concurrency reduction for outputs and increase of concurrency for inputs, as well as signal insertion. It is believed that it better reflects the intuition of what a valid transformation is.

5.2.1 Validity

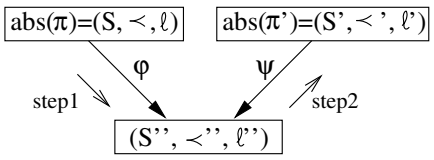
For the sake of generality, arbitrary LPNs (STGs being a special kind of them) are discussed. It is assumed that the transformation does not change the inputs and outputs of the system, and the original and transformed LPNs are denoted by Υ and Υ', respectively. Since one of the transformations discussed is concurrency reduction, it is convenient to use a partial order rather than interleaving semantics, and the discussion will be based on processes of LPNs.

Given processes π of Υ and π' of Υ', a relation between their abstractions, abs(π) and abs(π') (defined in 2.3.3), is defined which holds iff in π' the *inputs are no less concurrent* and the *outputs are no more concurrent* than in π. That is, the transformation is allowed, on one hand, to relax the assumptions about the order in which the environment will produce input signals, and, on the other hand, to restrict the order in which outputs are produced. Thus the modified LPN will not produce new failures and will not cause new failures in the environment.

The relation definition assumes the *weak fairness*, i.e. that a transition cannot remain enabled forever: it must either fire or be disabled by another transition firing. In particular, this guarantees that the inputs eventually arrive, and thus the concurrency reduction $i \dashrightarrow o$ cannot be declared invalid just because the input i fails to arrive and

so the output o is never produced.

Intuitively, abs(π) and abs(π') are bound by this relation iff abs(π) can be transformed into abs(π') in two steps (see the picture below): (i) the ordering constraints for inputs are relaxed (yielding a new order $\prec''$, which is a relaxation of $\prec$); (ii) new ordering constraints for outputs are added, yielding abs(π') (thus, $\prec''$ is also a relaxation of $\prec'$).

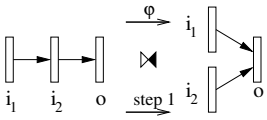


Let $(S, \prec)$ be a partially ordered set and $s \in S$. An $s' \in S$ is a *direct predecessor* of s if $s' \prec s$ and there is no $s'' \in S$ such that $s' \prec s'' \prec s$. The set of direct predecessors of an $s \in S$ is denoted by $DP_{\prec}(s)$.

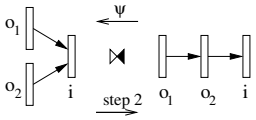
Definition 5.2. Relation between processes

Let π and π' be processes of Υ and Υ', respectively, abs(π) $= (S, \prec, \ell)$ and abs(π') $= (S', \prec', \ell')$. A relation is defined abs(π) $\bowtie$ abs(π') if there exist a labelled partially ordered set $(S'', \prec'', \ell'')$ and one-to-one mappings φ : abs(π) $\rightarrow (S'', \prec'', \ell'')$ and ψ : abs(π') $\rightarrow (S'', \prec'', \ell'')$ preserving the labels and such that:

- $\prec''= \varphi(\prec) \cap \psi(\prec')$ ($\prec''$ is a relaxation of $\prec$ and $\prec'$) ;
- if e is an output event and $f \in DP_{\prec}(e)$ then $\varphi(f) \in DP_{\prec''}(\varphi(e))$ (in step 1, existing *direct* ordering constraints for outputs are preserved, and existing indirect ones can become direct, e.g. as in the picture below);



- if e' is an input event and $f' \in DP_{\prec'}(e')$ then $\psi(f') \in DP_{\prec''\psi}(e')$ (in step 2, no new *direct* ordering constraints for inputs can appear, e.g. as in the picture below).



◊

Note that $\bowtie\!\!\blacktriangleleft$ is an order (if order-isomorphic partially ordered sets are not distinguished). In the sequel, slightly abusing the notation, $\pi \bowtie\!\!\blacktriangleleft \pi'$ is written instead of $\text{abs}(\pi) \bowtie\!\!\blacktriangleleft \text{abs}(\pi')$.

Definition 5.3. Validity

Υ' is a *valid realisation* of Υ, denoted $\Upsilon \multimap \Upsilon'$, if there is a relation $\propto$ between the finite processes of Υ and Υ' such that $\pi_\emptyset \propto \pi'_\emptyset$ (where $\pi_\emptyset$ and $\pi'_\emptyset$ are the empty processes of Υ and Υ', respectively), and for all finite processes π and π' such that $\pi \propto \pi'$:

- $\pi \bowtie\!\!\blacktriangleleft \pi'$
- For all maximal processes $\Pi' \sqsupseteq \pi'$, and for all finite processes $\widehat{\pi}' \sqsupseteq \pi'$ such that $\widehat{\pi}' \sqsubseteq \Pi'$, there exist finite processes $\widetilde{\pi}' \sqsupseteq \widehat{\pi}'$ and $\widetilde{\pi} \sqsupseteq \pi$ such that $\widetilde{\pi}' \sqsubseteq \Pi'$ and $\widetilde{\pi} \propto \widetilde{\pi}'$.
- For all maximal processes $\Pi \sqsupseteq \pi$, and for all finite processes $\widehat{\pi} \sqsupseteq \pi$ such that $\widehat{\pi} \sqsubseteq \Pi$, there exist finite processes $\widetilde{\pi} \sqsupseteq \widehat{\pi}$ and $\widetilde{\pi}' \sqsupseteq \pi'$ such that $\widetilde{\pi} \sqsubseteq \Pi$ and $\widetilde{\pi} \propto \widetilde{\pi}'$.

  ◊

Intuitively, every activity of Υ is *eventually* performed by Υ' (up to the $\bowtie\!\!\blacktriangleleft$ relation) and cannot be pre-emptied due to choices, and vice versa, i.e. Υ' and Υ simulate each other with a finite delay. Note that $\multimap$ is a pre-order, i.e. a sequence of two valid transformations is a valid transformation.

5.2.2 Concurrency reduction

A general definition of concurrency reduction is given, which introduces a causal constraint as illustrated in Figure 5.4.

Definition 5.4. Concurrency reduction

Given an LPN $\Upsilon = (\Sigma, \mathcal{I}, \mathcal{O}, \ell)$ where $\Sigma = (P, T, F, M_0)$, a non-empty set of transitions

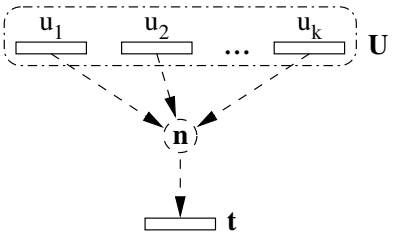


Figure 5.4: Concurrency reduction $U \overset{n}{\dashrightarrow} t$

$U \subset T$, a transition $t \in T \setminus U$ and an $n \in \mathbb{N}$, the transformation $U \overset{n}{\dashrightarrow} t$, yielding an LPN $\Upsilon' = (\Sigma', \mathcal{I}, \mathcal{O}, \ell)$ with $\Sigma' = (P', T, F', M_0')$ is defined as follows:

- $P' = P \cup \{p\}$, where $p \notin P \cup T$ is a new place;
- $F' = F \cup \{(u, p) | u \in U\} \cup \{(p, t)\}$;
- For all places $q \in P$, $M_0'(q) = M_0(q)$, and $M_0'(p) = n$.

The notation $U \dashrightarrow t$ is used instead of $U \overset{0}{\dashrightarrow} t$ and the notation $u \overset{n}{\dashrightarrow} t$ is used instead of $\{u\} \overset{n}{\dashrightarrow} t$. ◊

Note that concurrency reduction cannot add new behaviour to the system — it can only restrict it. Furthermore, one can easily show that if a concurrency reduction $U \overset{n}{\dashrightarrow} t$ such that $\ell(t) \notin \mathcal{I}$ is applied to an input-proper LPN Υ (defined in Def. 2.39), then the resulting LPN Υ' is also input proper.

The validity condition for general LPNs was proposed in [43], where it was simplified for non-auto-concurrent LPNs. The simplified version is presented below.

Proposition 5.2. *Validity condition for a concurrency reduction on non-auto-concurrent nets. Let $U \overset{n}{\dashrightarrow} t$ be a concurrency reduction transforming an input-proper LPN $\Upsilon = (\Sigma, \mathcal{I}, \mathcal{O}, \ell)$ into $\Upsilon' = (\Sigma', \mathcal{I}, \mathcal{O}, \ell)$, such that $\ell(t) \notin \mathcal{I}$, and t is non-auto-concurrent and such that for each t-labelled event e and for each maximal process $\Pi \supseteq [e]$ of Υ there is a finite set $E_U \subseteq \Pi$ of events with labels in U concurrent to e such that $n + \#_U[e] + |E_U| \geq \#_t[e]$. Then $\Upsilon \multimap \Upsilon'$.* □

Remark 5.1. This proposition requires the non-auto-concurrency of a particular transition rather than the absence of two transitions with the same label which can be

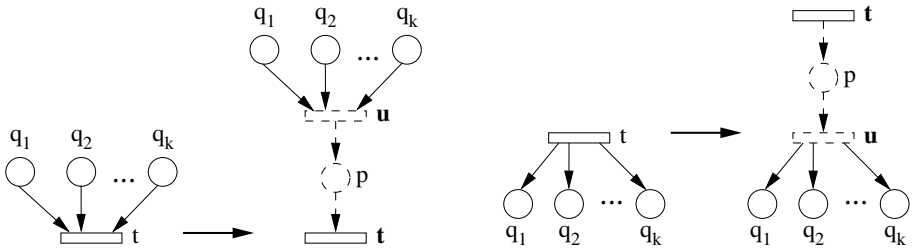
executed concurrently. That is, the non-auto-concurrency is required not on the level of LPN, but rather on the level of the underlying Petri net. In particular, the non-auto-concurrency is guaranteed for safe Petri nets.

5.2.3 Signal insertion

This section presents validity conditions for signal insertions. First, sequential insertions by means of signal transition splitting are shown followed by concurrent insertion of transitions.

5.2.3.1 Transition splitting

A silent τ transition can be inserted sequentially to an existing transition by splitting it and inserting τ either before or after it.



(a) insert before t: $\rightarrow\!\!\wr t$ (a) insert after t: $t\wr\!\!\rightarrow$

Figure 5.5: Transition splitting

Definition 5.5. Transition splitting
Given an LPN $\Upsilon = (\Sigma, \mathcal{I}, \mathcal{O}, \ell)$ where $\Sigma = (P, T, F, M_0)$, a transition $t \in T$, the transformation either $\rightarrow\!\!\wr t$ or $t\wr\!\!\rightarrow$, yielding an LPN $\Upsilon' = (\Sigma', \mathcal{I}, \mathcal{O}, \ell)$ with $\Sigma' = (P', T', F', M_0)$ is defined as follows:

- $T' = T \cup \{u\}$, where $u \notin P \cup T$ is a new transition;
- $P' = P \cup \{p\}$, where $p \notin P \cup T$ is a new place;
- if $u \rightarrow\!\!\wr t$: $F' = F \cup (\{(q,u) | q \in {}^\bullet t\} \cup \{(u,p)\} \cup \{(p,t)\}) \backslash \{(q,t) | q \in {}^\bullet t\}$;
- if $t\wr\!\!\rightarrow u$: $F' = F \cup (\{(t,p)\} \cup \{(p,u)\} \cup \{(q,t) | q \in t^\bullet\}) \backslash \{(t,q) | q \in t^\bullet\}$.

The transformation is called *input-proper* if $\ell(t) \notin \mathcal{I}$ for $\rightarrow\!\wr t$ and if $\forall t' \in (u^\bullet)^\bullet : \ell(t') \notin \mathcal{I}$ for $t\wr\!\rightarrow$, where u is a new silent transition such that $\ell(u) \notin \mathcal{I} \cup \mathcal{O}$. $\Diamond$

Note that the transition splitting $\rightarrow\!\wr t$ delays t by the added silent transition u and that $u^\bullet$ is always consumed by t. In the case of $t\wr\!\rightarrow$ $t^\bullet$ is produced by u and u delays the direct succeeding transitions of t. One can easily show that if an input proper transition splitting ($\rightarrow\!\wr t$ or $t\wr\!\rightarrow$) is applied to an input-proper LPN Υ, then the resulting LPN Υ' is also input proper.

Proposition 5.3. *Validity condition for transition splitting* $\rightarrow\!\wr t$
Let $\rightarrow\!\wr t$ *be an input-proper transition splitting transforming an input-proper LPN* $\Upsilon = (\Sigma, \mathcal{I}, \mathcal{O}, \ell)$ *into* $\Upsilon' = (\Sigma', \mathcal{I}, \mathcal{O}, \ell)$. *Then* $\Upsilon \multimap \Upsilon'$.

Proof. The relation $\propto$ between the finite processes of Υ and Υ' is defined as follows: $\pi \propto \pi'$ iff there exists a one-to-one mapping ξ between the nodes of π, non-u-labelled nodes of π' and non-p-labelled nodes of π', where u is the transition and p is the place added by the transition splitting (note that π contain neither u-labelled events nor p-labelled conditions), such that for every condition c and event e of π:

- $h(c) = h(\xi(c))$ and $h(e) = h(\xi(e))$ (i.e. ξ preserves the labels of events and conditions).
- $e \in {}^\bullet c$ iff $\xi(e) \in {}^\bullet\xi(c)$ and $c \in e^\bullet$ iff $\xi(c) \in \xi(e)^\bullet$ (i.e. ξ preserves the pre-sets of the conditions and the post-sets of events; note that post-sets of conditions and pre-sets of events in π' may be replaced by an u-labelled event and a p-labelled condition, respectively).
- no maximal τ-events in π'

Intuitively, $\pi \propto \pi'$ iff π' can be obtained from π by adding a few u-labelled events, p-labelled conditions and the corresponding arcs. Note that according to this definition, $\pi_\emptyset \propto \pi'_\emptyset$. Several relevant properties of this relation are proven below.

Claim 1: if $\pi \propto \pi'$ then $\pi \bowtie \pi'$.

The labelled partially ordered set $(S'', \prec'', \ell'')$ in Definition 5.2 is chosen to be

abs(π), φ to be the identity mapping, and ψ to be ξ^{-1} restricted to the events labelled by non-internal transitions. Both φ and ψ preserve the labels and $\prec''=\varphi(\prec)\cap\psi(\prec')$ (the latter holds because $\prec''=\prec$ by definition, and all the arcs in π are also present in π', i.e. $\prec$ is a relaxation of $\prec'$).

Since $\prec''=\prec$, the only property to be proven is that if e' is an input event and $f' \in DP_{\prec'}(e')$ then $\psi(f') \in DP_{\prec''}\psi(e')$. It holds because $\ell(t) \notin \mathcal{I}$ and Υ is input proper, and thus no input event is delayed (either directly or via a chain of τ-labelled events) by the transformation.

Claim 2: if $\pi \propto \pi'$ then $\xi(Cut(\pi)) = Cut(\pi')$.

Cuts are comprised of maximal conditions. Since no maximal τ-events are in π', p-labeled conditions are not in the $Cut(\pi')$ and $c^\bullet = \emptyset$ iff $\xi(c^\bullet) = \emptyset$, hence there exists an one-to-one mapping between the maximal conditions of π and π'. Thus the claim holds.

Claim 3: if $\pi \propto \pi'$ and π' can be extended by a finite set of events E' then there exist finite sets $\hat{E}' \supseteq E'$ and E, such that $\pi \oplus E \propto \pi' \oplus \hat{E}'$.

If E' is a singleton $\{e'\}$ and e' is not a u-labelled event then ${}^\bullet e' \subseteq Cut(\pi')$ and the result follows from Claim 2. If e' is a u-labelled event then by Claim 2 a t-labelled event e with the pre-set $\xi^{-1}({}^\bullet e')$ can be added to π', moreover, $\pi' \oplus \{e'\}$ can be extended by a t-labelled event e'' with the pre-set $e'^\bullet$ and $\pi \oplus \{e\} \propto \pi' \oplus \{e', e''\}$. Since any finite extension of π can be obtained by a finite sequence of single-event extensions, the claim follows by induction.

Claim 4: if $\pi \propto \pi'$, π can be extended by a finite set of events E then π' can be extended by a finite set of events E' such that $\pi \oplus E \propto \pi' \oplus \{E'\}$.

If E is a singleton $\{e\}$ and e is not a t-labelled event then ${}^\bullet e \subseteq Cut(\pi)$ and the result follows from Claim 2. If e is a t-labelled event then by Claim 2 π' can be extended by an u-labelled event e' with the pre-set $\xi({}^\bullet e)$, and then a t-labelled event with the pre-set $e'^\bullet$ can be added. Since any finite extension of π can be obtained by a finite sequence of single-event extensions, the claim follows by induction.

To demonstrate that the relation $\propto$ satisfies Definition 5.3, i.e. assuming that $\pi \propto \pi'$ we need to show that

1. $\pi \bowtie \pi'$.

   This property holds by Claim 1.

2. For all maximal processes $\Pi' \sqsupseteq \pi'$, and for all finite processes $\widehat{\pi}' \sqsupseteq \pi'$ such that $\widehat{\pi}' \sqsubseteq \Pi'$, there exist finite processes $\widetilde{\pi}' \sqsupseteq \widehat{\pi}'$ and $\widetilde{\pi} \sqsupseteq \pi$ such that $\widetilde{\pi}' \sqsubseteq \Pi'$ and $\widetilde{\pi} \propto \widetilde{\pi}'$.

   This property follows from Claim 3.

3. For all maximal processes $\Pi \sqsupseteq \pi$, and for all finite processes $\widehat{\pi} \sqsupseteq \pi$ such that $\widehat{\pi} \sqsubseteq \Pi$, there exist finite processes $\widetilde{\pi} \sqsupseteq \widehat{\pi}$ and $\widetilde{\pi}' \sqsupseteq \pi'$ such that $\widetilde{\pi} \sqsubseteq \Pi$ and $\widetilde{\pi} \propto \widetilde{\pi}'$.

   This property follows from Claim 4. □

Proposition 5.4. *Validity condition for transition splitting* $t\wr\!\!\rightarrow$

Let $t\wr\!\!\rightarrow$ *be an input-proper transition splitting transforming an input-proper LPN* $\Upsilon = (\Sigma, \mathcal{I}, \mathcal{O}, \ell)$ *into* $\Upsilon' = (\Sigma', \mathcal{I}, \mathcal{O}, \ell)$. *Then* $\Upsilon \multimap \Upsilon'$. □

The validity condition for the transition splitting $t\wr\!\!\rightarrow$ can be proven in a similar way as the one for $\rightarrow\!\wr t$.

5.2.3.2 Concurrent insertion

A silent τ transition can be inserted concurrently to existing transitions as shown in Figure 5.6.

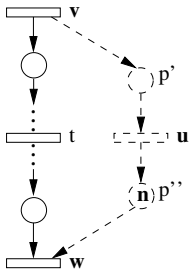


Figure 5.6: Concurrent insertion $v \stackrel{n}{\longrightarrow\!\!\!\!\!\rightarrow} w$

Definition 5.6. Concurrent insertion

Given an LPN $\Upsilon = (\Sigma, \mathcal{I}, \mathcal{O}, \ell)$ where $\Sigma = (P, T, F, M_0)$, two transitions $v, w \in T$

and an $n \in \mathbb{N}$, the transformation $v \stackrel{n}{\longrightarrow\!\!\!\!\!\rightarrow} w$, yielding an LPN $\Upsilon' = (\Sigma', \mathcal{I}, \mathcal{O}, \ell)$ with $\Sigma' = (P', T', F', M_0')$ is defined as follows:

- $T' = T \cup \{u\}$, where $u \notin P \cup T$ is a new transition;
- $P' = P \cup \{p', p''\}$, where $p', p'' \notin P \cup T$ are two new places;
- $F' = F \cup \{v, p'\} \cup \{p', u\} \cup \{u, p''\} \cup \{p'', w\}$;
- For all places $p \in P$, $M_0'(p) = M_0(p)$, $M_0'(p') = 0$ and $M_0'(p'') = n$.

The notation $v \longrightarrow\!\!\!\!\!\rightarrow w$ is used instead of $v \stackrel{0}{\longrightarrow\!\!\!\!\!\rightarrow} w$. The transformation $v \stackrel{n}{\longrightarrow\!\!\!\!\!\rightarrow} w$ is called *input-proper* if $\ell(w) \notin \mathcal{I}$ and u is a new silent transition such that $\ell(u) \notin \mathcal{I} \cup \mathcal{O}$. $\Diamond$

Note that if a transformation $v \stackrel{n}{\longrightarrow\!\!\!\!\!\rightarrow} w$ is preformed such that $v^\bullet = {}^\bullet w$ then it corresponds to a transition splitting either $v \wr\!\!\rightarrow$ or $\rightarrow\!\!\wr w$, where in the former $v^\bullet$ is redundant and is replaced by p' or in the latter ${}^\bullet w$ is redundant and is replaced by p''. Otherwise, there exist at least one transition which is concurrent with u. One can easily show that if an input proper concurrent insertion is applied to an input-proper LPN Υ, then the resulting LPN Υ' is also input proper.

Proposition 5.5. *Validity condition for a concurrent insertion*
Let $v \stackrel{n}{\longrightarrow\!\!\!\!\!\rightarrow} w$ be a concurrent insertion transforming an input-proper and safe LPN $\Upsilon = (\Sigma, \mathcal{I}, \mathcal{O}, \ell)$ into $\Upsilon' = (\Sigma', \mathcal{I}, \mathcal{O}, \ell)$, such that $\ell(w) \notin \mathcal{I}$, in every simple path from v and w and every place p on this path $|{}^\bullet p| = 1$ and $|p^\bullet| = 1$, and $n = 1$ if there is a place p on some of this path such that $M_0(p) = 1$, otherwise $n = 0$. Then $\Upsilon \multimap \Upsilon'$. $\square$

The proof of the validity condition for a concurrency reduction combines ideas from concurrency reduction and transition splitting.

5.3 Resolution concept

The compact representation of $\mathcal{X}$ conflicts is important for the resolution of encoding conflicts. $\mathcal{X}$ conflicts can be efficiently resolved by adding auxiliary signals, and some

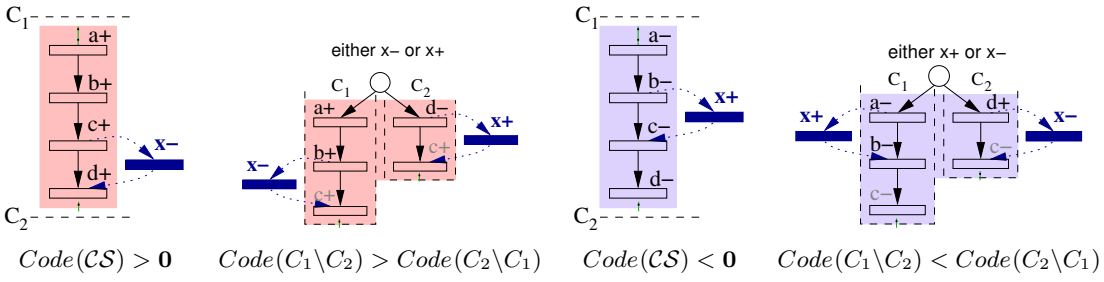
$\mathcal{X}$ conflicts can be eliminated by concurrency reduction. The former employs additional signals to resolve encoding conflicts and the latter reduces the state space and thus eliminates potential encoding conflicts. However, in the elimination of CSC_X^z cores the existing signals, which are not in X, are used to disambiguate the conflicts. The above mentioned approaches can also be applied, e.g. for logic decomposition. A framework is presented which uses additional auxiliary signals and concurrency reduction to eliminate $\mathcal{X}$ cores and the corresponding encoding conflicts.

First, a general approach is presented where the $\mathcal{X}$ cores are examined individually. A resolution strategy is proposed for each individual conflict depending on their properties. These strategies are then employed for the elimination of $\mathcal{X}$ cores by signal insertion and concurrency reduction.

5.3.1 $\mathcal{X}$ cores elimination: a general approach

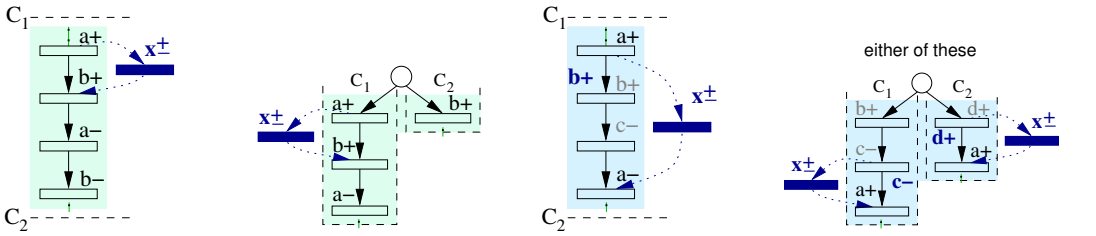
Encoding conflicts caused by $\mathcal{X}$ cores are similar in nature. An $\mathcal{X}$ conflict is represented by an $\mathcal{X}$ conflict pair of configurations $\langle C_1, C_2 \rangle$ whose final states are in an $\mathcal{X}$ conflict. In the case of normalcy violations it is necessary to make $Code(C_1)$ and $Code(C_2)$ not comparable, whereas in the USC/CSC/CSC_X^z violations it is necessary to make $Code(C_1)$ and $Code(C_2)$ distinguishable. This can be done by extending these conflicting states with an additional bit with different values in C_1 and C_2. However, to make C_1 and C_2 not comparable the value of the introduced bit is vital.

The strategies for eliminating $\mathcal{X}$ cores are shown in Figure 5.7. Recall that each core can be partitioned into $C_1 \backslash C_2$ and $C_2 \backslash C_1$. A normalcy core can be eliminated by incorporating a transition with a negative polarity (positive polarity) into one of the partitions if the encoding in this partition is greater (smaller) than the other. Moreover, if the normalcy core is of type I then one of these partitions becomes empty, whereas the other is the core $\mathcal{CS}$. Thus, if the encoding in $\mathcal{CS}$ is positive (negative) then a transition with a negative polarity (positive polarity) is added. The polarity is important, because the additional bit makes the encodings not comparable, resulting in the violation of the normalcy property $Code(C_1) < Code(C_2)$. The elimination of normalcy cores is schematically illustrated in Figure 5.7(a). For example, adding a transition x^- into the



$Code(\mathcal{CS}) > \mathbf{0}$ $Code(C_1\backslash C_2) > Code(C_2\backslash C_1)$ $Code(\mathcal{CS}) < \mathbf{0}$ $Code(C_1\backslash C_2) < Code(C_2\backslash C_1)$

(a) normalcy cores



$Code(\mathcal{CS}) = \mathbf{0}$ $Code(C_1\backslash C_2) = Code(C_2\backslash C_1)$ $Code_a(\mathcal{CS}) = \mathbf{0}$ $Code_a(C_1\backslash C_2) = Code_a(C_2\backslash C_1)$

(b) USC/CSC cores (c) CSC_a^z cores

Figure 5.7: Strategies for eliminating $\mathcal{X}$ cores

type I core $\mathcal{CS} = \{a^+, b^+, c^+, d^+\}$ makes the binary values in C_1 and C_2 not comparable, because the binary value of x is 1 in C_1 and 0 in C_2. In type II normalcy cores either a negative auxiliary transition is added in the positive partition or a positive auxiliary transition is added in the negative partition. Note that if $Code(C_1) < Code(C_2)$ in a n-normal core (violation) for a signal z than $Nxt_z(C_1) < Nxt_z(C_2)$, and consequentially, if $Code(C_1) < Code(C_2)$ in a p-normal core (violation) for a signal z than $Nxt_z(C_1) > Nxt_z(C_2)$.

If the encodings of C_1 and C_2 are equal then they must be made non-ambiguous by adding a transition, which can be either positive or negative, into the core. The polarity is not important, because it is enough to make the encoding in C_1 and C_2 disambiguate. The elimination is presented in Figure 5.7(b) and (c). 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 distinguishable. Due to the fact that the CSC_X^z violation corresponds to the support X for a signal z, this also makes it possible to extend the support X by incorporating an existing transition, which does

not correspond to X, in the core. For example the core $\mathcal{CS} = \{a^+, b^+, c^-, a^-\}$ can be eliminated by using a transition, e.g. b^+, from $\mathcal{CS}$ to disambiguate the conflicting states and take its signal into the support.

5.3.2 $\mathcal{X}$ core elimination by signal insertion

State coding conflicts can be efficiently resolved by introducing "additional memory" to the system in the form of internal signals. Depending on $\mathcal{X}$ the conflicting states are made either not comparable or disambiguated by the "additional memory". This approach requires behavioural equivalence of the specification and a guarantee that the signals are implemented without hazards, preserving speed independence [19]. In order to resolve the conflicting states caused by $\mathcal{X}$ cores, auxiliary signals must be added to the specification. Two methods of insertion are described. First, single signal insertion is described, which introduces one signal per targeted core. Secondly, dual signal insertion is presented which produces flip flops by inserting complementary signal pairs.

5.3.2.1 Single signal insertion

In order to resolve the conflicting states caused by an $\mathcal{X}$ core, an auxiliary signal aux is added to the specification. The elimination of an $\mathcal{X}$ core is schematically illustrated in Figure 5.8. An $\mathcal{X}$ core can be eliminated by inserting a transition of aux, say aux^+, somewhere in the core to ensure that it is no longer a conflict set. 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. The reason is that this would impose constraints on the environment which were not present in the original STG, making input signal "wait" for the newly inserted signal. A transition can be inserted either by splitting an existing transition, or by inserting a transition concurrent with an existing transition.

The elimination of $\mathcal{X}$ cores is illustrated by using typical cases in STG specifications. Figure 5.9 shows a case where two CSC cores are in sequence. They can be eliminated in

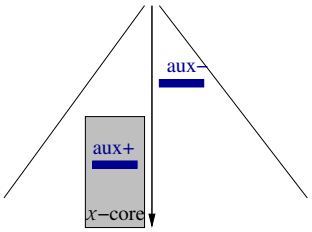
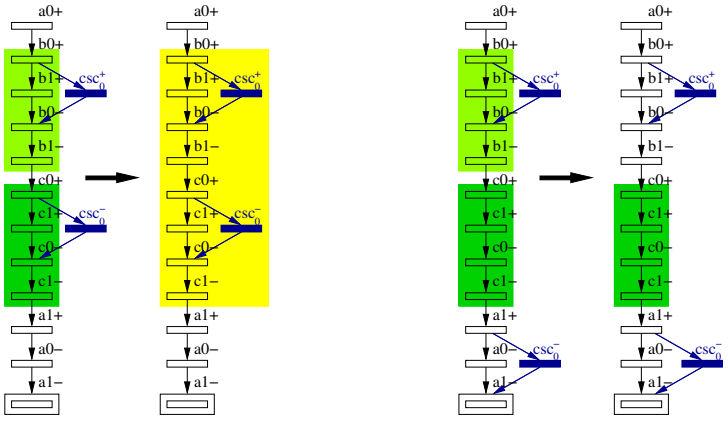


Figure 5.8: $\mathcal{X}$ core elimination by signal insertion

a "one-hot" manner. Each core is eliminated by one signal transition, and its complement is inserted outside the core, preferably into non-adjacent one. The union of two adjacent cores is usually a conflict set, which will not be destroyed if both the transition and its counterpart are inserted into it. This case is shown in Figure 5.9(a), where the rising transition of a newly added signal, csc_0^+, is inserted in the first core and its counterpart is inserted in the second core. This results in another core formed by the union of those cores including the newly inserted signal. However, this core can be eliminated by another internal signal. The insertion of a transition into the first core and its counterpart outside the core, but not in the adjacent core, is shown in Figure 5.9(b). The second core can be eliminated in the same way. Both approaches use two additional internal signals to eliminate the cores.



inputs: a_0, b_0, c_0; **outputs:** a_1, b_1, c_1; **internal:** csc_0

Figure 5.9: Example: elimination of CSC cores in sequence

The case where a CSC core is located in a concurrent part is shown in Figure

5.10(a). It can be also tackled in a one-hot way. A signal transition, say $csc+$, is inserted somewhere in the core and its counterpart outside the core. Note that in order to preserve the consistency the transition's counterpart cannot be inserted into the concurrent part, but it can be inserted before the fork transition or after the join. In this example, the counterpart transition $csc-$ can be only inserted after $b+$, splitting $b+$. Inserting $csc-$ before $b+$ would not eliminate the core but it would extend the core by the csc signal. The same happens if $csc-$ is inserted after $a+$. The insertion before $a+$ is not possible because $a+$ corresponds to an input signal.

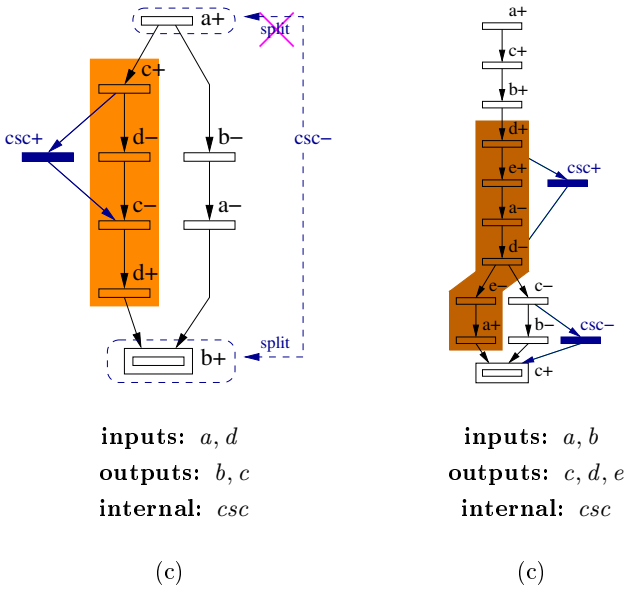
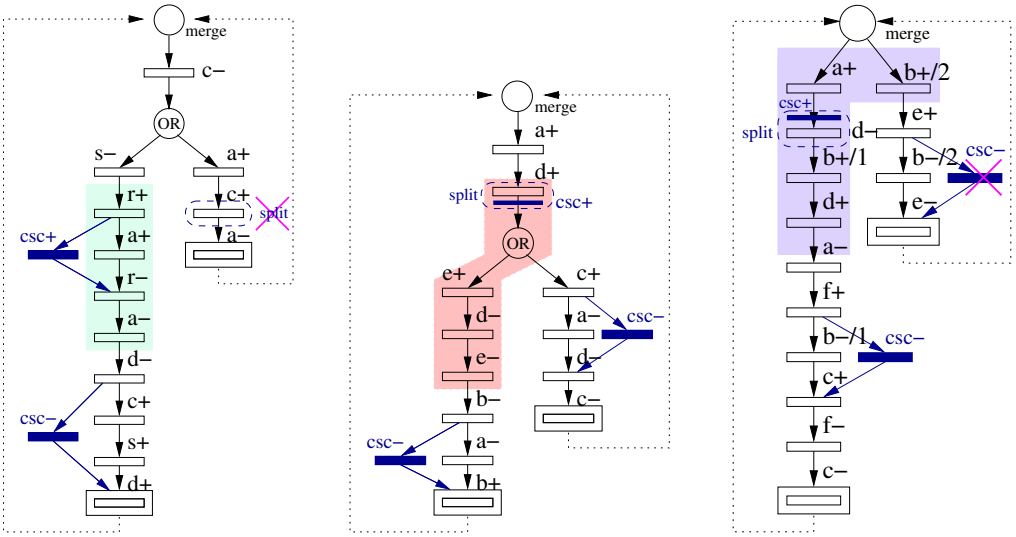


Figure 5.10: Example: elimination of CSC cores in concurrent parts

$\mathcal{X}$ cores partly located in a concurrent part as the one shown in Figure 5.10(b), where $\mathcal{X}$ is CSC, can be handled in the same way. If a transition is inserted in a core in a concurrent part then its counterpart cannot be inserted concurrent to it. However, if a transition is inserted in a core in a sequential part then its counterpart is inserted somewhere outside the core as illustrated in Figure 5.10(b).

The elimination of $\mathcal{X}$ cores which are located in branches which are in structural conflict (due to choice) are also done in a one-hot way. First consider the case where a CSC core is completely located in one conflicting branch as shown in Figure 5.11(a). In order to preserve consistency, the transition inserted outside the core cannot be inserted in the conflicting branch. It must be inserted in the same branch as the core, between



inputs: a, c, s
outputs: d, r
internal: *csc*

(a)

inputs: a, c
outputs: b, d, e
internal: *csc*

(b)

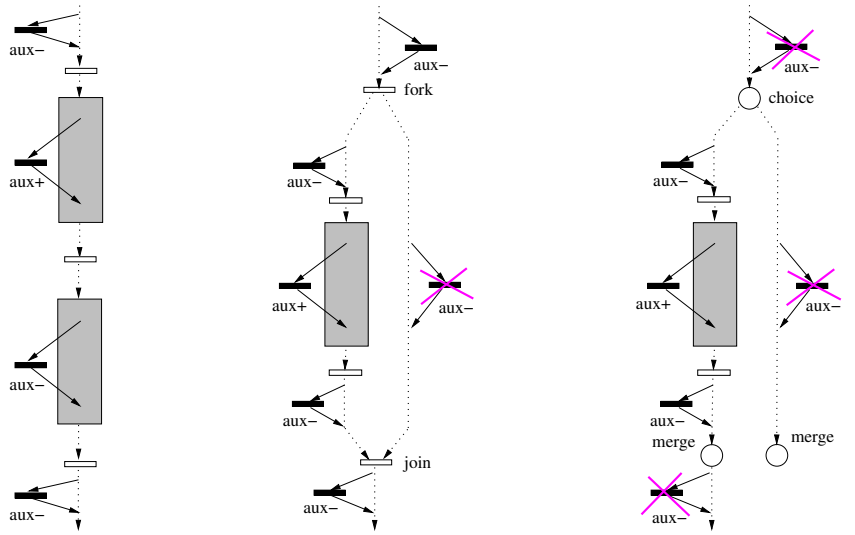
inputs: a, b, c
outputs: d, e, f
internal: *csc*

(c)

Figure 5.11: Example: elimination of CSC cores in choice branches

the choice and the merge point, as shown in Figure 5.11(a). Alternatively, a transition can be inserted in *each* branch and its counterpart before the choice point or after the merge point, or vice versa, as demonstrated in Figure 5.11(b). A core can be partly located in a conflicting branch or in more than one conflicting branch as shown in Figure 5.11(b) and (c), respectively. Those cases can be solved in the same way, as long as the counterpart to the additional transition in the $\mathcal{X}$ core is inserted in such a way that it does not validate the consistency property.

Figure 5.12 summarises 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 Figure 5.12(a). Each $\mathcal{X}$ core is eliminated by one signal transition, and its complement is inserted outside the $\mathcal{X}$ core, preferably into non-adjacent one. An STG that has a $\mathcal{X}$ core in one of the concurrent branches and in one of the branches which are in structural conflict, respectively, can also be tackled in a one-hot way, as shown in Figure 5.12(b) and (c), respectively. In order to preserve the consistency the transition's counterpart cannot be inserted into the concurrent branch or into the conflicting branch. Obviously, the described cases do not cover all possible situations or all possible insertions (e.g.



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

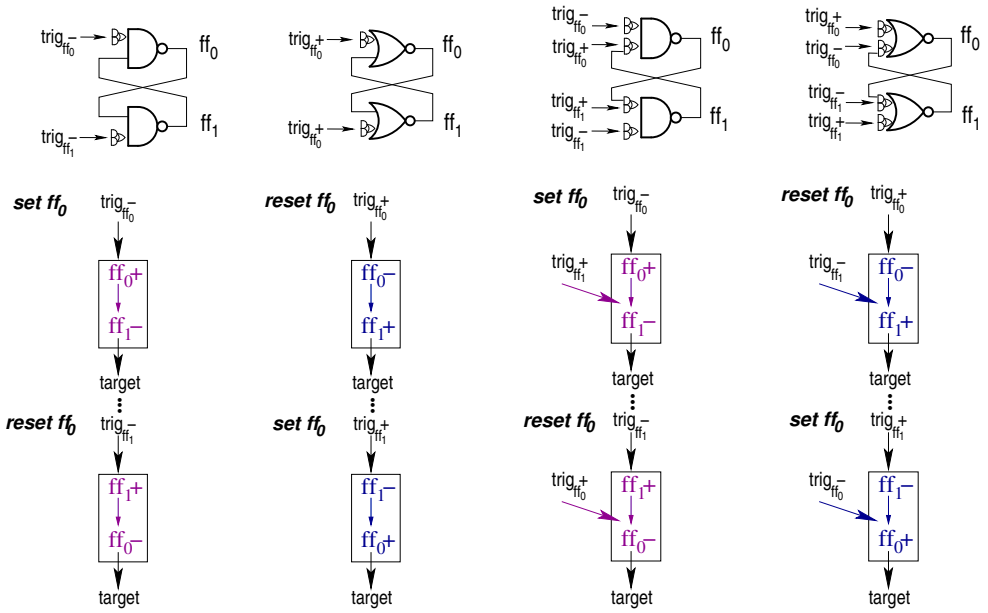
Figure 5.12: Strategies for $\mathcal{X}$ core elimination

a signal transition can sometimes be inserted before the choice point or after a merge point and its counterparts inserted into *each* branch, etc.), but they give an idea how the $\mathcal{X}$ cores can be eliminated.

5.3.2.2 Dual signal insertion (flip flop insertion)

The insertion of pairs of complementary signals $[ff_0^{\pm}, ff_1^{\mp}]$ in the STG implicitly adds a pair of negative gates to the circuit and creates a flip flop, which makes one part of the STG unordered with the other. This type of insertion gives the designer opportunities for potential improvements to the circuit. The insertion of pairs of complementary signals is illustrated in Figure 5.13. The insertion of two auxiliary signal transitions ff_0 and ff_1, such that $ff_0^{\pm}$ is a direct predecessor of $ff_1^{\mp}$ is denoted by $ff_0^{\pm} \twoheadrightarrow ff_1^{\mp}$. The transformation $[ff_0^{+}, ff_1^{-}]$ denotes the insertion of $ff_0^{+} \twoheadrightarrow ff_1^{-}$ and $ff_1^{+} \twoheadrightarrow ff_0^{-}$, and similarly the transformation $[ff_0^{-}, ff_1^{+}]$ denotes the insertion of $ff_0^{-} \twoheadrightarrow ff_1^{+}$ and $ff_1^{-} \twoheadrightarrow ff_0^{+}$. The former is realised by a flip flop constructed from two NAND gates and the latter is realised by a flip flop constructed from two NOR gates. The flip flop insertion is classified as follows:

Simple flip flop insertion illustrated in Figure 5.13(a) performs a transformation, e.g



(a) simple (b) complex

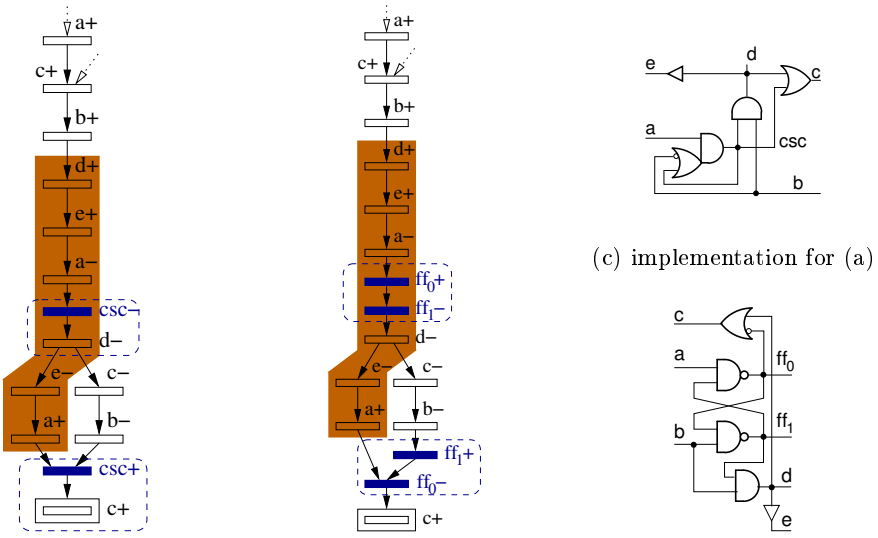
Figure 5.13: Strategy for flip flop insertion

$[ff_0^+, ff_1^-]$, where ff_1^- and ff_0^- each have only one direct predecessor, ff_0^+ and ff_1^+, respectively. The signal ff_0 is set by at least the trigger of ff_0^+ and is reset by at least the trigger of ff_0^-. Note that contextual signals may be involved for which additional logic is required at the inputs of the flip flop.

Complex flip flop insertion illustrated in Figure 5.13(b) works in a similar way, but if the transformation $[ff_0^+, ff_1^-]$ is inserted ff_1^- and ff_0^- have each at least one direct predecessor. If both have only one direct predecessor than the insertion corresponds to a simple flip flop insertion. Otherwise, the extra triggers contribute to additional logic at the criss-cross inputs of the flip flop. Note that contextual signal may also contribute to the additional logic. In Figure 5.13(b) in the transformation $[ff_0^+, ff_1^-]$ both ff_1^- and ff_0^- have two triggers, ff_1^- is triggered by ff_0^+ and $trig_{ff_1}^+$, and ff_0^- is triggered by ff_1^+ and $trig_{ff_0}^+$, which control the criss-cross inputs.

In both cases a flip flop is realised. Its complexity depends on the triggers of ff_0 and ff_1. Due to the nature of flip flops, and to avoid hazards, the set and reset functions

must be non-overlapping, e.g. if it is constructed from two NAND gates and both inputs go to '0' then both outputs go to '1', violating the fact that both outputs must be complements of each other.



(a) single signal insertion (b) dual signal insertion (c) implementation for (a) (d) implementation for (b)

inputs: a, b; **outputs:** c, d, e; **internal:** csc, ff_0, ff_1

Figure 5.14: Example: flip flop insertion

Consider the example in Figure 5.14, where single and dual signal insertion are used to eliminate the CSC core. The former is depicted in Figure 5.14(a), showing the sequential insertion of signal csc; csc^- is inserted in the core $\rightarrow\!\wr d^-$ and csc^+ is inserted outside the core $\rightarrow\!\wr c^+$. The resulting implementation is shown in Figure 5.14(c).

The dual signal insertion is presented in Figure 5.14(b), where $[ff_0^+, ff_1^-]$ are inserted realising a NAND flip flop; $ff_0^+ \twoheadrightarrow ff_1^-$ is inserted in the core $\rightarrow\!\wr d^-$ and $ff_1^+ \twoheadrightarrow ff_0^-$ is inserted outside the core $\rightarrow\!\wr c+$, such that ${}^\bullet({}^\bullet ff_0^-) = \{a^+, ff_1^+\}$. According to the rule above, this transformation should result in an complex flip flop because ff_0^- has more than one predecessor. However, due to the fact that a^- is a trigger of ff_0^+, the triggers of the signal ff_0 are a and ff_1. Thus the flip flop does not need additional control logic. It is set by a and reset by b. The implementation of this transformation is depicted in Figure 5.14(d). Note that the flip flop insertion corresponds to inserting the auxiliary signal ff_0 first ($ff_0^+ \rightarrow\!\wr d^-$ and $ff_0^- \rightarrow\!\wr c+$) followed by ff_1 ($ff_1^- \rightarrow\!\wr d^-$ and $ff_1^+\ b^- \wr\!\rightarrow$).

The dual signal insertion introduces a delay of two units whereas a single signal insertion introduces a delay of only one unit (but this one unit is typically a complex gate, which is implemented by two negative gates.) For example, d^- is delayed by ff_0^+ and ff_1^- in the former case and d^- is delayed by csc^- in the latter. However, due to the simplicity of the gates in the implementation with flip flops, the latency of both implementations is similar. Generally, the dual signal insertion can be used for the decomposition of complex gates.

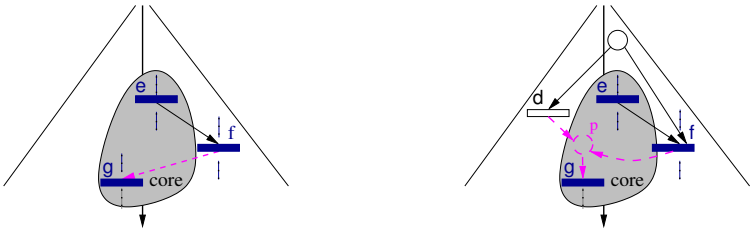
5.3.3 $\mathcal{X}$ core elimination by concurrency reduction

Concurrency reduction involves removing some of the reachable states of the STG, and can therefore be used for the resolution of encoding conflicts. The elimination of $\mathcal{X}$ cores by concurrency reduction involves the introduction of additional ordering constraints, which fix some order of execution. 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 resolve the conflicting states in an $\mathcal{X}$ core by delaying the starting point or bringing forward the end point of concurrency. Depending on $\mathcal{X}$ the conflicting states are either made not comparable or unambiguous. If there is an event concurrent to an $\mathcal{X}$ core, and a starting or end point of concurrency in the core, then this event can be forced into the core by an additional ordering constraint. Thus, the core is eliminated, unless the core corresponds to a normalcy violation and the polarity of the transition used does not eliminate the core.

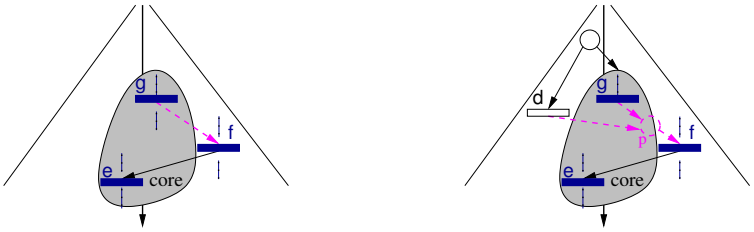
Two kinds of concurrency reduction transformations for $\mathcal{X}$ core eliminations are described below.

Forward concurrency reduction illustrated in Figure 5.15(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 $\mathcal{X}$ core in structural conflict with each other, and concurrent to g, an event in the $\mathcal{X}$ core. It is assumed that e is in the $\mathcal{X}$ core, either $e \prec g$ or $e \parallel g$, and for exactly one event $f \in E_U$, $e \prec f$.

Backward concurrency reduction illustrated in Figure 5.15(b) works in a similar



(a) forward



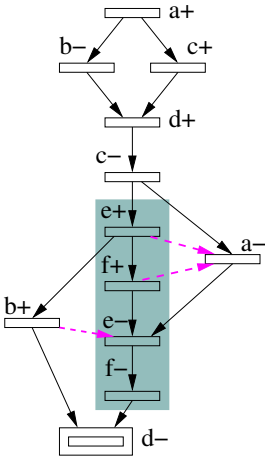
(b) backward

Figure 5.15: $\mathcal{X}$ core elimination by concurrency reduction

way, but the concurrency reduction $h(E_U) \overset{n}{\dashrightarrow} h(f)$ is performed. It is assumed that e is in the $\mathcal{X}$ core, f is an event outside the $\mathcal{X}$ core such that $f \prec e$, 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 $\mathcal{X}$ core, and either $g \prec e$ or $g \parallel e$.

In both cases the $\mathcal{X}$ core is eliminated by additional ordering constraints "dragging" f into the core. Note that the polarity of the transition corresponding to f is important for the elimination of a p-normal and an n-normal core (see section 5.3.1). The elimination of such a core requires the polarity of $h(f)$ to be opposite to the one in the core, i.e. if the targeted core partition is negative than $h(f)$ must be positive in order to eliminate the core and vice versa.

These two rules are illustrated by the examples in Figure 5.16 and 5.17 where they are applied to CSC cores of types I and II . In Figure 5.16(a) instances of b^+ and a^- are concurrent to the CSC 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. This means that the conflicts corresponding to the core are disambiguated by the value of the signal b. Note that f is an input and thus cannot be delayed, and so the concurrency reductions $b^+ \dashrightarrow f^+$ and $b^+ \dashrightarrow f^-$ would be invalid. The backward

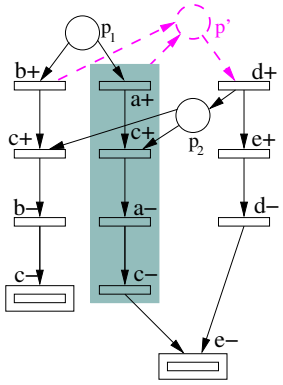


inputs: b, c, f;**outputs:** a, d, e

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

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

(a)



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

backward reduction: $\{a^+, b^+\} \dashrightarrow d^+$

(b)

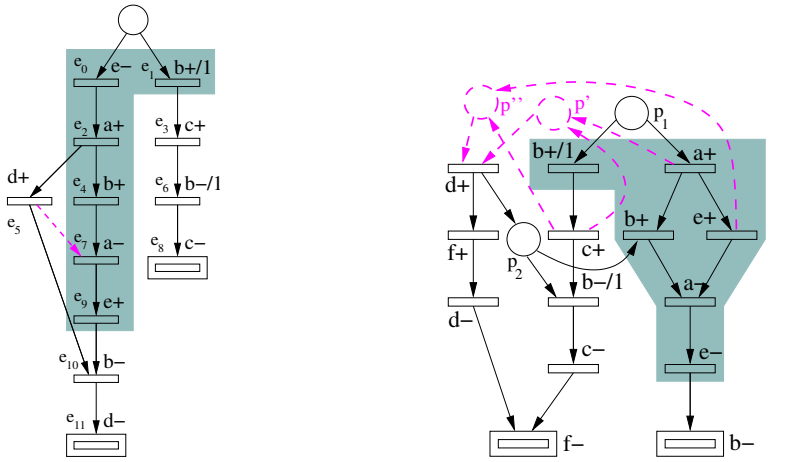
Figure 5.16: Elimination of type I CSC cores

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. The conflicts caused by the core are disambiguated by the signal a.

In Figure 5.16(b), d^+ is concurrent with events in the core and precedes c^+, an event in the core. The only event in the core which precedes or is concurrent to c^+ is a^+. However, $a^+ \dashrightarrow d^+$ is an invalid transformation, which introduces a deadlock. The concurrency reduction $\{a^+, b^+\} \dashrightarrow d^+$ is used instead, since $b^+ \# a^+$ and $b^+ \parallel d^+$.

Figure 5.17 shows the elimination of type II CSC cores. A forward concurrency reduction is illustrated in Figure 5.17(a). An instance of d^+ is concurrent to the core and succeeds a^+, an event in the core, and can therefore be used for a forward reduction. The only possible concurrency reduction is $d^+ \dashrightarrow a^-$, since $b^+$and e^+ is an input and thus cannot be delayed.

The backward concurrency reduction technique is illustrated in Figure 5.17(b), where d^+ is concurrent to a^+ and e^+ in the core and precedes b^+ in the core. The only



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

forward reduction: $d^+ \dashrightarrow a^-$

(a)

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

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

(b)

Figure 5.17: Elimination of type II CSC cores

events in the core which either precede or are concurrent with b^+ are a^+ and e^+, and either of these can be used to eliminate the core. However, both reductions $a^+ \dashrightarrow d^+$ and $e^+ \dashrightarrow d^+$ are invalid, since they introduce deadlocks, and are not allowed by the 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. (The backward concurrency reduction rule does not allow use of these two transformations, since only one event from the set E_U is allowed in the core.)

5.3.4 Resolution constraints

It is often the case that $\mathcal{X}$ cores overlap. In order to minimise the number of transformations (either concurrency reduction or signal insertion), 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 this method, viz. the *height map* showing the quantitative distribution of the $\mathcal{X}$ cores, is employed in the visualisation process. Figure 5.18 schematically illustrates the relationship between the representation of $\mathcal{X}$ cores and the height map. The events located in $\mathcal{X}$ cores are highlighted by shades of colours. The shade depends on the *altitude* of an event, i.e. on the number of cores to which it belongs. (The analogy with a topographical map showing altitude may be helpful here.) The greater the altitude, the darker the shade. "Peaks" with the highest altitude are good candidates for transformation, since they correspond to the intersection of maximum number of cores. In Figure 5.18 the area labelled "A3" is the area of highest density where all three cores intersect.

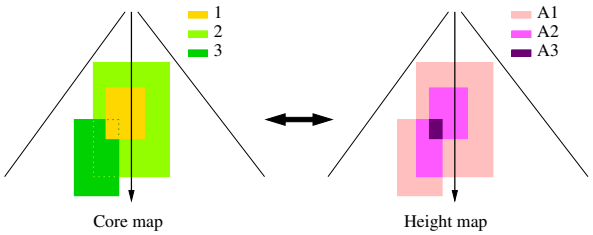
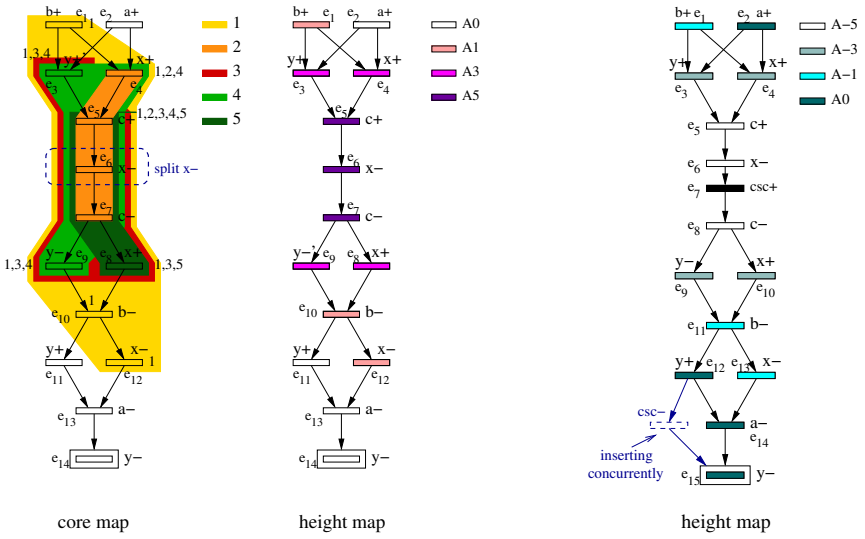


Figure 5.18: Visualisation of encoding conflicts

In Figure 5.19(a) there are five CSC cores altogether and all of them overlap. Peaks with the highest altitude are labelled "A5". It is possible to eliminate them by adding just one new signal: one of its transitions should be inserted into the intersection of these cores, and its counterpart into the remaining part as depicted in Figure 5.19. In the second phase of the signal insertion the altitudes are made negative in the height map by using the information from the height map in phase one. Note that the cores which are not targeted are not made negative. Additionally, in the height map those areas where a transition cannot be inserted are faded out preserving consistency. In the example in Figure 5.19 no events are in structural conflict with and concurrent with the transition inserted in phase one, thus no events are faded out.

The example in Figure 5.20 shows a n-normalcy violation for signal b. This is reflected in the equation for b, which is presented in Figure 5.20(b), where b is n-normal w.r.t a and c and p-normal w.r.t d. The violation is caused by two cores, $\mathcal{CS}_1$ and $\mathcal{CS}_2$



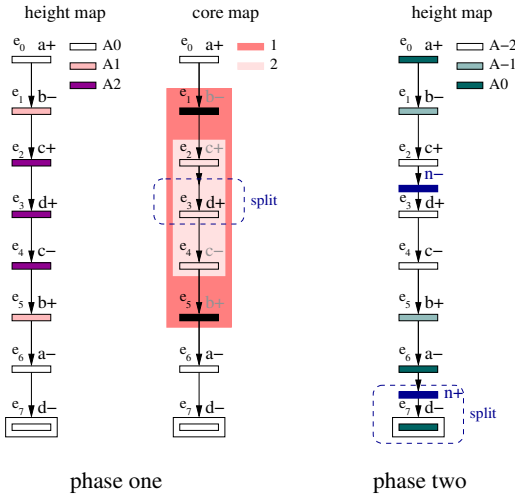
(a) phase one (b) phase two

inputs: a, b, c; **outputs:** x, y; **internal:** csc

Figure 5.19: Intersection of CSC cores

which overlap. Events labeled "A2" in the height map belong to the highest peaks and are good candidates for signal insertion. In phase one, a negative auxiliary transition n^- is inserted into the peak, because the encoding in $\mathcal{CS}_1$ and $\mathcal{CS}_2$ is positive. The only way to insert n^- into the peak is before $d+$. Note that the signal c is an input signal, and therefore cannot be delayed by an insertion. In phase two n^+ is inserted outside the cores, as indicated on the updated height map. The events with the altitude "A0" indicate that if n^+ is inserted in the highest peak, then the two targeted cores $\mathcal{CS}_1$ and $\mathcal{CS}_2$ are eliminated. This results in a normal STG, which can be implemented with monotonic gates as shown in Figure 5.20(c).

Using the height map representation, the designer can select areas for transformation and obtain a local, more detailed description of the overlapping cores with the selection. When an appropriate core cluster is chosen, the designer can decide how to perform the transformation optimally, taking into account the design constraints and knowledge of the system being developed.



(a) resolution process

$$b = \overline{c} \cdot d + \overline{a}$$
$$d = a \cdot d + c$$

(b) non-monotonic implementation

$$b = \overline{c} \cdot \overline{n} + \overline{a}$$
$$d = \overline{n}$$
$$n = \overline{c} \cdot \overline{d} + \overline{a}$$

(c) monotonic implementation

inputs: a, c; **outputs:** b, d; **internal:** n

Figure 5.20: N-normalcy violation for signal b

5.4 Resolution process

The advantage of using $\mathcal{X}$ cores is that only those parts of STGs which cause encoding conflicts, rather than the complete list of $\mathcal{X}$ conflicts, are considered. Since the number of $\mathcal{X}$ cores is usually much smaller than the number of encoding conflicts, this approach avoids the analysis of large amounts of information. Resolving encoding conflicts requires the elimination of cores by introducing either additional signals or causal constraints into the STG. A manual resolution and an automated resolution are presented. The former uses several interactive steps aimed at helping the designer to obtain a customised solution, and the latter uses heuristics for transformation which are based on $\mathcal{X}$ cores and their superpositions. However, in order to obtain an optimal solution, a semi-automated resolution process can be employed, where several precomputed solutions are suggested and the designer can choose the one which suits the design constraints best.

5.4.1 Manual resolution

The overview of the manual resolution process of encoding conflicts is illustrated in Figure 5.21. Given an STG, a finite complete prefix of its unfolding is constructed, and the $\mathcal{X}$ cores are computed. If there are none, the process stops. Otherwise, the height map is shown for the designer to choose a set of overlapping cores in order to perform a transformation. The transformation, either concurrency reduction or signal insertion, is chosen by the designer. The former can only be applied if concurrency exists and either starts or ends in the selected core overlaps, and if it is possible to either delay the starting point or bring forward the end point. If this is the case, a causal constraint is inserted and transferred to the STG, and the process is repeated.

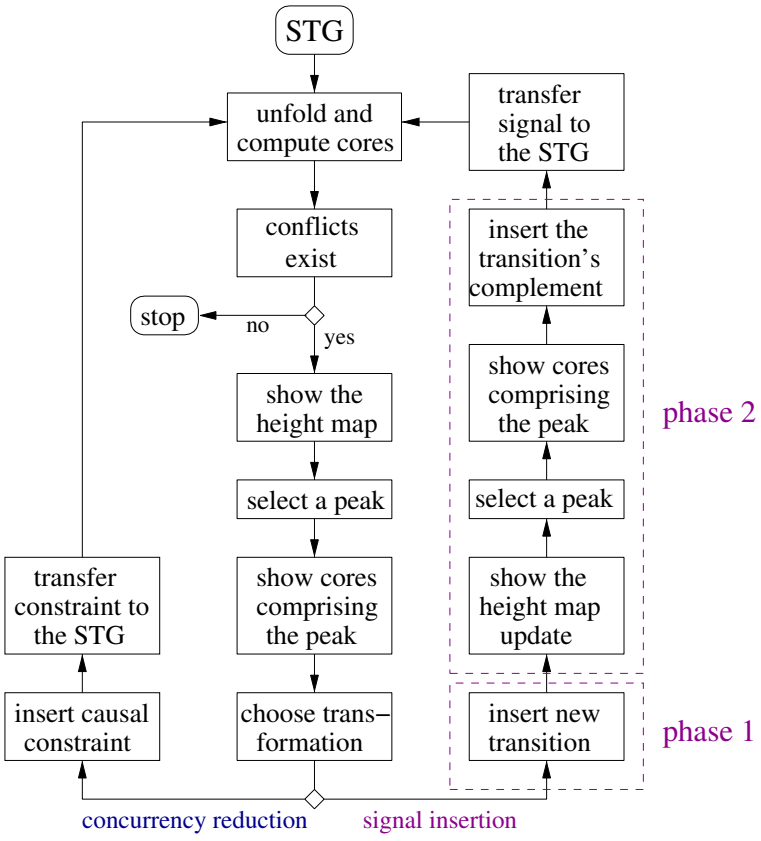


Figure 5.21: The resolution process of encoding conflicts

An auxiliary signal can be inserted into the specification in two phases. In phase one a signal transition is inserted and in phase two its counterpart is inserted eliminating the selected cores. The inserted signal is then transferred to the STG, and the process

is repeated. Depending on the number of $\mathcal{X}$ cores, the resolution process may involve several cycles.

After the completion of phase one, the height map is updated. The altitudes of the events in the core cluster where the new signal transition has been inserted are made negative, to inform the designer that if the counterpart transition is inserted there, some of the cores in the cluster will reappear. Moreover, in order to ensure that the insertion of the counterpart transition preserves consistency, the areas where it cannot be inserted (in particular, the events concurrent with or in structural conflict with this transition) are faded out.

5.4.1.1 Transformation condition

Next, conditions for transformation are presented, so that suitable transformations are selected. These conditions aim to minimise the circuit's latency and complexity. Transformations can introduce delays to non-input signal transitions, e.g. transition splitting delays at least one transition, whereas concurrency reduction may introduce a delay depending on the inserted causal constraint. Transformations such as concurrency reduction and concurrent signal transition insertion can introduce transitions which have a large number of incoming and outgoing arcs, respectively. This results in a circuit where some gates can be complex, i.e. have a large fan-in/fan-out. Modern asynchronous circuits are usually built of CMOS gates, with ASIC gate array libraries which consist of a restricted number of rather simple gates depending on four, or fewer variables. This means that the number of incoming and outgoing arcs of a transition should not exceed four. Clearly, the transformation which produces a minimal fan-in/fan-out should be chosen over the one with relatively high fan-in/fan-out.

The polarity of the introduced transitions can have an effect on the complexity of a circuit by introducing additional input inverters in the case of USC/CSC resolution. Those inverters are often assumed to have zero delay for the circuit to function correctly. If they are viewed as ordinary gates with delays, there could be a situation where such an inverter might switch concurrently with the gate it is connected to. Such a situation may lead to glitches. In the case of normalcy resolution, much more care is required

during the transformation not to destroy the existing normalcy. This is done in such a way that the polarity should not introduce different and/or contradictory triggers for existing normal signals.

The polarity for USC/CSC resolution is not crucial for the elimination of such encoding conflicts. Thus, the polarity could be chosen to preserve and/or to improve normalcy by means of triggers, e.g using consistent and non-contradictory triggers. This would ensure that the triggers, which generally arrive at the latest point, do not need to be inverted and thus reducing the risk of hazards. Additionally, it is essential that transitions in an STG alternate up and down to increase the use of negative gates. A transformation which generates an alternating polarity can reduce the complexity. The latter can also be applied to normalcy resolution.

The above conditions are summarised as follows, and should be applied to select a suitable transformation:

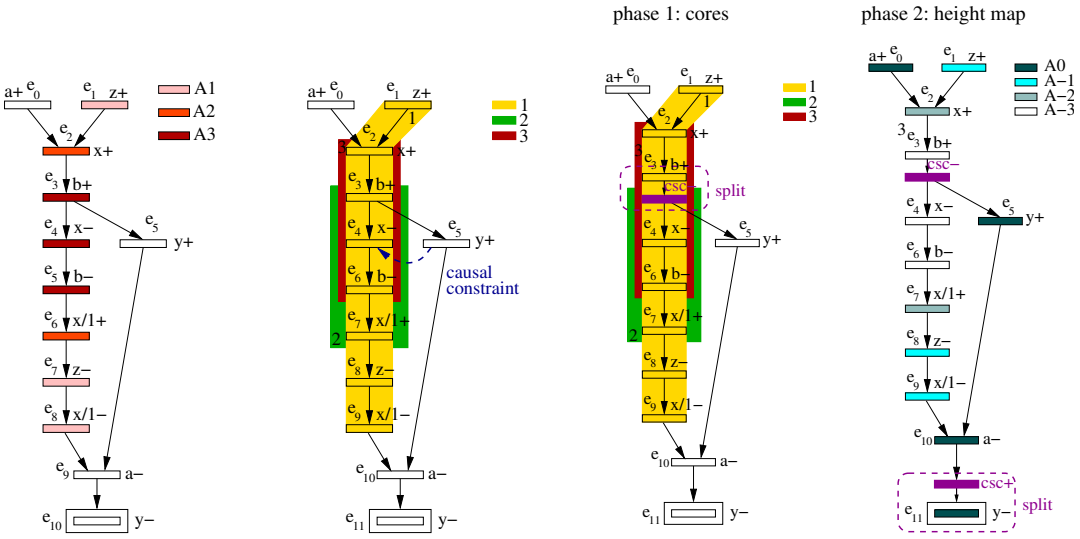
1. minimum delay
2. minimum fan-in/fan-out
3. consistent polarity of triggers
4. non-contradictory triggers
5. alternating polarity of transitions

Note, that conditions 3 and 4 are necessary for normalcy resolution, where e.g. the introduction of inconsistent triggers would violate the normalcy property.

5.4.1.2 Resolution examples

In order to demonstrate the resolution of encoding conflicts, two examples are presented showing CSC and normalcy resolution.

CSC conflict resolution Figure 5.22 illustrates the transformation process. The height map in Figure 5.22(a) shows the distribution of CSC cores. The events labeled "A3" belong to the highest peak, corresponding to three overlapping cores depicted



(a) height map (b) concurrency reduction (c) signal insertion

inputs: a, b; **outputs:** x, y, z; **internal:** csc

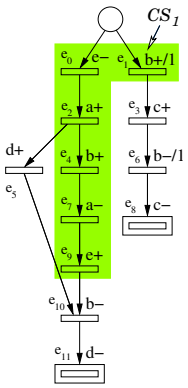
Figure 5.22: CSC conflict resolution

in Figure 5.22(b) and (c). The designer can choose either concurrency reduction or signal insertion. A forward concurrency reduction is possible because e_5 is concurrent with the events in the highest peak and e_5 succeeds e_3, which belongs to the highest peak. The causal constraint $h(e_5)$ $\dashrightarrow$ $h(e_4)$ is inserted and this eliminates the cores (see Figure 5.22(b)). Note the order of execution: e_5 is executed before e_4 and the arc between b^+ and x^- becomes redundant. Thus the signal y, which corresponds to e_5, is used to disambiguate the conflicting states corresponding to the cores. However, this transformation introduces a delay, x^- is delayed by y^+. Due to the fact that the polarities of the triggers of the signal x have been contradictory, the transformation does not effect this, the positive trigger of x^- is replaced by another positive trigger. In order to make the triggers of x non-contradictory the triggers of either x^+ and $x^-/1$ or x^- and $x^+/1$ must be modified.

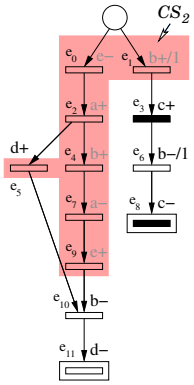
Signal insertion is shown in Figure 5.22(c). In phase one an auxiliary signal transition csc^- is inserted somewhere into the peak, e.g. csc^- can be inserted $h(e_3)\wr\!\!\rightarrow$. After updating the height map the second phase can be performed. In order to eliminate all three cores csc^+ must be inserted in the highest peak, which corresponds to events with

the altitude "A0", e.g. csc^+ is inserted $\rightarrow\wr e_{11}$. The signal csc is used to disambiguate the state coding conflicts. The transformation is transferred to the STG resulting in a conflict-free STG. The transformation introduces the following delays: the insertion in phase one delays x^- and y^+, and the insertion in phase two delays y^-. The new inserted signal csc and the signal y have non-contradictory triggers.

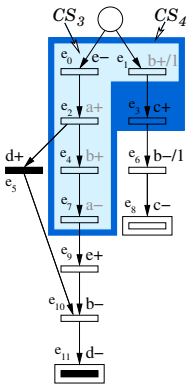
Normalcy conflict resolution The example shown in Figure 5.23 has a CSC conflict and a p-normalcy violation for signals c and d. The corresponding cores $\mathcal{CS}_1$ to $\mathcal{CS}_4$ are illustrated in Figure 5.23(a) to (c). It is necessary to eliminate $\mathcal{CS}_1$, which corresponds to CSC conflicts, to make the specification implementable. In order to obtain monotonic functions the remaining three cores must also be eliminated. The CSC core $\mathcal{CS}_1$ can be eliminated by applying a forward concurrency reduction $h(e_5) \dashrightarrow h(e_7)$ as shown in Figure 5.23(d). This transformation will also eliminate $\mathcal{CS}_3$ and $\mathcal{CS}_4$, because d^+ will make the conflicting states corresponding to $\mathcal{CS}_3$ and $\mathcal{CS}_4$ not comparable. Note that the encoding in $\mathcal{CS}_3$ only differs by the value of signal e and the encoding in $\mathcal{CS}_4$ only differs by the value of c and e. Therefore, adding a positive transition into the left partition of $\mathcal{CS}_3$ and $\mathcal{CS}_4$ results in the elimination of these cores. However, $\mathcal{CS}_2$ still remains. This is reflected in the equation for signal c (see Figure 5.23(d) step 1), where c is neither p-normal nor n-normal. It is p-normal w.r.t b and n-normal w.r.t. a and d. In order to eliminate $\mathcal{CS}_2$ a negative transition must be added into the left partition, say n_0^-, and its counterpart n_0^+ must be added in the same branch outside the core to preserve consistency. The only way to insert n_0^+ is $\rightarrow\wr d^-$, however n_0^+ is positive and would make d not normal, i.e. n_0^+ would trigger d^- (this would suggest that d is n-normal), but d^+ is triggered by a^+(this would suggest that d is p-normal). To overcome this n_0^- could be inserted $\rightarrow\wr d^+$ making the triggers of d n-normal as shown in Figure 5.23(d) in step 2. This solution satisfies the normalcy property, which is reflected in the equations. However, n-normal cores for d could appear after inserting n_0. This transformation would delay two transitions, as d^+ and d^- are delayed by n_0^- and n_0^+, respectively. Note that the concurrency reduction does not introduce any delay; d^+ is executed concurrently with $b+$.



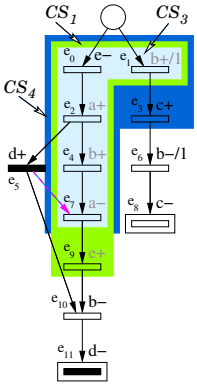
(a) CSC conflict



(b) p-normalcy violation for signal c

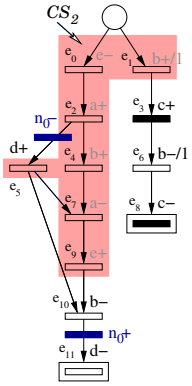


(c) p-normalcy violation for signal d



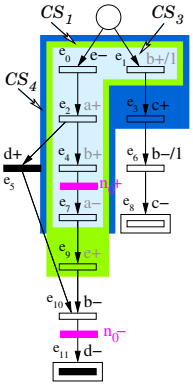
$a = \overline{e} \cdot (\overline{d} + \overline{b})$
$c = b \cdot \overline{d} \cdot \overline{a}$
$d = b \cdot d + a$

step 1



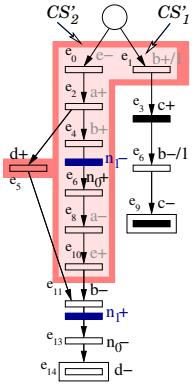
$a = \overline{e} \cdot (\overline{d} + \overline{b})$
$c = n_0 \cdot b \cdot e$
$d = \overline{n_0}$
$n_0 = \overline{a} \cdot (\overline{d} + \overline{b})$

step 2



$a = \overline{e} \cdot \overline{n_0}$
$c = b \cdot \overline{n_0} \cdot \overline{a}$
$d = n_0 + a$
$n_0 = b \cdot (n_0 + a)$

step 1



$a = \overline{e} \cdot \overline{n_0}$
$c = n_1 \cdot e \cdot b$
$d = n_0 + a$
$n_0 = \overline{n_1}$
$n_1 = \overline{a} \cdot \overline{n_0} + \overline{b}$

step 2

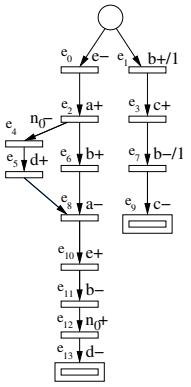
(d) first transformation

(e) second transformation

inputs: b, e; **outputs:** a, c, d; **internal:** n_0, n_1

Figure 5.23: Normalcy conflict resolution

Alternatively, the CSC core $\mathcal{CS}_1$ can be eliminated by a signal insertion, say n_0, which can be used to eliminate either the normalcy violation for c or the normalcy violation for d. Depending on the polarity of the transition, the insertion of one of these can also eliminate a normalcy violation. A positive transition of n_0 eliminates $\mathcal{CS}_3$ and $\mathcal{CS}_4$, and a negative transition eliminates $\mathcal{CS}_2$. The former is illustrated in Figure 5.23(e) in step 1. From the equation for d it can be seen that it is p-normal. However, c is still not normal. An additional signal insertion is needed to eliminate the new p-normal cores C_1' and C_2' (see Figure 5.23(e) in step 2), which arise after the first step of the resolution process. These cores can be eliminated by inserting a negative auxiliary transition n_1^- into the intersection of the cores and its counterpart outside the core, in such a way that the existing normalcy is not destroyed, e.g. without changing the polarity of triggers of the normal non-input signals. Inserting n_1^- before a^+ would not violate n-normalcy for a, since a^+ already has a negative trigger e^-. However, it is not possible to insert n_1^+ in the same branch without destroying some normalcy, e.g. if n_1^+ is inserted before n_0^- than n_0 would have contradictory triggers, positive trigger for both n_0^+ and n_0^-. The transition n_1^- can be inserted $\rightarrow\wr n_0^+$ and its counterpart $\rightarrow\wr n_0^-$. Based on their triggers this would make n_0 and n_1 n-normal. This transformation is presented in Figure 5.23(e) in step 2 with results in a monotonic implementation. This transformation delays a^- by n_0^+ and n_1^- and it also delays d^- by n_0^- and n_1^+.



(a) unfolding prefix

$$\begin{aligned} a &= \overline{e} \cdot (\overline{d} + \overline{b}) \\ c &= n_0 \cdot b \cdot e \\ d &= \overline{n_0} \\ n_0 &= \overline{a} \cdot (\overline{d} + \overline{b}) \end{aligned}$$

(b) equation before decomposition

$$\begin{aligned} a &= \overline{e} \cdot \overline{n_1} \\ c &= n_0 \cdot b \cdot e \\ d &= \overline{n_0} \\ n_0 &= \overline{a} \cdot \overline{n_1} \\ n_1 &= d \cdot b \end{aligned}$$

(c) equation after decomposition

Figure 5.24: Logic decomposition $(\overline{d} + \overline{b})$: signal insertion $n_1^+ \rightarrow\wr a^-$ and $n_1^- \rightarrow\wr n_0^+$

In the equations in Figure 5.23(d) in step 2 (also shown in Figure 5.24(b)) the term

$(\overline{d} + \overline{b})$ calls for a possible decomposition. A new signal n_1 can be inserted such that $n_1 = \overline{d} + \overline{b}$, which is then used as a new signal in the support for a and n_0. Such a transformation would contribute to the derivation of simpler equations. However, care must be taken not to increase the complexity of other functions.

In the equation $a = \overline{e} \cdot (\overline{d} + \overline{b})$ a^+ is triggered by e^- and a^- is triggered by d^+ and b^+, and in the equation $n_0 = \overline{a} \cdot (\overline{d} + \overline{b})$ n_0^+ is triggered by b^- and n_0^- is triggered by a^+. In order to decompose the term $(\overline{d} + \overline{b})$ by the new signal n_1, n_1 should be in the support of n_0 and a. Thus, the triggers of a^- and n_0^+ should be replaced by the transition of n_1. Note that a signal's triggers are guaranteed to be in the support of this signal. To retain normalcy n_1^+ is inserted $\rightarrow\wr a^-$. The result is that a is triggered by n_1 and e, and n_1 is triggered by b and d . Furthermore, the n-normality of the triggers of a is maintained. The counterpart transition n_1^- is inserted $\rightarrow\wr n_0^+$. The result is that b^-, the trigger of n_0^+, is replaced by n_0^+ making a and n_1 the triggers of n_0, and also maintaining the n-normalcy of the triggers of n_0. Note also that the newly inserted signal n_1 has p-normal triggers, and its triggers are also maintained, i.e. b is already a trigger for n_1^+. The equations of the resulting STG are shown in Figure 5.24(c), where logic $n_1 = d \cdot b$ is shared by a and n_0.

5.4.2 Automated resolution

In the automated resolution process heuristics for transformations (either signal insertion or concurrency reduction) are employed, based on the $\mathcal{X}$ cores, their suppositions and insertion constraints. Also, a heuristic *cost function* (see section 5.4.3) is applied to select the best transformation for the resolution of conflicts. It takes into account: (i) the delay caused by the applied transformation; (ii) the estimated increase in the complexity of the logic and (iii) the number of cores eliminated by the transformation.

The resolution process involves finding an appropriate transformation (either concurrency reduction or signal insertion) for the elimination of $\mathcal{X}$ cores in the STG unfolding prefix. An overview is illustrated in Figure. 5.25. The following steps are used to resolve the encoding conflicts conflicts:

1. Construct an STG unfolding prefix.

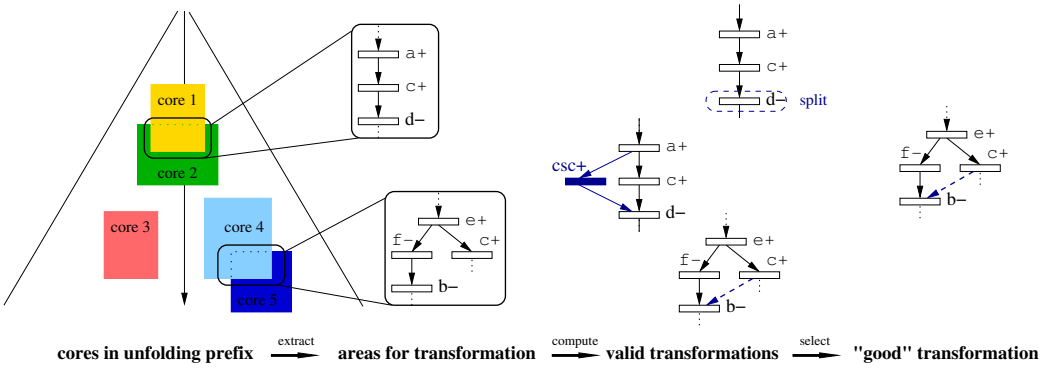


Figure 5.25: Overview: automated resolution process

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 go to 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.

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 best transformation and continue with step 1.

5.4.3 Cost function

A cost function was developed to select heuristically the best transformation at each iteration of the encoding conflict resolution process (either a concurrency reduction or a signal insertion). It is based on specification (STG unfolding prefix) level optimisation and is very approximate, but it is not computationally expensive. The cost function is

composed of three parts, taking into account the delay penalty inflicted by the transformation, the estimated increase in the complexity of the logic, and the number of cores eliminated by the transformation:

$$cost = \alpha_1 \cdot \Delta\omega + \alpha_2 \cdot \Delta logic + \alpha_3 \cdot \Delta core. \tag{5.1}$$

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).

The first part of the cost function 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 complete 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 [14, 18]. It is quite crude, but it is hard to improve it significantly, since the exact time behaviour is only known after the circuit and its environment have been synthesised.

Weighted events' depths in the unfolding prefix are used to determine the delay penalty of the transformation. The weighted depth ω_e of an event e is defined as follows:

$$\omega_e = \begin{cases} \omega_{h(e)} & \text{if } {}^\bullet({}^\bullet e) = \emptyset \\ \omega_{h(e)} + \max_{e' \in {}^\bullet({}^\bullet e)} \omega_{e'} & \text{otherwise,} \end{cases} \tag{5.2}$$

where ω_t is the delay associated with transition $t \in T$, which are chosen as follows:

$$\omega_t = \begin{cases} 3 & \text{if } t \text{ is an input transition} \\ 1 & \text{otherwise.} \end{cases} \tag{5.3}$$

These parameters can be fine-tuned by the designer if necessary.

For a concurrency reduction $h(U) \dashrightarrow^{n} t$, the delay penalty $\Delta\omega$ is computed as the difference in the weighted depths of a t-labelled event after and before the concurrency reduction. The value of $\Delta\omega$ is positive if t is delayed by the transformation, otherwise

it is 0. Note that the event's depth after the reduction is calculated *using the original prefix*. In the example in Figure 5.22(b) the transformation $y^+ \dashrightarrow x^-$ introduces one delay unit, since y^+ delays x^-, and y^+ corresponds to an output signal, whereas the transformation $d^+ \dashrightarrow a^-$ in Figure 5.23(d) in step 1 introduces a zero delay, because $\Delta\omega$ stays the same after the transformation.

For a signal insertion, several (at least two) transitions of an auxiliary signal *aux* are added to the STG. For each such a transition t, the inflicted delay penalty $\Delta\omega_t$ is computed, and then these penalties are added up to obtain the total delay penalty $\Delta\omega = \sum_t \Delta\omega_t$. If the insertion is concurrent, no additional delay is inflicted ($\Delta\omega_t = 0$), since in our delay model the transitions corresponding to internal signals are fast, and so their firing time cannot exceed that of the concurrent transitions. If the insertion is sequential, the inflicted delay penalty $\Delta\omega_t$ is computed by adding up the delay penalties of all the transitions u delayed by t: $\Delta\omega_t = \sum_u \Delta\omega_t^u$, where for each such u, the delay penalty $\Delta\omega_t^u$ is computed as the difference in the weighted depths of a u-labelled event after and before the transformation. Note that $\Delta\omega$ is calculated *using the original prefix*. In Figure 5.22(c) the transformation $b^+\wr\!\rightarrow$ and $\rightarrow\!\wr y^-$ introduces three delay units, because csc^- delays x^- and y^+ and csc^+ delays y^-.

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 local signal. Unfortunately, the estimation of the complexity can be inaccurate because it does not take into account contextual signals, which cannot be determined entirely until the CSC conflicts are resolved due to the impossibility deriving equations. The set of triggers of a signal $z \in Z$ is defined on the (full) unfolding as

$$trg(z) = \left\{ z' \in Z \;\middle|\; \exists e' \in \bigcup_{(\lambda \circ h)(e) = z^{\pm}} trg(e) : (\lambda \circ h)(e') = z'^{\pm} \right\}, \tag{5.4}$$

and can be approximated from below using a finite prefix of the STG unfolding.

The number of triggers of $z \in Z$ after the transformation is denoted by $trg'(z)$. (Note that for the transformations used, $trg'(z)$ can be approximated *using the original prefix*.)

For a concurrency reduction $U \overset{n}{\dashrightarrow} t$, where t is a z-labelled transition, the estimated increase in complexity of the logic $\Delta logic$ is computed as

$$\Delta logic = \mathfrak{C}(|trg'(z)|) - \mathfrak{C}(|trg(z)|), \tag{5.5}$$

where

$$\mathfrak{C}(n) = \begin{cases} 0 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ \lceil \frac{2^n}{n} \rceil & \text{if } n > 2 \end{cases}$$

estimates the number of binary gates needed to implement an n-input Boolean function. This formula was chosen because the asymptotic average number of binary gates in a Boolean circuit implementing an n-input Boolean function is $\frac{2^n}{n}$ [107], and because all the triggers of a signal z are always in the support of the complex gate implementing z. Note that the maximal number of triggers which can be added is $|U|$; the actual number of added triggers can be smaller if some of the signals labelling the transitions in U are already triggers of z. In fact, this number can even be negative, e.g. if an existing trigger ceases to be a trigger of z due to the transformation, and if the signals labelling the transitions in U are already triggers of z. The transformation in Figure 5.22(b) $y^+ \dashrightarrow x^-$ introduces one trigger y^- to the signal x. Note that even b^+ ceases to be the trigger of x^- its signal is still a trigger of the signal x, because b^- triggers $x^+/1$. Thus, the complexity $\Delta logic = 2$ for this transformation, where x had three triggers, a, b and z before transformation, and four, including y, after transformation.

The definition of $\Delta logic$ discourages solutions using complex gates with too many inputs: the penalty is relatively small if the number of triggers is small, but grows exponentially if the transformation adds new triggers to a signal which already had many triggers.

For a signal insertion, several local signals in the modified STG can be triggered by the new signal aux. Let Z' denote the set of all such signals. For each signal $z \in Z'$, the increase $\Delta logic_z$ in the complexity of the logic implementing z is estimated, and then these estimates are added up. (Note that $\Delta logic_z$ can be negative for some $z \in Z'$, e.g.

when *aux* replaces more than one trigger of z.) Moreover, the added signal *aux* has to be implemented, and thus introduces additional logic complexity, which is estimated and added to the result: $\Delta logic = \left(\sum_{z \in Z'} \Delta logic_z\right) + \Delta logic_{aux}$, where

$$\Delta logic_z = \mathfrak{C}(|trg'(z)|) - \mathfrak{C}(|trg(z)|) \tag{5.6}$$

for all $z \in Z'$, and

$$\Delta logic_{aux} = \mathfrak{C}(|trg'(aux)|). \tag{5.7}$$

Note that in the case of signal insertion, at most one additional trigger (viz. *aux*) per signal can be introduced. In the example in Figure 5.22(c) the transformation $b^{+}\wr\!\!\rightarrow$ and $\rightarrow\!\!\wr y^{-}$ introduces the complexity $\Delta logic = 2$. The transformation added *csc* to the triggers of x making the number of triggers four and $\Delta logic_x = 2$. It also replaced the trigger of y, a and b, by *csc*, resulting in $\Delta logic_y = -1$. In addition *csc* has two triggers a and b, where $\Delta logic_{csc} = 1$.

The third part of the cost function, $\Delta core$, estimates how many cores are eliminated by the transformation. It is computed by checking for each core 'touched' by the transformation whether it is eliminated or not, *using the original prefix*. During concurrency reduction the number of cores which are eliminated by the causal constraint is used. Recall the rules for concurrency reduction in section 5.3.3, where a transformation $h(f) \overset{n}{\dashrightarrow} h(g)$ and $h(g) \overset{n}{\dashrightarrow} h(f)$, respectively, is performed. Each transformation eliminates a core to which e and g belong. In the case where the core corresponds to normalcy violations it is eliminated if the appropriate polarity is used. The number of cores to which e and g belong excluding the cores to which f belong are used for the estimation. It may happen that during such a transformation also neighbouring cores to which f belongs but not e and g are also eliminated, e.g. if a forward reduction is performed it may happens that this reduction acts at the same time as a backward reduction for the cores to which f belong. Then the number of these additional cores is also included.

During signal insertion the height map's altitude in phase two is used for the estimation. In phase one some cores are targeted, which can be eliminated depending on

how the signal transition is reset in phase two. The altitudes of events where a signal transition is inserted are used to estimate the number of eliminated cores. This number also includes cores which have not been targeted in phase one but could be eliminated in phase two. While doing this, the following consideration should be taken into account: if both rising and falling transitions of the new signal are inserted into the same conflict set, it is not eliminated; in particular, if these transitions are inserted into adjacent cores, the conflict set obtained by uniting these cores will resurface as a new core on the next iteration (even though the original cores are eliminated).

The number of cores eliminated $\Delta core = -3$ in Figure 5.22 for both concurrency reduction $y^+ \dashrightarrow x^-$ and signal insertion $b^+ \wr\!\rightarrow$ and $\rightarrow\!\wr y^-$; whereas the transformation $d^+ \dashrightarrow a^-$ in Figure 5.23(d) in step 1 results in eliminating three cores out of four cores. The remaining core is not eliminated because to resolve the normalcy violation caused by it a transition having a negative polarity is required.

Note that for efficiency reasons the cost function should be computed on the original unfolding prefix at each iteration in the resolution procedure. This strategy significantly reduces the number of times the unfolding prefix has to be built, saving time.

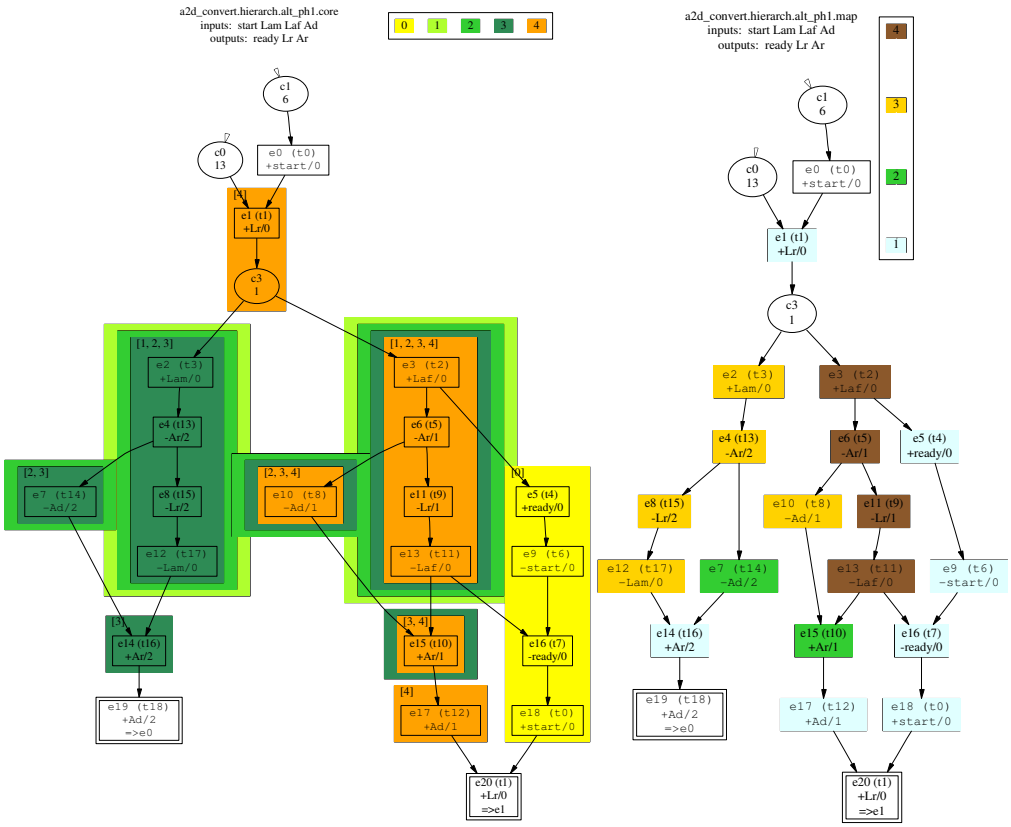
5.5 Tool ConfRes: CSC conflict resolver

An interactive tool ConfRes [23, 59, 58] has been developed for the resolution of CSC conflicts, which is based on the concept of cores. First, the applied visualisation method is discussed. Then an overview of the tool and how it can be used is described.

5.5.1 Applied visualisation method

The interactive conflict resolver ConfRes is intended to be used within the logic synthesis of asynchronous circuits. The synthesis is done by tools such as Petrify [18] or VeriSAT [40]. They are command-driven and use graph drawing tools for visualisation of the specification and its underlying modes. The applicability of ConfRes in the synthesis process suggests to reuse the existing graph drawing tools and not to use other more advanced and sophisticated 2D or 3D visualisation methods. This means

that the designer can retain the known environment of the used synthesis tool without the inconvenience of installing and understanding a new graphical tool.



(a) core map (b) height map

Figure 5.26: Visualisation of encoding conflict: an example

One of the tools used by the synthesis tools is Dot [51], which draws directed graphs as hierarchies. In addition to basic layouts, Dot allows edge labels, clusters, and other options that control layouts. It is able to produce graphics formats such as GIF or PostScript from attributed graph text files. These properties make Dot suitable for the visualisation of conflict cores and their superpositions. In addition to the graphical representation of the STG unfolding prefix, colours are used for visualisation. The cores are represented by assigning each core a colour and forming clusters of connected nodes. A set of 20 distinguishable colours have been chosen, which are ordered by their shades (light to dark). An example of core visualisation is illustrated in Figure 5.26(a), where

five cores are shown. Each core is assigned a number (0-4) and it corresponds to a colour. The nodes which belong to a cluster, e.g. node labeled with $e3, e6, e11$ and $e13$, are drawn in one block.

The height map is visualised in a similar way, but without clustering. Each node is assigned a colour which corresponds to the number of overlaps. The colours have been chosen in a similar way as in topographical maps. The lowers altitude (number of overlaps) is blue and the highest altitude is brown. The altitudes in between are green and yellow. The five cores in Figure 5.26(a) correspond to the height map shown in Figure 5.26(b). The altitude is assigned a colour and a number, which corresponds to the core overlaps. The highest altitude labelled "4" is brown, whereas the lowest altitude labelled "1" is blue.

5.5.2 Description

The resolution process based on CSC core visualisations is implemented as a software tool ConfRes . It is written in C++ and supports the manual and semi-automated resolution of encoding conflicts by either concurrency reduction or signal insertion. ConfRes depends on other tools as illustrated in Figure 5.27. It takes an STG in the *astg* format supported by Petrify [18], an STG-based synthesis tool. It uses Punf [38], a Petri net unfolder, to produce a finite and complete prefix of the STG, and VeriSAT [40], a SAT based model checker and an STG unfolding based synthesis tool, to detect encoding conflicts in STG unfoldings. Alternatively, conflicts can also be detected in STG unfoldings by the Clp tool [36], a linear programming model checker.

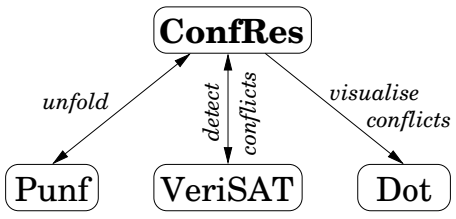


Figure 5.27: ConfRes dependencies

During the resolution process the cores and the corresponding height map are visualised using Dot [51], a graph drawing software by AT&T. After the resolution process

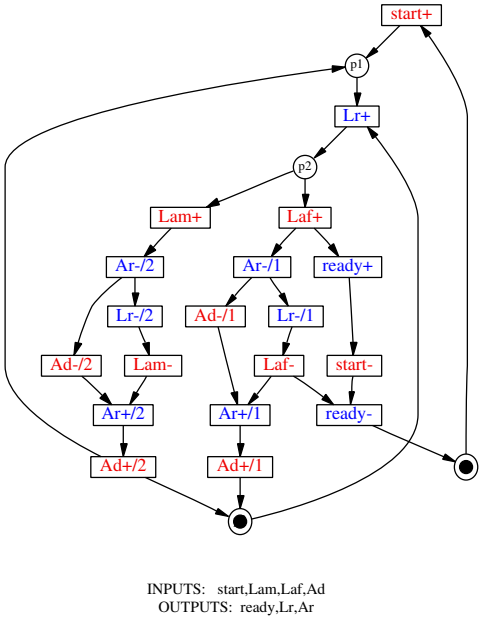
is completed, a synthesis tool, e.g. VeriSAT, or Petrify, can be used to synthesise the circuit.

The manual resolution works as shown in the process in Figure 5.21. 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.

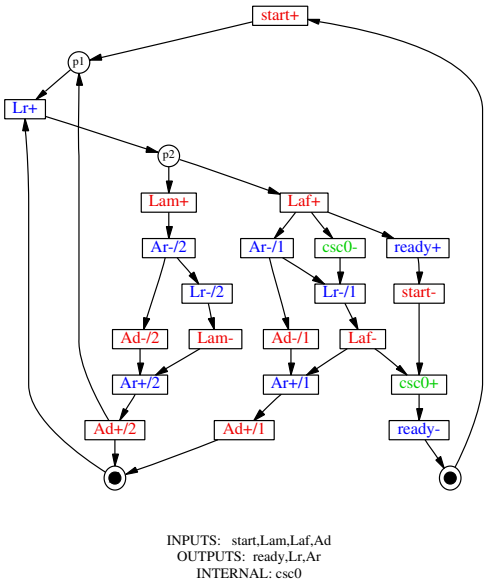
The semi-automated resolution works in a similar way. At every stage of the resolution process possible transformation are pre-computed, which are sorted according to the cost function, and act as a guideline for the designer. Note that for signal insertion the transformations include the insertions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

In Figure 5.28 and 5.26 an example is shown how ConfRes interacts with the user. The STG in Figure 5.28(a) does not satisfy the CSC property and the designer would like to intervene in the refinement process. The core representation in Figure 5.26(a) 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 (Figure 5.26(b)) 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 (semi-automated mode) are presented in Figure 5.28(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 sorted according to the cost function, where $\alpha_1 = \alpha_2 = \alpha_3 = 1$. The designer is free to select a



(a) initial STG



(b) final STG after insertion No 3

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

Figure 5.28: Interaction with ConfRes: an example

transformation, for example the transformation 3 is selected. The resulting conflict-free STG is depicted in Figure 5.28(b) with the new inserted signal $csc0$. The resulting STG can be synthesised by either PETRIFY or VERISAT.

5.5.3 Implementation

The implementation of encoding conflicts resolution based on core visualisation is presented in Algorithm 8. First, the finite and complete unfolding prefix is constructed from the given STG Γ using the PN unfolder PUNF. Then, the unfolding prefix is checked for CSC, by VERISAT, which detects pairs of configurations whose final states are in CSC conflicts. If any conflict pairs exist, cores are extracted from the corresponding conflict sets; otherwise, the STG is conflict free. The cores can be eliminated in two modes, manual and semi-automated. If the former is chosen then the designer has a complete freedom to decide the locations for the transformations (i.e. transformations can also be considered which violate the I/O interface preservation), and in the latter, transformations are pre-computed by the tool and are selected by the designer. The STG Γ is transformed by the selected transformation resulting in a new STG Γ'. The resolution process is repeated until all conflicts are resolved. Note that at each iteration the unfolding prefix is constructed from the transformed STG Γ'.

In the manual resolution the designer inspects the cores and the height map in order to decide the type and the location of a transformation. In case of signal insertion the designer has also to choose the location for the insertion in phase two, which is preformed by the procedure *insert_ph2*. This procedure repeats the insertion after updating the height map and fading out nodes which are concurrent with and in structural conflict with the transition inserted in the phase one. The polarities of the transitions are determined by the designer.

In the semi-automated resolution, procedure *computeTransformations,* computes all possible transformations (either signal insertion or concurrency reduction). After inspecting the cores and the height map the designer can choose one of the transformations, which are sorted according to the cost function.

The procedure *computeTransformations* is presented in Algorithm 9. First, trans-

Algorithm 8 Resolution of encoding conflicts

```
conflictResolution(Γ, resolutionType)
{
   Γ′ = Γ
   repeat
      Pref_Γ= constructPrefix(Γ′) //construction of finite complete prefix by PUNF
      CS = getConflictPairs(Pref_Γ) //detection of conflict sets by VERISAT
      cores = computeCores(CS) //extract cores from conflict sets
      if (cores ≠ ∅)
      {
         selectedTransf = ∅
         if (resolutionType is "manual")
         {
            view core/height map
            selectedTransf ⟸ "select a transformation"
            if (selectedTransf is "signal insertion")
               insert_ph2(Pref_Γ, selectedTransf) //perform phase two
            transformSTG(selectedTransf, Γ′)
         }
         else //semi-automated resolution
         {
            transformations = ∅
            computeTransforations(Pref_Γ,cores, transformations) //set possible transformations
            view core/height map and transformations
            if (transformations ≠ ∅)
            {
               selectedTransf ⟸ "select a transformation"
               transformSTG(selectedTransf, Γ′)
            }
            else //no transformation found
               cores = ∅ //terminate resolution process
         }
      }
   until cores = ∅
}
```

Algorithm 9 Computation of possible transformations

```
computeTransformations(Pref_Γ, cores, transformations)
{
   transformations_ph1 = ∅
   setTransformation_ph1(Pref_Γ, cores, transformations, transformations_ph1 )
   forall ((τ ∈ transformation_ph1) and (τ is signal insertion))
   {
      Pref′_Γ = Pref_Γ
      insert τ in Pref′_Γ
      fade out ∀x ∈ B ∪ E,τ#x and τ||x //set constraints for insertion
      update height map
      setTransforation_ph2(Pref′_Γ, cores, transformations)
   }
   sort(transformations) //sort transformations according cost function
}
```

formations for phase one are determined by the procedure *setTransformation_ph1* (see Algorithm 10), which include concurrency reductions and the first phase of signal insertions. For the latter the second phase is computed for every insertion τ in the phase one. This is done by inserting τ in the prefix $Pref_\Gamma$, resulting in a new prefix $Pref'_\Gamma$, by fading out nodes which are concurrent with or in structural conflict with τ and by updating the height map. Then, the procedure *setTransformation_ph2* computes the possible transformation for phase two similar to *setTransformation_ph1* but without considering concurrency reduction and by using the updated height map.

The computation of possible transformations in phase one is described in detail in Algorithm 10. The events corresponding to the highest peaks in the height map are set to E_{peak}. For every set of events Π which belong to the same set of cores transformations are determined. In the case no transformations are found the procedure is repeated by expanding the transformation area and taking into account events with the next highest peaks. If still no transformations are found after considering all events belonging to cores the procedure is terminated.

The transformations are determined as follows. First, signal insertion in phase one is computed. For each event $f \in \Pi$ transition splitting is determined by the functions *aSI* and *bSI*. The former checks if it is possible to insert before the event $f \rightarrow\!\wr h(f)$ and the latter checks if it is possible to insert after f $h(f)\wr\!\rightarrow$. The function *CI* checks if a concurrent insertion is possible and if it is possible it returns an event $g \in E$ then a concurrent insertion $h(g) \stackrel{n}{\longrightarrow} h(f)$ can be added where n is set by *calcN*. Then, possible concurrency reductions are determined by the functions *fCR* and *bCR*. If they exist, the former finds forward reduction $h(E_u) \stackrel{n}{\dashrightarrow} h(f)$ after determining E_u and n, and the latter finds backward reduction $h(E_u) \stackrel{n}{\dashrightarrow} h(i)$ if i is returned by *bCR*, where $i \in E$, and after determining E_u and n. Note that f is in the core and in order to determine a backward concurrency reduction an event i must exist which is concurrent with f and outside the considered cores, such that $f \in E_u$. The calculation of the existence of a marking in the introduced place is determined by *calcN(E_u, e)*, where $e \in E$. It returns 0 if all u-labelled events in the prefix do not contain e-labelled events in their configurations, otherwise it returns 1.

Algorithm 10 Computation of valid transformations (phase one)

```
setTransforation_ph1(Pref_Γ, cores, transformations, transformations_ph1)
{
   peak = max(|cores(e)|), e ∈ E //highest peak
   E_peak = ⋃ e ∈ E, |cores(e)| = peak
   repeat
      forall (e ∈ E_peak,"e not tagged as seen")
      {
         Π = ∅
         forall (f ∈ E_peak,"f not tagged as seen")
            if (cores(e) = cores(f)) //e and f belong to the same set of cores
            {
               tag f as "seen"
               Π = Π ∪ f //events which belong to the same set of cores
               //signal insertion phase one
               if (aSI(f)) //insert after f
                  transformations_ph1 ⟸ "h(f)⇝"
               if (bSI(f)) //insert before f
                  transformations_ph1 ⟸ "⇝h(f)"
               if (((g = CI(f, Π)) ∈ E) //concurrent insertion
               {
                  n = calcN(g, f)
                  transformations_ph1 ⟸ "h(g) ⟿^n h(f) "
               }
               //concurrency reduction
               if (fCR(f, Π))//forward concurrency reduction
               {
                  E_U = calcE_U(f, Π)
                  n = calcN(E_U, f)
                  transformations ⟸ "h(E_u) -⇢^n h(f)"
               }
               if ((i = bCR(f, Π)) ∉ Π) //backward concurrency reduction
               {
                  E_U = calcE_U(f, i, Π)
                  n = calcN(E_U, i)
                  transformations ⟸ "h(E_u) -⇢^n h(i)"
               }
            }
      }
      if (transformations_ph1 = ∅ or transformations = ∅ ) //no transformations exist
      {
         E_peak = ⋃ e ∈ E, |cores(e)| ≥ peak − 1 //include the next highest peak
         peak = peak − 1
         reset "seen" e ∈ E
      }
   until "transformations exist" and peak > 0
}
```

5.6 Conclusion

A framework for interactive refinement aimed at resolution of encoding conflicts in STGs has been presented. It is based on the visualisation of conflict cores, which are sets of transitions "causing" encoding conflicts. Cores are represented at the level of STG unfolding prefix, which is a convenient model for the understanding of the behaviour of the system due to its simple branching structure and acyclicity. Several types of encoding conflicts have been considered, among these the CSC conflicts, whose elimination is necessary for implementation, which is reflected in the quality of the refinement.

The advantage of using cores is that only those parts of STGs which cause encoding conflicts, rather than the complete list of CSC conflicts, are considered. Since the number of cores is usually much smaller than the number of encoding conflicts, this approach saves the designer from analysing large amounts of information. Resolution of encoding conflicts requires the elimination of cores by either signal insertion or concurrency reduction. The refinement contains several interactive steps aimed at helping the designer to obtain a customised solution.

Heuristics for transformations based on the height map and exploiting the intersections of cores use the most essential information about encoding conflicts, and thus should be quite efficient. In the developed tool manual or semi-automated resolution process is employed in order to obtain an optimal solution. The tool suggests the areas for transformations based on relatively simple cost functions, which are used as guidelines. Yet, the designer is free to intervene at any stage and choose an alternative location, in order to take into account the design constraints. These cost functions are based on specification level optimisation, and are very approximate, but computationally inexpensive. However, they do not take into account gate and physical level optimisations. For example, contextual signals are not taken into account, and physical parameters such as interconnect area, transistor sizes and parasitics are also not taken into account.

Chapter 6

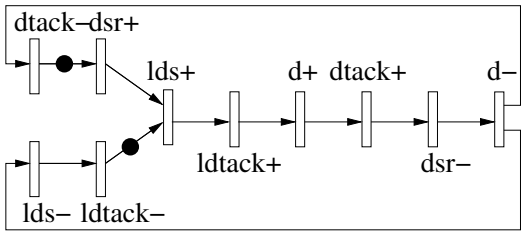
Interactive synthesis

In this chapter several interesting examples are discussed to demonstrate the proposed interactive resolution of encoding conflicts based on core visualisation. First, the VME-bus controller example introduced in section 3.4 is re-examined, and alternative solutions for CSC conflicts are presented involving not only signal insertion but also flip flop insertion and concurrency reduction. Then, a range of real life design cases are presented. They show that the combination of the visualisation approach and the intellectual effort of a human designer achieves better solutions than automated ones. After understanding the problem by applying the proposed visualisation approach, the designer is able to use their experience and intuition to produce optimal solutions. In addition, the human perspective allows the individual interpretation of specifications, for example whether they involve regular or familiar structures, and allows resolution on a case-by-case basis.

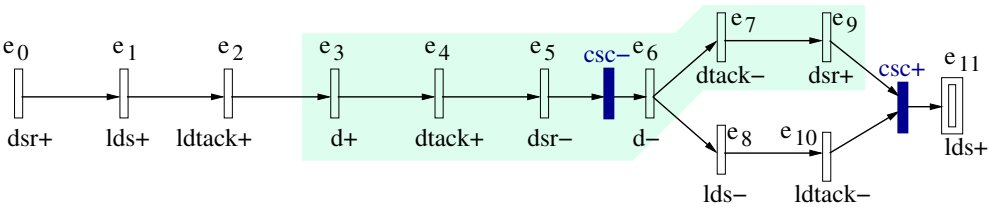
6.1 VME-bus controller

The STG of the read cycle of the VME-bus controller introduced in section 3.4 is presented in Figure 6.1(a). This example is re-examined, and the interactive resolution process based on core visualisation is applied, showing the proposed encoding conflict resolution methods.

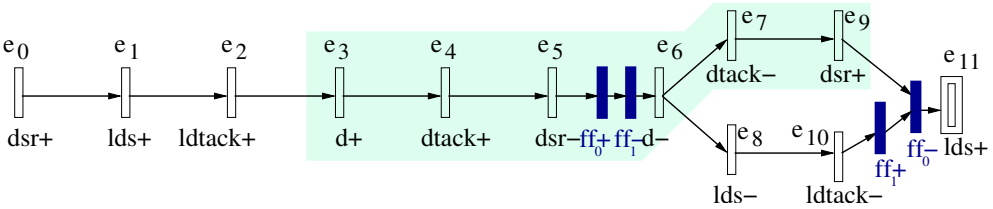
The STG has a CSC conflict resulting in a core shown in Figure 6.1(b). The core can



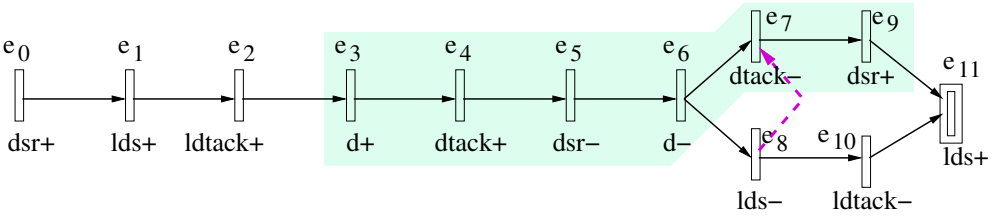
(a) initial STG



(b) signal insertion $\rightarrow\wr d^-$ and $\rightarrow\wr lds^+$



(c) flip flop insertion $[ff_0^+, ff_1^-]$, $\rightarrow\wr d^-$ and $\rightarrow\wr lds^+$



(d) forward concurrency reduction $lds^- \dashrightarrow dtack^-$

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

Figure 6.1: VME-bus controller: read cycle

be eliminated by signal insertion. For example, an auxiliary signal *csc* can be inserted such that csc^- is added in the core $\rightarrow\!\wr d^-$ and csc^+ is added outside the core $\rightarrow\!\wr lds^+$. Other single signal insertions are also possible and are presented in Table 6.1, where the column 'lits' shows the total number of literals in the corresponding equations. Solution 1 is one of the best solutions in terms of the cost function if $\alpha_1 - \alpha_3$ are equal to one. It delays two transitions, d^- and lds^+, and it reduces one trigger for the signal *lds*. Other transformations, e.g. solution 4, delays only one transition, lds^+, while introducing one additional signal to the triggers of *d*. Solution 12, on the other hand, does not delay any transitions but it introduces one additional trigger to the signals *d* and *lds*. In all three transformations the *csc* signal itself has two triggers. The equations for these transformations are shown in Figure 6.2(a)-(c); the complexity of the estimated logic is reflected in the equations. Solution 1 has the lowest complexity, followed by solution 4. Solution 12 has the highest complexity. However, not only do the additional triggers for the signals *d* and *lds* cause the complexity, but the increased concurrency also affects the complexity because of the expanded reachable state, which allows less room for Boolean minimisation. Note that the cost function does not take context signals into account, which can also contribute to the complexity. Contextual signals are computed during logic derivation, however, if CSC conflicts exist the next-state functions of some signals are ill defined, and thus these signals are not implementable as logic gates before the CSC conflicts are resolved.

Flip flop insertion can also be applied to eliminate the core. In Figure 6.1(c) the pairs of complementary auxiliary signals, $[ff_0^+, ff_1^-]$, are introduced such that $ff_0^+ \twoheadrightarrow ff_1^-$ are inserted in the core $\rightarrow\!\wr d^-$ and ff_0^- is inserted $\rightarrow\!\wr lds^+$ while ff_1^+ is inserted $ldtack^-\wr\!\rightarrow$. This results in a NAND flip flop, which is set by *dsr* and reset by *ldtack*. Note that a simple flip flop is produced since ff_0 is only triggered by *dsr*. The equations for this transformation is presented in Figure 6.2(d), which can be built out of simple gates.

The existence of a fork transition in the core, and the fact that events exist which are concurrent to the core, means that concurrency reduction can also be undertaken. In this case the concurrent events e_8 and e_{10} can be "dragged" into the core and thus disambiguate the encoding conflict by their corresponding binary values. Since the

#	*cost* : $\Delta\omega, \Delta logic, \Delta cores$	transformation phase 1	transformation phase 2	lits
1	2: 2,1,1	→≀e6,	→≀e11	8
2	2: 2,1,1	→≀e6,	e10≀→	9
3	2: 1,2,1	e4→→e6,	e10≀→	11
4	2: 1,2,1	e3→→e6,	e10≀→	11
5	2: 1,2,1	e5→→e7,	→≀e11	12
6	3: 1,3,1	e4→→e6,	→≀e11	11
7	3: 1,3,1	e3→→e6,	→≀e11	11
8	3: 2,2,1	→≀e4,	e10≀→	12
9	3: 1,3,1	→≀e6,	e8→→e11	12
10	3: 3,1,1	e6≀→,	e10≀→	14
11	3: 0,4,1	e4→→e6,	e8→→e11	14
12	3: 0,4,1	e3→→e6,	e8→→e11	14
13	3: 1,3,1	e4→→e7,	→≀e11	14
14	3: 1,3,1	e3→→e7,	→≀e11	14
15	3: 2,2,1	e6≀→,	e8→→e11	16
16	4: 3,2,1	e6≀→,	→≀e11	11
17	4: 2,3,1	→≀e4,	→≀e11	12
18	4: 2,3,1	→≀e7,	→≀e11	13
19	4: 1,4,1	→≀e4,	e8→→e11	15

Table 6.1: Possible single signal insertions

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

(a) solution 1

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

(b) solution 4

$$\begin{aligned} 0 &= lds \cdot \overline{csc} \\ d &= dsr \cdot d + 0 \cdot ldtack \\ 2 &= \overline{lds} \cdot (\overline{dsr} + ldtack) + csc \\ lds &= d + \overline{2} \\ dtack &= d \\ csc &= lds \cdot (d + csc) \end{aligned}$$

(c) solution 12

$$\begin{aligned} d &= ff1 \cdot ldtack \\ lds &= \overline{ff0} + d \\ dtack &= d \\ ff0 &= \overline{ff1} + \overline{dsr} \\ ff1 &= \overline{ldtack} + \overline{ff0} \end{aligned}$$

(d) flip flop insertion $[ff_0^+, ff_1^-]$

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

(e) $lds^- \dashrightarrow dtack^-$

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

(f) $ldtack^- \dashrightarrow dtack^-$

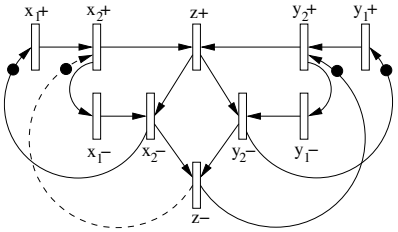
Figure 6.2: Selected equations

signal *dsr* corresponds to an input, only a causal constraint can be added to the output signal *dtack*. Two forward concurrency reductions are possible, $lds^- \dashrightarrow dtack^-$ and $ldtack^- \dashrightarrow dtack^-$. The former delays *dtack* by *lds* whereas the latter delays *dtack* by *lds* and *ldtack*. Note that *ldtack* is an input signal and thus might take a long time to complete. The former reduction allows the resetting of the input *ldtack* concurrently with the output *dtack* and the input *dsr*, resulting in a faster circuit. The latter reduces the concurrency completely. The equations are shown in Figure 6.2(e) and (f). Note that due to existing concurrency in the first reduction its equations are more complex than the second reduction.

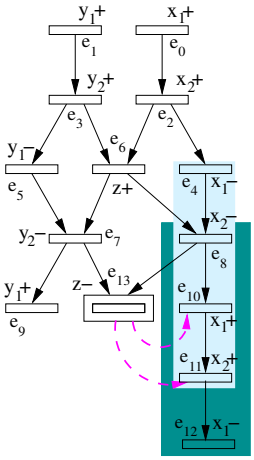
After considering several alternative resolutions, the designer is able to decide which solution is best suitable according to the design constraints. For example, the flip flop solution could be chosen because it consists of simple gates. In terms of area and latency it is similar to those of solution 1 or the concurrency reduction $lds^- \dashrightarrow dtack^-$. The flip flop insertion has a worst case delay of four transitions between adjacent input transitions compared to three transition delays for the other two. However, the usage of simple gates compensates for this. Solutions 4 and 12 have only two transition delays between adjacent input transitions. Despite the complex logic they have a lower latency than the other three mentioned above.

6.2 Weakly synchronised pipelines

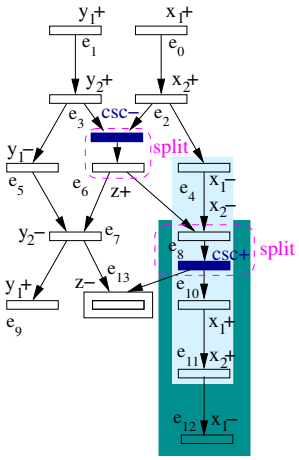
Figure 6.3(a) shows an STG modelling two weakly synchronised pipelines without arbitration [40]. The STG exhibits encoding conflicts resulting in two cores shown in Figure 6.3(b) and (c) respectively. In Figure 6.3(b) two possible concurrency reductions resolving the CSC conflicts are shown. Both cores can be eliminated by forward concurrency reduction because z^- is concurrent to the core overlap, where concurrency starts. Thus "dragging" z^- into the core overlap would eliminate the cores, because z is used to disambiguate the conflicts. A causal constraint, either $z^- \overset{1}{\dashrightarrow} x_1^+$ or $z^- \overset{1}{\dashrightarrow} x_2^+$, can be added. However, the first reduction delays x_1^+ and adds z to the triggers of x_1, whereas the second reduction has no effect on the delay (z^- can be executed concurrently with



(a) initial STG



(b) concurrency reduction



(c) signal insertion

$$
\begin{aligned}
x_1 &= \overline{x_2} \\
x_2 &= \overline{z} \cdot (x_1 + x_2) + x_1 \cdot x_2 \\
y_1 &= \overline{y_2} \\
y_2 &= y_1 \cdot (y_2 + \overline{z}) + y_2 \cdot \overline{z} \\
z &= y_2 \cdot (z + x_2) + x_2 \cdot z
\end{aligned}
$$

$$
\begin{aligned}
x_1 &= \overline{x_2} \cdot \mathrm{csc} \\
x_2 &= x_2 \cdot (\overline{z} + \mathrm{csc}) + x_1 \\
y_1 &= \overline{y_2} \\
y_2 &= y_1 \cdot (y_2 + \overline{z}) + y_2 \cdot \overline{z} \\
z &= z \cdot y_2 + \overline{\mathrm{csc}} \\
\mathrm{csc} &= \mathrm{csc} \cdot (\overline{y_2} + z) + \overline{x_2}
\end{aligned}
$$

$$
\begin{aligned}
x_1 &= \overline{x_2} \cdot \mathrm{csc} \\
x_2 &= x_2 \cdot (\overline{z} + \mathrm{csc}) + x_1 \\
y_1 &= \overline{y_2} \\
y_2 &= y_1 \cdot (y_2 + \overline{z}) + y_2 \cdot \overline{z} \\
z &= z \cdot y_2 + x_2 \cdot \overline{\mathrm{csc}} \\
\mathrm{csc} &= \mathrm{csc} \cdot (\overline{y_2} + z) + \overline{x_2}
\end{aligned}
$$

$z^- \xrightarrow{1} x_2^+$

$h(e_8) \wr\!\!\rightarrow$ and $\rightarrow\!\wr h(e_6)$

$\rightarrow\!\wr h(e_{10})$ and $\rightarrow\!\wr h(e_6)$

(d) equations

outputs: x_1, x_2, y_1, y_2, z; **internal:** csc

Figure 6.3: Weakly synchronised pipelines

its predecessor) or on the number of triggers of x_2 (as z^+ already triggers x_2^-). Thus the second reduction is preferable according to the cost function, resulting in the STG shown in Figure 6.3(a), with the dashed arc. The corresponding equations are presented in Figure 6.3(d).

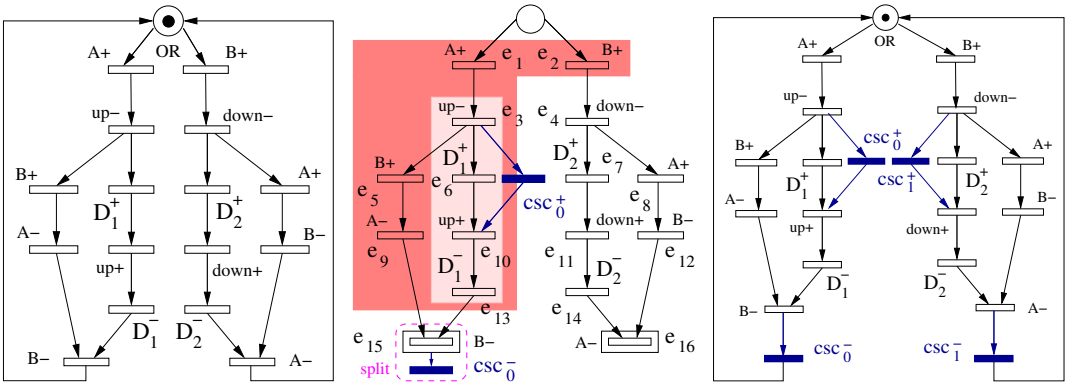
The cores can also be eliminated by an auxiliary signal *csc*. Phase one of the resolution process inserts a signal transition somewhere into the highest peak in the height map, which comprises the events e_8, e_{10} and e_{11}. For example, in Figure 6.3(c) a signal transition csc^+ is inserted $h(e_8)\wr\rightarrow$ and its counterpart is inserted outside the cores $\rightarrow\wr h(e_6)$, ensuring that the cores are eliminated. Other valid insertions are possible, e.g. inserting $csc^+ \rightarrow\wr h(e_{10})$ and its counterpart $\rightarrow\wr h(e_6)$. Both these transformations eliminate all the cores, and in both of them the newly inserted signal has two triggers, but the former insertion delays three transitions, adds the trigger *csc* to x_1 and replaces the trigger x_2 of z with *csc*, whereas the latter insertion delays two transitions and adds the trigger *csc* to x_1 and z. The equations corresponding to these two solutions are shown in Figure 6.3(d).

One can see that the implementations derived by signal insertion are more complex than the one obtained by concurrency reduction. The estimated area for the former is 19 literals for the transformation $h(e_8)\wr\rightarrow$ and $\rightarrow\wr h(e_6)$, and 20 literals for $\rightarrow\wr h(e_{10})$ and $\rightarrow\wr h(e_6)$, and for the concurrency reduction is 17 literals. Note that the equations for the signal insertion are equal except for the signal z, which in the second transformation has an additional trigger (x_2), compared to the triggers (*csc* and y_2) in the first transformation. These two implementations also delay signals z and x_1, whereas the one derived using concurrency reduction does not have additional delays. Additionally, the solution obtained by concurrency reduction results in a symmetrical STG.

6.3 Phase comparator

An STG model of a device which detects the difference between the phases of two input signals, A and B, and activates a handshake, (up, D_1) or $(down, D_2)$, that corresponds to the direction of the difference is shown in Figure 6.4(a). Its outputs can control an

up/down counter. Such a device can be used as a part of a phase locked loop circuit [20]. The initial STG does not satisfy the CSC property, and PETRIFY fails to solve the CSC problem reporting irreducible CSC conflicts. However, the method based on core visualisation allows to solve the conflicts manually.



(a) initial STG (b) unfolding prefix with cores in one of the branches (c) final STG

inputs: A, B, D_1, D_2; **outputs:** $up, down$; **internal:** csc_0, csc_1

Figure 6.4: Phase comparator

The encoding conflicts are caused by four CSC cores. Note that the model is symmetrical, and for simplicity only the cores in the "up" branch are depicted in Figure 6.4(b). The events e_3, e_6, e_{10} and e_{13} comprise the highest peak since each of them belongs to both depicted cores, and insertion of an additional signal transition, csc_0^+, eliminates these cores. In order to reduce the latency, csc_0^+ is inserted concurrently to e_6, $h(e_3) \longrightarrow\!\!\!\!\!\rightarrow h(e_{10})$. Note that e_{13} corresponds to an input signal and thus the newly inserted signal cannot "delay" it. Similarly, csc_1^+ is inserted concurrently to e_7, $h(e_4) \longrightarrow\!\!\!\!\!\rightarrow h(e_{11})$. The corresponding falling transitions, csc_0^- and csc_1^-, should be inserted somewhere outside the cores. In order to preserve the consistency they cannot be inserted concurrently to or in structural conflict with, respectively, csc_0^+ and csc_1^+. Yet, this cannot be done without "delaying" input events! That is the reason why PETRIFY reports irreducible CSC conflicts.

At this point the intervention of the designer is crucial. Having analysed the information provided by this method it is decided to insert csc_0^- $B^- \wr\!\!\rightarrow$ and csc_1^- $A^- \wr\!\!\rightarrow$. The resulting STG is shown in Figure 6.4(c) has the CSC property. However, the performed

transformation has changed the original input/output interface. Yet, it is often reasonable to assume that the environment acts slower than the gates implementing these two inserted signals (such assumption can be later enforced by transistor sizing etc.), and by introducing this *timing assumption* [17], the original interface can be retained.

This example clearly shows how the proposed interactive method can be useful in designs which originate from real-life specifications. In such designs, there is often no possibility to progress by relying only on automatic tools like PETRIFY because of a combination of obstacles such as irreducible CSC conflicts, potential need of changing the input/output interface and/or introduction of timing assumptions. The proposed visualisation method enables the designer to understand the cause of such irreducible conflicts and make an appropriate decision.

6.4 AD converter

This example comes from the design of an asynchronous AD converter described in [17, 45]. The structure of the converter is shown in Figure 6.5. 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.

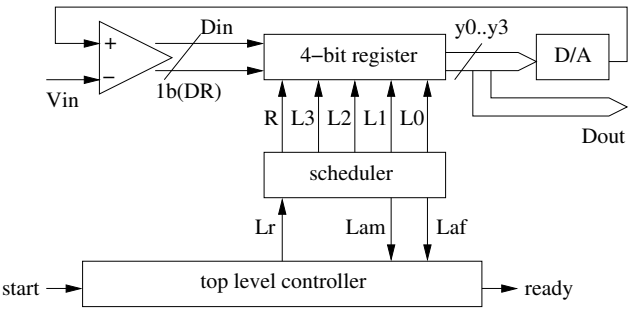


Figure 6.5: Block diagram of the AD converter

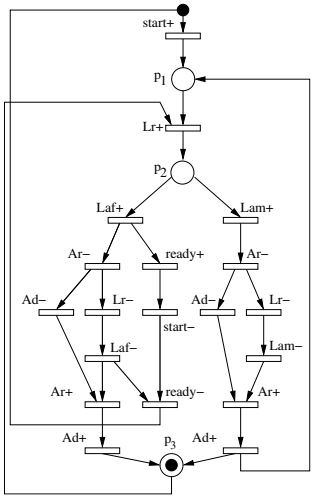
The 4-bit register is effectively a combination of a demultiplexor and a register with serial dual-rail input and parallel single-rail output. It is built from transparent latches. The register is controlled by five command signals arriving from the control logic in a one-hot (non-overlapping) manner. 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.

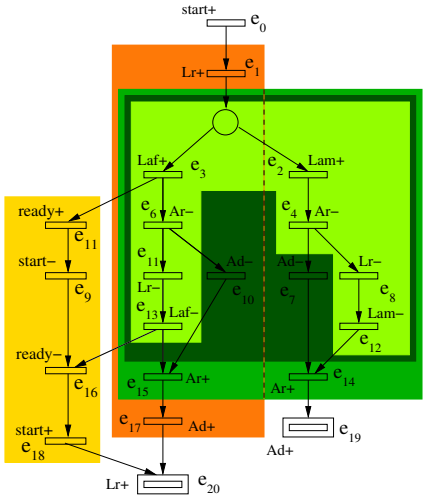
6.4.1 Top level controller

The control logic consists of a five-way scheduler and a top level controller. The latter is described in Figure 6.6(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 (load 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.

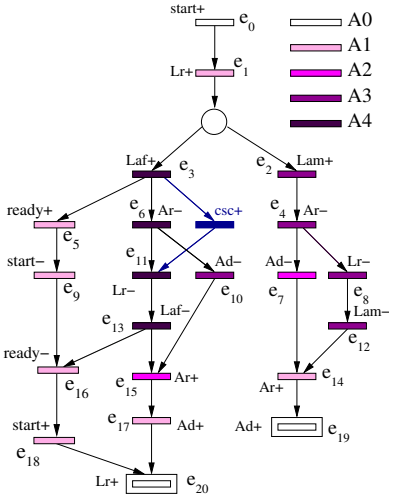
The STG in Figure 6.6(a) has several CSC conflicts. It contains five type I and three type II CSC conflict pairs corresponding to two type I and three type II cores, shown in Figure 6.6(b). The events e_3, e_6, e_{11} and e_{13} comprise the highest peak, as each of them belongs to four cores. They can 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. Valid concurrency reductions are presented in the table in Figure 6.7(a), where the column 'lits' shows the total number of literals in the corresponding equations. The first four solutions eliminate all the cores in the peak, and the last solution 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), whereas the second one does not delay e_{11} but introduces an additional trigger. The equations for these two solutions are shown in Figure 6.7(c). The third solution delays e_6 by e_5, and the fourth solution delays e_6 by e_5 and e_9;



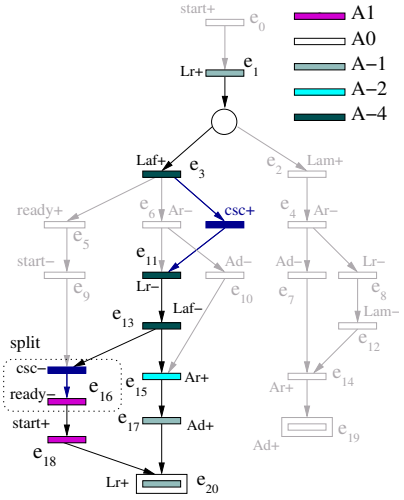
(a) initial STG



(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 6.6: STG transformation of the top level controller

moreover, both these solutions introduce an additional trigger to Ar (which already had three triggers), and thus are inferior according to the cost function.

Alternatively, the encoding conflicts can be solved using signal insertion, by inserting a transition csc^+ into the peak and into its counterpart outside the cores belonging to the peak, preserving the consistency and ensuring that the cores are destroyed. As previously stated, the input signal transitions cannot be delayed by newly inserted transitions, i.e. in the peak csc^+, cannot delay e_3 and e_{13}. In the second phase, 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 csc^- is inserted there. At the same time, the elimination of the remaining core $\{e_5, e_9, e_{16}, e_{18}\}$ can be attempted. The valid signal insertions are shown in the table in Figure 6.7(b). The solution 6 is illustrated in Figure 6.6(c), where in phase 1 csc^+ is inserted concurrently to Ar^-, $Laf^+ \longrightarrow\!\!\!\!\!\rightarrow Lr^-$, and csc^- is inserted sequentially, $\rightarrow\!\wr ready^-$.

Solution 6 introduces the smallest delay (only $ready^-$ is delayed), whereas solution 7 has the smallest estimated logic complexity, but the largest delay (the insertion delays $ready^+$, Ar^- and $ready^-$). Solutions 9 and 11 have the greatest estimated logic complexity. The equations for solution 6, 7 and 9 are presented in Figure 6.7(c). The equations for solution 7 and 9 have the same number of literals, even though their estimated logic complexities are quite different. This shows that the cost function is not perfect, since $\mathfrak{C}(n)$ is quite a rough estimate of complexity, and since the cost function does not take the context signals into account. However, it is not trivial to significantly improve this cost function without introducing a considerable time overhead in computing it. In particular, the context signals cannot be computed for a particular signal z until all the encoding conflicts for z are resolved.

The solution 2 does not introduce any additional delay and thus is the best in terms of latency. It has a worst case delay of two transitions between adjacent input transitions compared with at least three transition delays for the other solutions.

concurrency reduction			
#	causal constraint	$\Delta\omega;\Delta logic;\Delta cores$	lits
1	$h(e_9)$ --➝ $h(e_{11})$	3;-1;-5	11
2	$h(e_5)$ --➝ $h(e_{11})$	0;2;-5	14
3	$h(e_5)$ --➝ $h(e_6)$	1;2;-5	14
4	$h(e_9)$ --➝ $h(e_6)$	4;2;-5	11
5	$h(e_{10})$ --➝ $h(e_{11})$	3;0;-1	n/a

(a) possible concurrency reductions

signal insertion				
#	phase 1	phase 2	$\Delta\omega;\Delta logic;\Delta cores$	lits
6	$h(e_3) \rightarrow\!\!\!\!\rightarrow h(e_{11})$	$\rightarrow\!\!\wr h(e_{16})$	1;3;-5	16
7	$h(e_3)\wr\!\!\rightarrow$	$\rightarrow\!\!\wr h(e_{16})$	3;2;-5	15
8	$\rightarrow\!\!\wr h(e_6)$	$\rightarrow\!\!\wr h(e_{16})$	2;3;-5	16
9	$\rightarrow\!\!\wr h(e_{11})$	$\rightarrow\!\!\wr h(e_{16})$	2;4;-5	15
10	$h(e_3)\wr\!\!\rightarrow$	$h(e_9\wr\!\!\rightarrow)$	3;3;-5	18
11	$h(e_3)\wr\!\!\rightarrow$	$h(e_5) \rightarrow\!\!\!\!\rightarrow h(e_{16})$	2;4;-5	20

(b) possible signal insertions

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

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

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

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

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

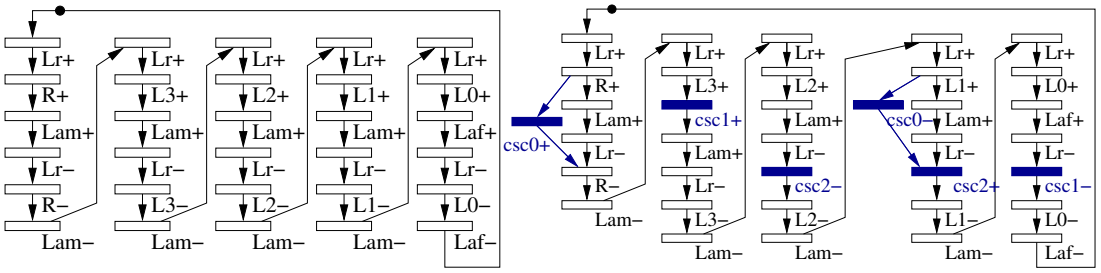
(c) a selection of equations

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

Figure 6.7: Valid transformations for top level controller

6.4.2 Scheduler

The STG model of the scheduler is shown in Figure 6.8(a). It consists of five identical fragments. Note the similarity with the STG model of a modulo-N counter, which operates as a frequency divider between Lr and Laf. It produces an acknowledgement on the Lam output N-1 times followed by one acknowledgement on Laf.



(a) initial STG (b) STG derived by Petrify

inputs: Lr; **outputs:** $Lam, Laf, L0, L1, L2, L3, R$; **internal:** csc_0, csc_1, csc_2

Figure 6.8: STG transformation of the scheduler

This STG has several CSC conflicts and it requires internal memory to record the position of the token in this counter. The solution obtained automatically by Petrify gives the STG depicted in Figure 6.8(b). This STG leads to the circuit whose gates are described by the equations in Figure 6.9(a). The logic is clearly irregular. Some *csc* signals are inserted in series with the output signals and, as a result, the circuit has, in the worst case, three signal events between two adjacent input events.

The coding conflicts are caused by four CSC cores as shown in Figure 6.10(a). The elimination of all cores does not always guarantee the solution of all CSC conflicts. If conflict sets are formed from different cores, and these cores are eliminated by signals of different polarity, the conflict sets formed by those cores might not be eliminated. For example, a conflict set $\mathcal{CS}$ is formed by core 1 and 2. Destroying core 1 by csc^+ and core 2 by csc^- would result in that $\mathcal{CS}$ becoming a core in the next step of the resolution process. This can happen if the number of *csc* signal is minimised.

The first attempt to resolve the conflicts is a straightforward one without any consideration of regularity in the specification. Two cores can be eliminated by one *csc*

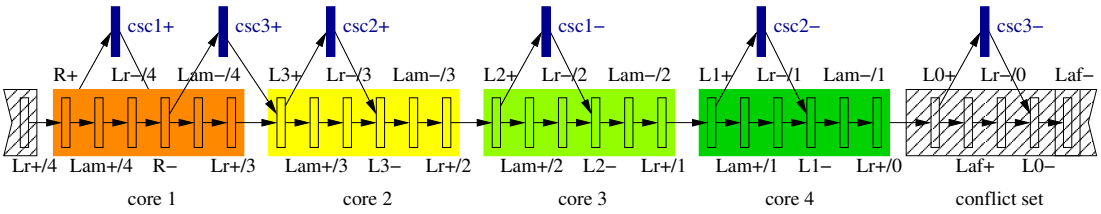
$$
\begin{aligned}
\text{Lam} &= \text{L3}\cdot\text{csc1} + \text{L1} + \text{L2} + \text{R}\\
\text{Laf} &= \text{L0}\\
\text{L0} &= \text{csc1}\cdot(\text{Lr}\cdot\overline{\text{csc0}}\cdot\text{csc2} + \text{Laf})\\
\text{L1} &= \overline{\text{csc2}}\cdot(\text{L1} + \text{Lr})\\
\text{L2} &= \text{csc2}\cdot(\text{Lr}\cdot\overline{\text{L3}}\cdot\text{csc1}\cdot\text{csc0} + \text{L2})\\
\text{L3} &= \text{Lr}\cdot(\overline{\text{csc1}}\cdot\overline{\text{R}}\cdot\text{csc0} + \text{L3})\\
\text{R} &= \overline{\text{csc0}}\cdot(\text{Lr}\cdot\overline{\text{csc1}} + \text{R}) + \text{Lr}\cdot\text{R}\\
\text{csc0} &= \text{csc0}\cdot\overline{\text{L1}} + \text{R}\\
\text{csc1} &= \text{csc1}\cdot(\overline{\text{Laf}} + \text{Lr}) + \text{L3}\\
\text{csc2} &= \text{csc2}\cdot(\overline{\text{L2}} + \text{Lr}) + \overline{\text{Lr}}\cdot\overline{\text{csc0}}
\end{aligned}
$$

(a) solution by PETRIFY

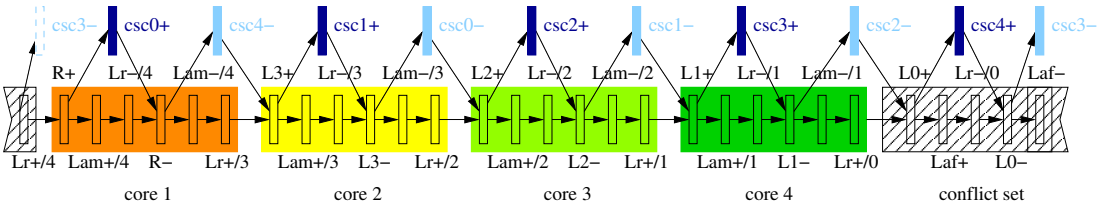
$$
\begin{aligned}
\text{Lam} &= \text{L1} + \text{L2} + \text{L3} + \text{R}\\
\text{Laf} &= \text{L0}\\
\text{L0} &= \text{Lr}\cdot\overline{\text{csc2}}\cdot\text{csc3}\cdot\ [\overline{\text{csc4}}]{+}[L0\cdot(\overline{csc4} + Lr)]\\
\text{L1} &= \text{Lr}\cdot\overline{\text{csc1}}\cdot\text{csc2}\cdot\ [\overline{\text{csc3}}]{+}[L1\cdot(\overline{csc3} + Lr)]\\
\text{L2} &= \text{Lr}\cdot\overline{\text{csc0}}\cdot\text{csc1}\cdot\ [\overline{\text{csc2}}]{+}[L2\cdot(\overline{csc2} + Lr)]\\
\text{L3} &= \text{Lr}\cdot\text{csc0}\cdot\overline{\text{csc4}}\cdot\ [\overline{\text{csc1}}]{+}[L3\cdot(\overline{csc1} + Lr)]\\
\text{R} &= \text{Lr}\cdot\overline{\text{csc3}}\cdot\text{csc4}\cdot\ [\overline{\text{csc0}}]{+}[R\cdot(\overline{csc0} + Lr)]\\
\text{csc0} &= \text{csc0}\cdot\overline{\text{csc1}} + \text{L3} + \text{R}\\
\text{csc1} &= \text{csc1}\cdot\overline{\text{csc2}} + \text{L2} + \text{L3}\\
\text{csc2} &= \text{csc2}\cdot\overline{\text{csc3}} + \text{L2} + \text{L1}\\
\text{csc3} &= \text{csc3}\cdot\overline{\text{csc4}} + \text{L0} + \text{L1}\\
\text{csc4} &= \text{csc4}\cdot\overline{\text{csc0}} + \text{L0} + \text{R}
\end{aligned}
$$

(b) manual solution

Figure 6.9: Logic equations



(a) a straightforward solution



(b) symmetric solution

inputs: Lr; **outputs:** $Lam, Laf, L0, L1, L2, L3, R$; **internal:** $csc_0 - csc_4$

Figure 6.10: Resolution attempts for the scheduler

signal and after the insertion a new core emerges, formed from the eliminated cores. By inserting *csc* signal transitions in such a way that they interleave the cores, the formation of new cores would be reduced. For example, core 1 could be eliminated by inserting a *csc* transition signal, say csc_1^+, concurrently $R^+ \longrightarrow\!\!\!\!\!\rightarrow R^-$. It is done in this way to maintain the regular structure. Since the acknowledgement signal *Lam* is used several times the insertion of csc_1^+ between its falling and rising transition would result in a relatively deep logic, given that it depends on four signals. Other combination of insertion in this core would also result in relatively deep logic.

The complement of csc_1^+ could be inserted in core 3, interleaving the cores, between the rising and falling transition signal $L2$. The cores 2 and 4 are eliminated in a similar way using csc_2. After inserting csc_1 and csc_2 not all conflicts are reduced. Another conflict core emerges, which is the combination of cores 1-4. Its complement is also a conflict set, see Figure 6.10(a), because the set of transitions in the STG is a conflict set. The new core can be eliminated, e.g. by inserting csc_3^+ concurrently $R^- \longrightarrow\!\!\!\!\!\rightarrow L3^+$ and csc_3^+ $L0^+ \longrightarrow\!\!\!\!\!\rightarrow L0^-$. It can be seen that the resulting STG offers a better solution to the one derived by PETRIFY, because all *csc* signals are inserted concurrently. However, it does not maintain the regular structure.

In the second attempt a conflict-free and regularly-structured STG is obtained by introducing more internal memory, i.e. more *csc* signals. Each core is eliminated by one *csc* signal. Inserting csc_i^+ into each core and their counterparts in such a way that they interleave the cores is illustrated in Figure 6.10(b). An additional *csc* signal is required to remove the new core which is formed by the conflict set, which is the complement of the one formed by core 1-4. Although five additional internal signals have been required the specification has a regular structure. This is reflected in the equations in Figure 6.9(b) which can be implemented as relatively compact and fast gates.

The added *csc* signal transitions are inserted concurrently with the input transitions. This produces only two output events between two adjacent input events. It can be assumed that the internal gates fire before the next event arrives on Lr. This leads to the following timing assumptions: $csc_0^+ < Lr^-/4$, $csc_3^- < Lr^+/3$, $csc_1^+ < Lr^-/3$, $csc_0^- < Lr^+/2$ ... using the relative timing approach described in [14]. As a result the

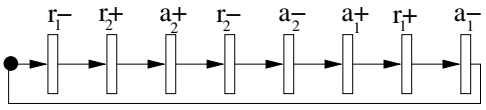
implementation is simplified because the terms in the squared brackets in the equations were made redundant.

6.5 D-element

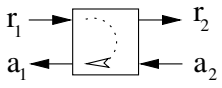
The STG of a handshake decoupling D-element [10, 105], is shown in Figure 6.11(a). It controls two handshakes, where one handshake initiates another. The first handshake waits for the other to complete. Then, the first handshake is completed, and the cycle is repeated.

The initial STG has a CSC conflict, resulting in a CSC core shown in Figure 6.11(c) and (e), respectively. The conflict occurs because the states before and after executing the handshake of the second handshake ($r_2^+ \rightarrow a_2^+ \rightarrow r_2^- \rightarrow a_2^-$) are equal. Two types of insertion are considered. In the first, single signal insertion is applied, where a new auxiliary signal is inserted, either sequentially or concurrently. In the second, flip flop insertion is examined, where two 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.

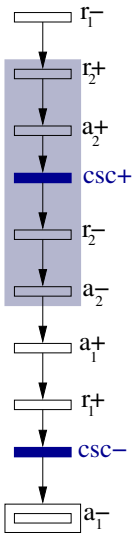
The first experiment introduces an auxiliary signal *csc*. The sequential insertion of *csc* is performed first. This is illustrated in Figure 6.11(c), where the positive transition csc^+ is inserted in the core $\rightarrow\!\wr r_2^-$ and the negative transition csc^- is inserted outside the core $\rightarrow\!\wr a_1^-$. Note that input signals cannot be delayed by *csc* and thus this is the only location to insert *csc* in series. This transformation introduces an inconsistent polarity of triggers to 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^-. This makes a_1 and *csc* neither p-normal nor n-normal. Similarly, reversing the polarity of *csc* to the transformation $csc^- \rightarrow\!\wr r_2^-$ and $csc^+ \rightarrow\!\wr a_1^-$ also introduces inconsistent triggers to signal r_2 and *csc*. The concurrent insertion adds csc^+ in the core between a_2^+, $r_2^+ \longrightarrow\!\!\!\!\rightarrow r_2^-$, and csc^- outside the core, between r_1^-, $a_1^+ \longrightarrow\!\!\!\!\rightarrow a_1^-$ (see Figure 6.11(c)). Note that there is no other location to insert *csc* concurrently. In this transformation the polarity of triggers for the signal a_1 and *csc* is also inconsistent. The reverse of the polarities



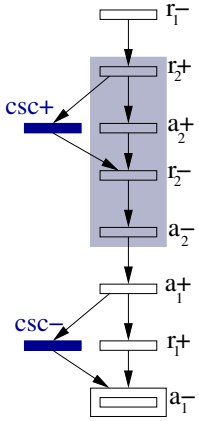
(a) STG



(b) symbol

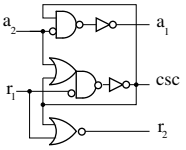


sequential

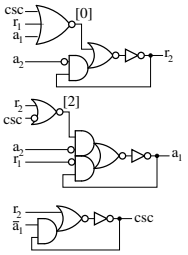


concurrent

(c) signal insertion

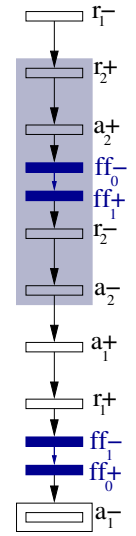


sequential

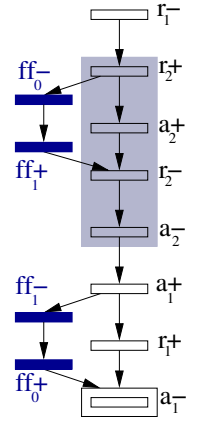


concurrent

(d) implementation for (c)

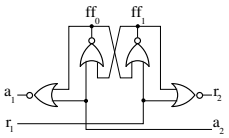


sequential

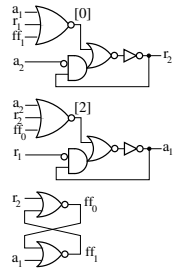


concurrent

(e) flip flop insertion



sequential



concurrent

(f) implementation for (e)

inputs: r_1, a_2; **outputs:** r_2, a_1; **internal:** csc, ff_0, ff_1

Figure 6.11: D-element

of the inserted signal *csc* also introduces inconsistency in the polarity of triggers of 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 and output inverters in the negative logic implementation of these solutions in Figure 6.11(d). The sequential solution consists of simpler logic, resulting in a smaller area. The estimated area for the sequential solution is 80 units and the estimated area for the concurrent solution is 160 units. The sequential solution has two transition delays between two adjacent input transitions, whereas the concurrent solution has only one transition delay. The maximum latency in the sequential solution occurs, e.g. between the input r_1^+ and the output a_1^-, giving the following sequence of switching gates and taking into account that CMOS logic is built out of negative gates: $[r_1 \uparrow] \rightarrow [\overline{r_1} \downarrow] \rightarrow [\overline{csc} \uparrow] \rightarrow [csc \downarrow] \rightarrow [\overline{a_1} \uparrow] \rightarrow [a_1 \downarrow]$. In the concurrent solution the maximum latency occurs, e.g between the input r_1^- and the output r_2^+, with the sequence of switching gates of $[r_1 \downarrow] \rightarrow [0 \uparrow] \rightarrow [\overline{r_2} \downarrow] \rightarrow [r_2 \uparrow]$. The sequential solution have an estimated latency of five negative gate delays and the concurrent solution have an estimated latency of three negative gate delays.

The second experiment realises a flip flop by introducing a pair of complementary signals. The flip flop insertion is shown in Figure 6.11(e), where $[ff_0^-, ff_1^+]$ are inserted realising a NOR flip flop. The sequential flip flop insertion is performed first. The core is eliminated by inserting $ff_0^- \twoheadrightarrow ff_1^+$ in the core $\rightarrow\!\wr r_2^-$ and $ff_1^- \twoheadrightarrow ff_0^+$ outside the core $\rightarrow\!\wr a_1^-$ realising an implementation shown in Figure 6.11(f). Note that this transformation produces an n-normal STG. This is reflected in the implementation, which is monotonic and consists of negative gates. The concurrent flip flop insertion adds $ff_0^- \twoheadrightarrow ff_1^+$ in the core between a_2^+, $r_2^+ \longrightarrow\!\!\!\!\!\rightarrow r_2^-$, and $ff_1^- \twoheadrightarrow ff_0^+$ outside the core between r_1^-, $a_1^+ \longrightarrow\!\!\!\!\!\rightarrow a_1^-$ (see Figure 6.11(e)). In this transformation the polarity of triggers is consistent; the transformed STG is not normal, however. For example, normalcy is violated for the signal r_2. Its triggers suggest that it is n-normal, but the normalcy conflict set $\{a_2^+, r_2^-, a_2^-\}$ contradicts that. A similar violation occurs for the signal a_1. Thus additional inverters are needed in the implementation (see Figure 6.11(f)). The areas estimated for the sequential and concurrent solutions are 96 units

and 176 units, 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 latency in the former occurs, e.g. between r_1^+ and ff_0^+ corresponding to the sequence of switching gates of $[r_1 \uparrow] \rightarrow [ff_1 \downarrow] \rightarrow [ff_0 \uparrow] \rightarrow [a_1 \downarrow]$. In the latter, maximum latency occurs, e.g. between r_1^- and r_2^+ corresponding to the sequence of switching gates of $[r_1 \downarrow] \rightarrow [0 \uparrow] \rightarrow [\overline{r_2} \downarrow] \rightarrow [r_2 \uparrow]$. In both cases the estimated latency is equal to three negative gate delays.

This example shows that the concurrent insertion, although not delaying output signals, has a latency comparable with 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.

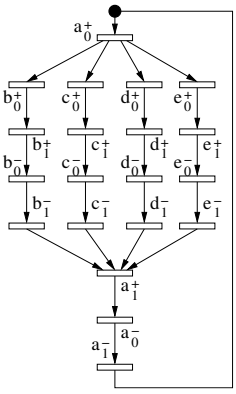
6.6 Handshake decoupling element

The specification of a handshake decoupling element is shown in Figure 6.12(a). The "parent" handshake at port a initiates four "child" handshakes at ports $b, \ldots, e$ and waits for them to complete. Then the parent handshake completes, and the cycle continues.

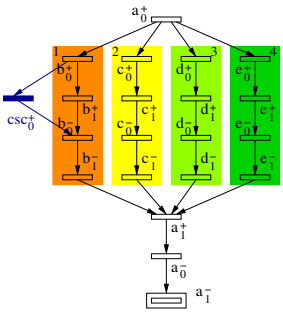
The four mutually concurrent handshakes result in 888 CSC conflict pairs — clearly too many for the designer to cope with. Yet, despite the huge number of encoding conflicts, it has only four CSC cores, as shown in Figure 6.12(b) (phase one).

In this case the height map is quite "plain" since the cores do not overlap and thus no event has an altitude greater than one. The cores are concurrent and can be eliminated independently by adding four new signals. The elimination of the first core is illustrated in Figure 6.12(b), where an auxiliary signal csc is inserted concurrently. In the core csc_0^+ is inserted concurrently with b_1^+, $b_0^+ \longrightarrow\!\!\!\!\rightarrow b_0^-$. Since its counterpart csc_0^- cannot be inserted concurrently to csc_0^+, parts of the prefix are faded out and csc_0^- is inserted into the remaining part, concurrently with a_0^-, $a_1^+ \longrightarrow\!\!\!\!\rightarrow a_1^-$.

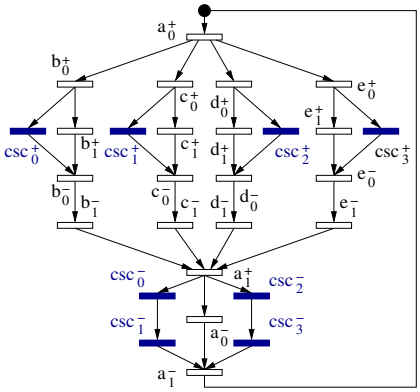
After transferring these transitions to the STG and unfolding the result, three cores



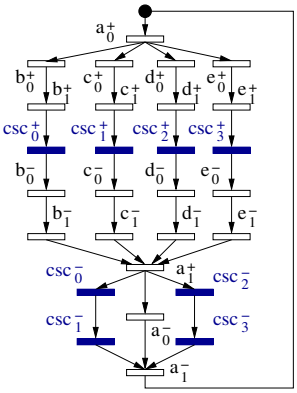
(a) initial STG



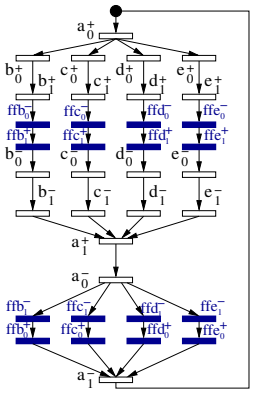
(b) resolution process (step 1)



(c) concurrent insertion



(d) sequential insertion



(e) flip flop insertion

$$
\begin{aligned}
a1 &= \overline{b1}\cdot\overline{c1}\cdot\overline{d1}\cdot\overline{e1}\cdot\overline{b0}\cdot\overline{c0}\cdot\overline{d0}\cdot\\
&\quad\ \overline{e0}\cdot csc0\cdot csc1\cdot csc2\cdot csc3\\
&\quad +a1\cdot(csc1+csc3+a0)\\
b0 &= a0\cdot\overline{a1}\cdot\overline{csc0}+\overline{b1}\cdot b0\\
c0 &= a0\cdot\overline{a1}\cdot\overline{csc1}+c0\cdot\overline{c1}\\
d0 &= a0\cdot\overline{a1}\cdot\overline{csc2}+\overline{d1}\cdot d0\\
e0 &= a0\cdot\overline{a1}\cdot\overline{csc3}+\overline{e1}\cdot e0\\
csc0 &= \overline{a1}\cdot csc0+b0\\
csc1 &= csc1\cdot(csc0+\overline{a1})+c0\\
csc2 &= \overline{a1}\cdot csc2+d0\\
csc3 &= csc3\cdot(csc2+\overline{a1})+e0
\end{aligned}
$$

(f) equations for (c)

$$
\begin{aligned}
a1 &= \overline{b1}\cdot\overline{c1}\cdot\overline{d1}\cdot\overline{e1}\cdot\\
&\quad\ \overline{csc0}\cdot\overline{csc1}\cdot\overline{csc2}\cdot\overline{csc3}\\
&\quad +a1\cdot(\overline{csc3}+\overline{csc1}+a0)\\
b0 &= a0\cdot\overline{a1}\cdot csc0\\
c0 &= a0\cdot\overline{a1}\cdot csc1\\
d0 &= a0\cdot\overline{a1}\cdot csc2\\
e0 &= a0\cdot\overline{a1}\cdot csc3\\
csc0 &= \overline{b1}\cdot csc0+a1\\
csc1 &= \overline{c1}\cdot csc1+a1\cdot csc0\\
csc2 &= csc2\cdot\overline{d1}+a1\\
csc3 &= \overline{e1}\cdot csc3+a1\cdot csc2
\end{aligned}
$$

(g) equations for (d)

$$
\begin{aligned}
a1 &= \overline{b1}\cdot\overline{c1}\cdot\overline{d1}\cdot\overline{e1}\cdot\\
&\quad\ \overline{ffc0}\cdot\overline{ffb0}\cdot\overline{ffd0}\cdot\overline{ffe0}\cdot\\
&\quad +a1\cdot(\overline{ffc0}+\overline{ffb0}+\overline{ffe0}+\overline{ffd0})\\
b0 &= \overline{ffb1}\cdot a0\\
c0 &= \overline{ffc1}\cdot a0\\
d0 &= \overline{ffd1}\cdot a0\\
e0 &= \overline{ffe1}\cdot a0\\
ffb0 &= \overline{ffb1}\cdot\overline{b1}\\
ffc0 &= \overline{c1}\cdot\overline{ffc1}\\
ffd0 &= \overline{d1}\cdot\overline{ffd1}\\
ffe0 &= \overline{e1}\cdot\overline{ffe1}\\
ffb1 &= \overline{ffb0}\cdot a0\\
ffc1 &= \overline{ffc0}\cdot a0\\
ffd1 &= \overline{ffd0}\cdot a0\\
ffe1 &= \overline{ffe0}\cdot a0
\end{aligned}
$$

(h) equations for (e)

inputs: a_0, b_1, c_1, d_1, e_1; **outputs:** a_1, b_0, c_0, d_0, e_0;
internal: $csc_0 - csc_3, ffb_0, ffb_1, ffc_0, ffc_1, ffd_0, ffd_1, ffe_0, ffe_1$

Figure 6.12: Handshake decoupling element

remain and can be eliminated in a similar way. The final STG is presented in Figure 6.12(c). Note that in order to reduce the fan-in at a_1^- and the fan-out at a_1^+, some of the falling signal transitions were inserted sequentially. Such an insertion utilises the designer's extra knowledge that the time taken by two internal events can hardly be greater than the time taken by input event a_0^-, and so the latency of the circuit will not be increased. In order to reduce the complexity of the logic due to increased concurrency, an alternative transformation is presented in Figure 6.12(d), where the positive auxiliary transitions ($csc_0^+ - csc_3^+$) are inserted sequentially into the cores. However, their counterparts are inserted in the same way as the previous transformation to reduce latency. Otherwise, the four *csc* signals, if inserted sequentially, would delay the output a_1^-, Alternatively, a combination of sequential and concurrent insertions could be applied to reset the *csc* signals.

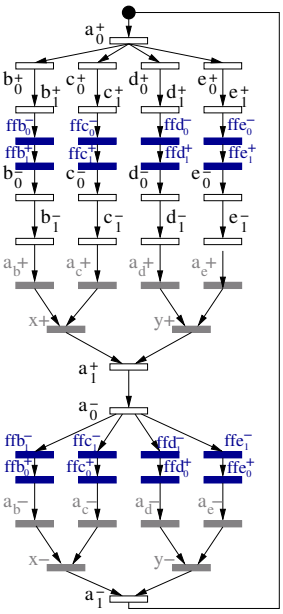
The previous case study shows that flip flop insertion offers a good implementation, which uses simpler gates. Since the handshake decoupling element is similar in nature, it controls five handshakes, where the first handshake initiates the others concurrently. The flip flop approach could be applied to the handshake decoupling element. In this case it is necessary to insert four pairs of complementary signals $[ff\chi_0^-, ff\chi_1^+]$, which realises NOR flip flops. The χ in the signals stands for the ports $b, \ldots, e$. The cores are eliminated by inserting $ff\chi_0^- \twoheadrightarrow ff\chi_1^+$ in the core $\rightarrow\!\wr\chi_0^-$ and $ff\chi_1^- \twoheadrightarrow ff\chi_0^+$ outside the core $\rightarrow\!\wr a_1^-$, such that each resetting phase $ff\chi_1^- \twoheadrightarrow ff\chi_0^+$ is concurrent with each other. The conflict-free STG is shown in Figure 6.12(e).

The three transformed STGs (in Figure 6.12(c)-(e)) are synthesised by PETRIFY using complex gates resulting in the equations shown in Figure 6.12(f)-(h). The flip flop insertion realises a NOR flip flop for each port $b, \ldots, e$, which is set by a_0 and reset by $b_1, \ldots, e_1$, depending on the port. The sequential insertions use more complex gates for the *csc* signals than the flip flop insertion. The concurrent insertion uses more complex logic not only for the *csc* signals but also for the requests $b_0, \ldots, e_0$. However, all transformations have a very complex gate for a_1, which is non-implementable and must be decomposed. The complexity occurs mainly due to the large fan-in. The triggers for a_1 are eight for the flip flop insertion and six for the remaining two. Additionally,

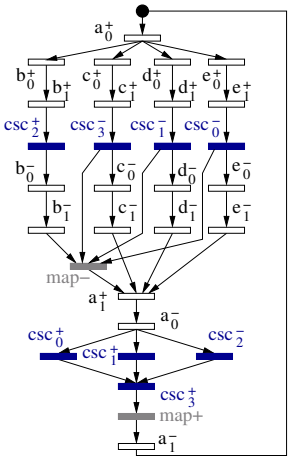
contextual signals are needed to disambiguate $\mathrm{CSC}_{a_1}^{X}$ conflicts during synthesis, where the signals in X are triggers for a_1. The logic decomposition into gates with at most four literals by PETRIFY is not satisfactory, because additional mapping signals are introduced. In particular, some of the mapping signals are inserted in critical paths and contain deep logic, and thus result in slow circuits. Therefore, manual decomposition is applied to the flip flop insertion, which is promising due to the use of simple gates.

The decomposition of the STG resolved by flip flop insertion is presented in Figure 6.13(a). The additional signals x and y are used to minimise the fan-in of the signal a_1. At this stage the logic for the signals x and y is too complex to map it to elements in the library. Therefore, additional signals $a_b, ..., a_e$ are inserted, which help to simplify the logic by reducing the number of triggers of x and y. Note that the additional signals have been inserted in such a way that they preserve the symmetry of the specification. The synthesis of the decomposed STG in Figure 6.13(a) with logic decomposition into complex gates results in the equation shown in Figure 6.13(e). The implementation is depicted in Figure 6.14(a). It contains four D-elements and three C-elements, which are used for synchronisation of the acknowledgement at port a. The implementation of the D-element for port b is presented in Figure 6.14(b). Note that the signals $a_b, ..., a_e$ used for decomposition act as an intermediate acknowledgement for port $a...e$, which are then synchronised by the C-elements.

The initial specification was also given to PETRIFY, which solved CSC conflicts automatically and used one additional signal, *map*, for decomposition into gates with at most four literals. The derived STG is shown in Figure 6.13(b). PETRIFY also uses four signals to resolve the CSC conflicts, but resets the *csc* signals (in the handshake of port a) differently. The polarities of the csc_2 signals are opposite to the other *csc* signals, making the polarities of the *csc* signals asymmetrical. The implementation is shown in Figure 6.13(d), which is more compact than the flip flop implementation. However, it has more complex logic, especially for the signal csc_3 and *map*, which contribute to the worst case delay. This occurs between the input a_0^- and the output a_1^-, e.g. with a trace of $a_0^- \rightarrow csc_0^+ \rightarrow csc_3^+ \rightarrow map^+ \rightarrow a_1^-$. Taking into account the fact that CMOS logic is built out of negative gates these events correspond to the following sequence



(a) final STG (flip flop insertion)



(b) STG derived by PETRIFY

$$
\begin{aligned}
a1 &= a1 \cdot (x + y) + x \cdot y \\
b0 &= \overline{ffb1} \cdot a0 \\
c0 &= \overline{ffc1} \cdot a0 \\
d0 &= \overline{ffd1} \cdot a0 \\
e0 &= \overline{ffe1} \cdot a0 \\
ffb0 &= \overline{ffb1} \cdot \overline{b1} \\
ffc0 &= \overline{c1} \cdot \overline{ffc1} \\
ffd0 &= \overline{d1} \cdot \overline{ffd1} \\
ffe0 &= \overline{e1} \cdot \overline{ffe1} \\
ffb1 &= \overline{ffb0} \cdot a0 \\
ffc1 &= \overline{ffc0} \cdot a0 \\
ffd1 &= \overline{ffd0} \cdot a0 \\
ffe1 &= \overline{ffe0} \cdot a0 \\
ab &= \overline{b1} \cdot \overline{ffb0} \\
ac &= \overline{ffc0} \cdot \overline{c1} \\
ad &= \overline{d1} \cdot \overline{ffd0} \\
ae &= \overline{e1} \cdot \overline{ffe0} \\
x &= x \cdot (ac + ab) + ab \cdot ac \\
y &= y \cdot (ad + ae) + ad \cdot ae
\end{aligned}
$$

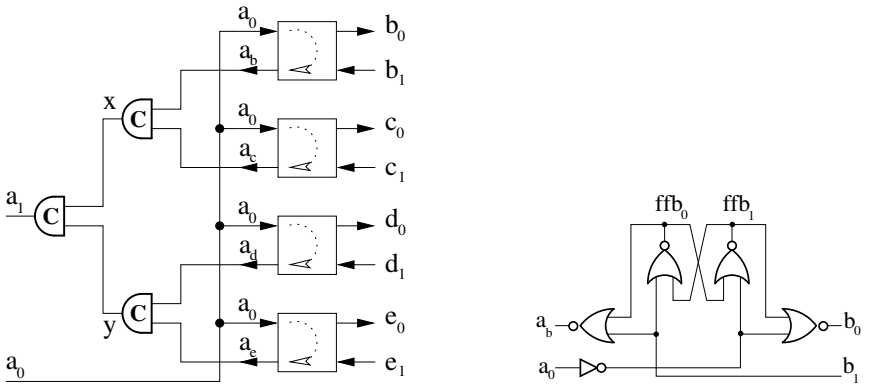
(c) equations for (a)

$$
\begin{aligned}
a1 &= \overline{d1} \cdot \overline{e1} \cdot \overline{c1} \cdot \overline{map0} \\
b0 &= a0 \cdot \overline{csc2} \\
c0 &= a0 \cdot csc3 \\
d0 &= csc1 \cdot a0 \\
e0 &= csc0 \cdot a0 \\
csc0 &= \overline{e1} \cdot (\overline{a0} + csc0) \\
csc1 &= \overline{d1} \cdot csc1 + \overline{a0} \\
csc2 &= a0 \cdot (csc2 + b1) \\
8 &= csc0 \cdot csc1 \cdot \overline{a0} \cdot \overline{csc2} \\
csc3 &= \overline{c1} \cdot csc3 + 8 \\
10 &= map(\overline{csc2} + csc0) + csc3 \\
map &= map \cdot (b1 + csc1) + 10
\end{aligned}
$$

(d) equations for (b)

inputs: a_0, b_1, c_1, d_1, e_1; **outputs:** a_1, b_0, c_0, d_0, e_0;
internal: $csc_0 - csc_3, map, ffb_0, ffb_1, ffc_0, ffc_1, ffd_0, ffd_1, ffe_0, ffe_1, a_b, a_c, a_d, a_e, x, y$

Figure 6.13: Handshake decoupling element: final implementation



(a) block diagram

(b) D-element for port b

Figure 6.14: Implementation of the decomposed flip flop solution

of gate switching: $[a_0\downarrow] \rightarrow [\overline{csc_0}\downarrow] \rightarrow [csc_0\uparrow] \rightarrow [\overline{8}\uparrow] \rightarrow [8\downarrow] \rightarrow [\overline{csc_3}\downarrow] \rightarrow [csc_3\uparrow] \rightarrow [\overline{10}\uparrow] \rightarrow [10\downarrow] \rightarrow [\overline{map}\downarrow] \rightarrow [map\uparrow] \rightarrow [a_1\downarrow]$. This gives the latency estimation of eleven negative gate delays compared with six negative gate delays for the flip flop solution. The latter occurs between the input a_0^- and the output a_1^-, e.g with a trace of $a_0^- \rightarrow ffb_1^- \rightarrow ffb_0^+ \rightarrow a_b^- \rightarrow x^+ \rightarrow a_1^-$. The gates switching between these transitions are $[a_0\downarrow] \rightarrow [\overline{a_0}\uparrow] \rightarrow [ffb_1\downarrow] \rightarrow [ffb_0\uparrow] \rightarrow [a_b\downarrow] \rightarrow [x\uparrow] \rightarrow [a_1\downarrow]$, giving a latency estimate of six negative gate delays. The corresponding delays obtained from simulation is 2.67 ns for Petrify's solution and 1.97 ns for flip flop solution (using AMS-0.35μ CMOS technology).

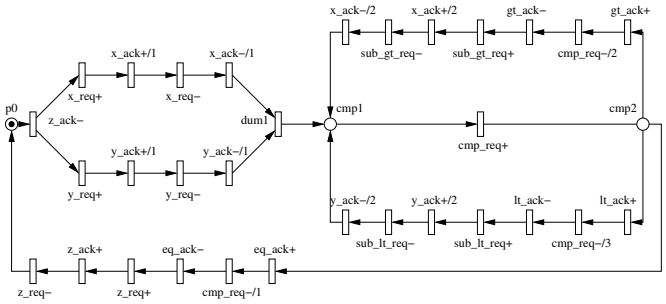
Despite the larger number of signals used for decomposition, the flip flop solution has an estimate lower latency than the automated one. Additionally, it is built out of simple gates and C-elements. This example shows that manual intervention with the use of the visualisation resolution approach produces a better solution that the automated one. The understanding of the cause of CSC conflicts, by the designer makes such a solution practicable.

6.7 GCD

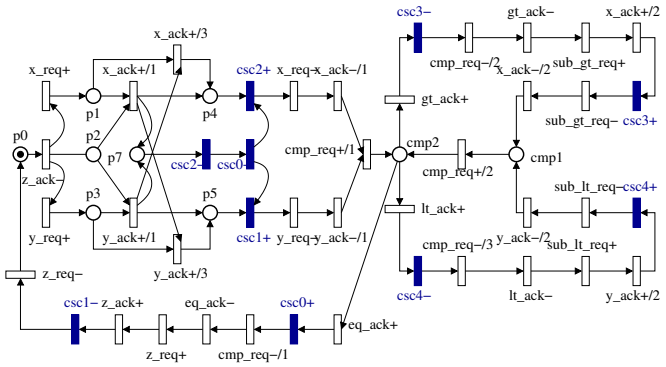
GCD is a module that computes the greatest common divisor of two integers. To find the GCD of two numbers x and y an algorithm is used, which repeatedly replaces

the larger by subtracting the smaller from it until the two numbers are equal. This algorithm requires only subtraction and comparison operations, and takes a number of steps proportional to the difference between the initial numbers. The STG for the GCD control unit [97] is presented in Figure 6.15(a). The labels gt, eq and lt correspond to the comparison of x and y values and stand for "greater than", "equal" and "less then", respectively. The signal cmp_req is used for the comparison. The assignment of the subtraction results in the signal being split into the subtraction operation (sub_gt and sub_lt). The signals z_ack and z_req compose the handshake interface to the environment. The z_req signal, when set, means that the computation is complete and output data is ready to be consumed. The z_ack signal is set when the output of the previous computation cycle is consumed and the new input data is ready to be processed. The dummy signal $dum1$ is used for synchronisation for the comparison.

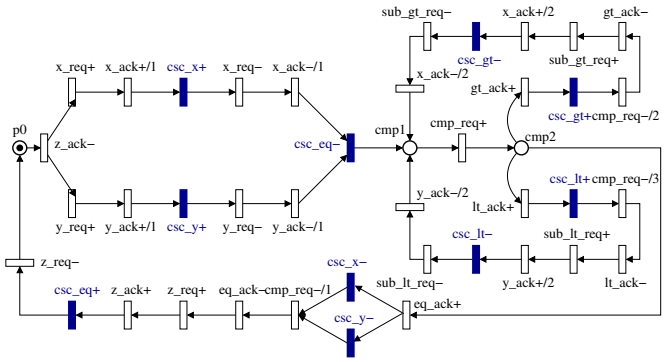
The STG in Figure 6.15(a) has several CSC conflicts. They are resolved by using the tool PETRIFY based on the theory of regions, and the tool CONFRES based on core visualisation. The former is derived automatically, resulting in the conflict-free STG in 6.15(b). Due to the fact that the resolution process takes place at the SG level, the STG is transformed back into an STG, with a different structure because the structural information is lost during this process. The different structure might be inconvenient for further manual modification. The derived STG in Figure 6.15(b) has two changes to the structure which are not due to signal insertion. Firstly, the transition cmp_req^+ is split into $cmp\_req^+/1$ and $cmp\_req^+/2$. Secondly, the concurrent input of x and y is synchronised on $cmp\_req^+/1$ instead of the dummy transition. PETRIFY resolves the GCD control unit by adding five new signals, namely csc_0 to csc_4. The synthesis of the conflict-free specification with logic decomposition into gates with at most four literals results in the equation shown in Figure 6.16. The estimated area is 432 units and the maximum and average delay between the inputs is 4.00 and 1.75 signal events respectively. The worst case latency is between the input $x\_ack^+/1$ and the output x_req^-. The trace of transitions is $x\_ack^+/1 \rightarrow csc_2^- \rightarrow csc_0^- \rightarrow csc_2^+ \rightarrow x\_req^-$. Taking into account that CMOS logic is built out of negative gates these events correspond to the following sequence of gate switching: $[x\_ack \uparrow] \rightarrow [\overline{z\_ack} \downarrow$



(a) initial STG



(b) STG derived by Petrify



(c) STG derived by visualisation (sequential insertion)

inputs: $x\_ack, y\_ack, z\_ack, eq\_ack, lt\_ack, gt\_ack$;
outputs: $x\_req, y\_req, z\_req, cmp\_req, sub\_lt\_req, sub\_gt\_req$;
dummy: $dum1$; **internals:** $csc0 - csc4, csc\_y, csc\_x, csc\_eq, csc\_lt, csc\_gt$

Figure 6.15: GCD

$] \rightarrow [\overline{csc_2}\uparrow] \rightarrow [csc_2\downarrow] \rightarrow [\overline{csc_0}\uparrow] \rightarrow [\overline{9}\downarrow] \rightarrow [9\uparrow] \rightarrow [\overline{csc_2}\downarrow] \rightarrow [\overline{x\_req}\uparrow] \rightarrow [x\_req\downarrow]$. This gives the latency estimate equal to the delay of nine negative gates. The corresponding delay obtained from simulation is 1.91 ns (using AMS-0.35μCMOS technology).

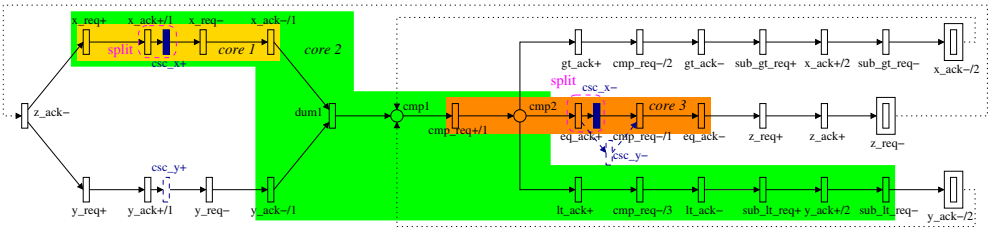
$$
\begin{aligned}
\text{x\_req} &= \overline{\text{z\_ack}} \cdot (\text{csc0} \cdot \overline{\text{csc1}} + \overline{\text{csc2}}) \\
\text{y\_req} &= \overline{\text{z\_ack}} \cdot \overline{\text{csc1}} \\
\text{z\_req} &= \text{csc0} \cdot \overline{\text{eq\_ack}} \cdot \text{csc1} \\
3 &= \overline{\text{csc4}} + \overline{\text{csc3}} + \text{csc0} + \overline{\text{csc2}} \\
\text{cmp\_req} &= \overline{3} \cdot \overline{\text{x\_ack}} \cdot \overline{\text{y\_ack}} \cdot \text{csc1} \\
\text{sub\_gt\_req} &= \overline{\text{csc3}} \cdot \overline{\text{gt\_ack}} \\
\text{sub\_lt\_req} &= \overline{\text{csc4}} \cdot \overline{\text{lt\_ack}} \\
\text{csc0} &= \text{csc2} \cdot (\text{eq\_ack} + \text{csc0}) \\
\text{csc1} &= \overline{\text{csc0}} \cdot \text{y\_ack} + \overline{\text{z\_ack}} \cdot \text{csc1} \\
9 &= \overline{\text{csc0}} \cdot (\text{csc2} + \text{x\_ack}) \\
\text{csc2} &= \overline{\text{x\_ack}} \cdot \text{csc0} \cdot \overline{\text{y\_ack}} + 9 \\
\text{csc3} &= \overline{\text{gt\_ack}} \cdot (\text{csc3} + \text{x\_ack}) \\
\text{csc4} &= \text{csc4} \cdot \overline{\text{lt\_ack}} + \text{y\_ack}
\end{aligned}
$$

Figure 6.16: Equations for GCD controller obtained by PETRIFY

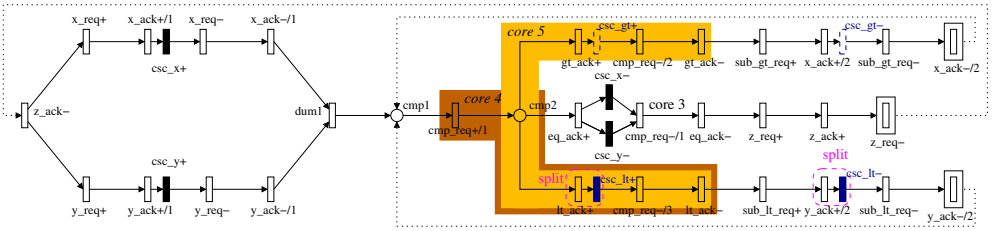
The encoding conflicts in the GCD control unit are also resolved by using CONFRES, which provides an interaction with the user during the resolution process. The process of core elimination is illustrated in Figure 6.17. The encoding conflicts are caused by ten overlapping CSC cores. However, the cores would hardly be distinguishable, even if different colours are used. That is the reason why only those cores whose resolution is discussed are shown. Note that CONFRES offers an option to depict a subset of cores, chosen by the user.

Two experiments are considered. In the first, sequential signal insertion is exploited in order to compete the automatic conflict resolution in circuit size. In the second experiment, the auxiliary signals are inserted concurrently (where possible) in order to achieve lower latency.

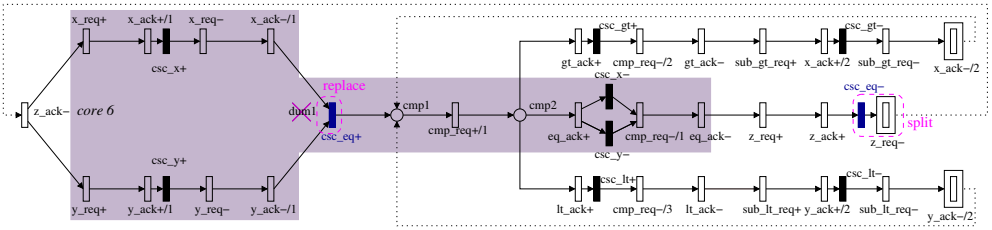
The cores 1 and 2 shown in Figure 6.17(a) are eliminated by inserting csc_x^+ transition sequentially $x\_ack^+/1 \wr\rightarrow$ in phase one, and in phase two the transition csc_x^- is also inserted sequentially $eq\_ack^+ \wr\rightarrow$, thereby eliminating the core 3. Two other cores, symmetrical to cores 1 and 2 (not shown for readability), are eliminated in a similar way. The transition csc_y^+ is inserted $y\_ack^+/1 \wr\rightarrow$. However, csc_y^- is



(a) step 1: elemination of core 1-3, and cores symetric to core 1 and 2



(b) step 2: elimination of core 4 and 5, and the core symetric to the core 4



(c) step 3: elimination of core 6

inputs: $x\_ack, y\_ack, z\_ack, eq\_ack, lt\_ack, gt\_ack$;
outputs: $x\_req, y\_req, z\_req, cmp\_req, sub\_lt\_req, sub\_gt\_req$;
dummy: $dum1$; **internals:** $csc\_y, csc\_x, csc\_eq, csc\_lt, csc\_gt$

Figure 6.17: Resolution process of GCD controller

inserted concurrently to csc_x^- $eq\_ack^+ \rightarrowtail\!\!\!\!\!\rightarrow cmp\_req^-/1$ to reduce latency. Then, two remaining cores, cores 4 and 5, shown in Figure 6.17(b) are eliminated by inserting csc_lt^+ sequentially $lt\_ack^+ \wr\!\!\rightarrow$. The counterpart transition csc_lt^- is inserted outside the cores sequentially $y\_ack^+/2 \wr\!\!\rightarrow$. Likewise, the core which is symmetrical to core 4 (not shown for readability) is destroyed by inserting $csc\_gt^+ gt\_ack^+ \wr\!\!\rightarrow$ and csc_gt^- $x\_ack^+/2 \wr\!\!\rightarrow$. Finally, only one core remains, core 6, which is shown in Figure 6.17(c). It is eliminated by replacing the dummy transition $dum1$ by csc_eq^-, and csc_eq^+ is inserted outside the core $\rightarrow \wr z\_req^-$. The resulting conflict-free STG is depicted in Figure 6.15(c). The equations in Figure 6.18(a) have been synthesised by PETRIFY with logic decomposition into gates with at most four literals. The estimated area is 432 units, which is the same as PETRIFY's automated resolution. However, the maximum and average delays between the inputs are significantly improved: 2.00 and 1.59 signal events respectively. The worst case latency of the circuit is between $eq_ack^+$and $cmp\_req^-/1$. If the circuit is implemented using CMOS negative gates then this latency corresponds to the following sequence of gate switching: $[eq\_ack \uparrow] \rightarrow [\overline{eq\_ack} \downarrow] \rightarrow [\overline{csc\_x} \uparrow] \rightarrow [\overline{3} \downarrow] \rightarrow [3 \uparrow] \rightarrow [\overline{cmp\_req} \uparrow] \rightarrow [cmp\_req \downarrow]$. This gives the latency estimate equal to the delay of six negative gates, which is better than PETRIFY's automated resolution. The corresponding delay obtained from simulation is 1.70 ns (using AMS-0.35μ CMOS technology).

The other experiment aims at lower latency of the GCD control unit. The auxiliary signal transitions are inserted as concurrently as possible. Namely, csc_x^+ is inserted concurrently with $x\_ack^+/1$ $x\_req^+ \rightarrowtail\!\!\!\!\!\rightarrow x\_req^-$; csc_y^+ is inserted $y\_req^+ \rightarrow$ $\rightarrow y\_req^-$; csc_gt^- is inserted $sub\_gt\_req^+ \rightarrowtail\!\!\!\!\!\rightarrow sub\_gt\_req^-$; csc_lt^- is inserted $sub\_lt\_req^+ \rightarrowtail\!\!\!\!\!\rightarrow sub\_lt\_req^-$. The other transitions are inserted in the same way as in the previous experiment. The synthesis of the constructed conflict-free STG produces the equations in Figure 6.18(b). Some of the equations become more complex due to fact that the extended concurrency increases the number of combinations of reachable signal values and thus reduces the "don't care" values for Boolean minimisation. In order to decomposed these into library gates with at most four literals PETRIFY adds two new signals, map_0 and map_1. This results in a larger estimated circuit size, 592

units. The average input to input delay of the circuit becomes 1.34, which is smaller than in the previous experiment. However, the maximum latency of the circuit is seven negative gate delays. It occurs for example between gt_ack^+ and cmp_req^-. The gates switched between these transitions are: $[gt\_ack\uparrow] \rightarrow [\overline{csc\_gt}\downarrow] \rightarrow [csc\_gt\uparrow] \rightarrow [\overline{map_0}\downarrow] \rightarrow [map_0\uparrow] \rightarrow [\overline{5}\downarrow] \rightarrow [\overline{cmp\_req}\uparrow] \rightarrow [cmp\_req\downarrow]$. The worst case latency in this implementation is greater than the latency in the previous design due to the internal map_0 and map_1 signals, which are used for decomposition of non-implementable functions.

$$
\begin{aligned}
\text{x\_req} &= \overline{\text{csc\_x}} \cdot \overline{\text{z\_ack}} \cdot \text{csc\_eq} \\
\text{y\_req} &= \overline{\text{csc\_y}} \cdot \overline{\text{z\_ack}} \cdot \text{csc\_eq} \\
\text{z\_req} &= \overline{\text{csc\_x}} \cdot \overline{\text{eq\_ack}} \cdot \overline{\text{csc\_eq}} \\
3 &= \overline{\text{csc\_y}} \cdot \overline{\text{csc\_x}} + \text{csc\_gt} + \text{csc\_lt} \\
\text{cmp\_req} &= \overline{3} \cdot \text{y\_ack} \cdot \overline{\text{x\_ack}} \cdot \overline{\text{csc\_eq}} \\
\text{sub\_gt\_req} &= \text{csc\_gt} \cdot \overline{\text{gt\_ack}} \\
\text{sub\_lt\_req} &= \text{csc\_lt} \cdot \overline{\text{lt\_ack}} \\
\text{csc\_x} &= \overline{\text{eq\_ack}} \cdot (\text{x\_ack} + \text{csc\_x}) \\
\text{csc\_y} &= \text{csc\_y} \cdot \overline{\text{eq\_ack}} + \text{y\_ack} \\
\text{csc\_gt} &= \overline{\text{x\_ack}} \cdot \text{csc\_gt} + \text{gt\_ack} \\
10 &= \text{csc\_eq} \cdot (\overline{\text{csc\_x}} + \overline{\text{csc\_y}}) + \text{z\_ack} \\
\text{csc\_eq} &= \text{csc\_eq} \cdot (\text{x\_ack} + \text{y\_ack}) + 10 \\
\text{csc\_lt} &= \overline{\text{y\_ack}} \cdot (\text{lt\_ack} + \text{csc\_lt})
\end{aligned}
$$

(a) sequential insertion

$$
\begin{aligned}
0 &= \overline{\text{csc\_x}} \cdot \text{csc\_eq} \\
\text{x\_req} &= \text{x\_req} \cdot \overline{\text{map0}} + 0 \cdot \overline{\text{z\_ack}} \\
2 &= \overline{\text{csc\_y}} \cdot \text{z\_ack} \cdot \text{csc\_eq} \\
\text{y\_req} &= \overline{\text{y\_ack}} \cdot \text{y\_req} + 2 \\
\text{z\_req} &= \overline{\text{csc\_y}} \cdot \overline{\text{eq\_ack}} \cdot \overline{\text{csc\_eq}} \\
5 &= \overline{\text{csc\_y}} \cdot \overline{\text{csc\_x}} + \text{map0} + \text{csc\_eq} \\
\text{cmp\_req} &= \overline{\text{sub\_lt\_req}} \cdot \overline{5} \cdot (\text{map1} + \text{eq\_ack}) \\
\text{sub\_gt\_req} &= \overline{\text{gt\_ack}} \cdot (\text{sub\_gt\_req} \cdot \text{map1} + \text{csc\_gt}) \\
\text{sub\_lt\_req} &= \overline{\text{lt\_ack}} \cdot (\text{sub\_lt\_req} \cdot \text{map1} + \text{csc\_lt}) \\
\text{csc\_x} &= \text{csc\_x} \cdot \overline{\text{eq\_ack}} + \text{x\_req} \\
\text{csc\_y} &= \overline{\text{eq\_ack}} \cdot (\text{csc\_y} + \text{y\_req}) \\
\text{csc\_lt} &= \overline{\text{sub\_lt\_req}} \cdot (\text{lt\_ack} + \text{csc\_lt}) \\
\text{csc\_gt} &= \overline{\text{sub\_gt\_req}} \cdot (\text{gt\_ack} + \text{csc\_gt}) \\
\text{csc\_eq} &= \overline{\text{map1}} \cdot (\text{csc\_eq} + \text{z\_ack}) \\
\text{map0} &= \text{sub\_gt\_req} + \text{csc\_gt} + \text{csc\_lt} + \text{x\_ack} \\
15 &= \overline{\text{csc\_x}} + \text{x\_req} + \overline{\text{csc\_y}} \\
\text{map1} &= \overline{15} \cdot \text{y\_ack} \cdot \overline{\text{y\_req}} \cdot \overline{\text{x\_ack}}
\end{aligned}
$$

(b) concurrent insertion

Figure 6.18: Equations for GCD controller obtained by ConfRes

The complex gate implementation of the GCD controller, where CSC conflict is resolved manually by inserting new signals in series with existing ones is shown in Figure 6.19. This is the best solution (in terms of size and latency) synthesised by Petrify with the help of the ConfRes tool. It consists of 120 transistors and exhibits a latency of five negative gates delay.

Clearly, the method based on core visualisation gives the designer a lot of flexibility in choosing between the circuit size and latency. It also maintains the symmetry of the specification whilst solving encoding conflicts.

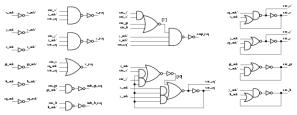


Figure 6.19: Complex gates implementation of GCD controller (sequential solution)

6.8 Conclusion

The design examples show that the transparent and interactive synthesis, together with the experience of the designer, makes it possible to produce tailor-made solutions. The designer is able to understand the cause of encoding conflicts and resolve them according to the design constraints, whilst considering different types of transformations.

	transformation		area [unit]		latency [unit]	
benchmark	auto	man	auto	man	auto	man
VME-bus read cycle	SI-seq	SI-conc*	96	176	7.5	5.0
Weakly synchronised pipeline	SI-seq	CR	216	224	n.a.	n.a.
Phase comparator	n.a.	SI-seq‡	n.a.	240	n.a.	7.0
AD converter: top level	SI-seq	CR†	176	152	14.5	10.0
AD converter: scheduler	SI-mix	SI-conc‡	512	456	15.5	6.5
D-element	SI-seq	FF-seq	80	96	6.0	4.5
Handshake decoupling element	SI-seq	FF-seq	384	576	16.0	10.5
GCD	SI-seq	SI-mix	432	432	13.0	8.0

auto: automatic refinement by PETRIFY
man: interactive refinement by using CONFRES
SI: signal insertion
FF: flip flop insertion
CR: concurrency reduction
seq: sequential insertion
conc: concurrent insertion
mix: both sequential and concurrent insertion
*solution 12
†solution 2
‡timing assumptions used

Table 6.2: Comparison: automatic and manual refinement

The synthesis solutions obtained automatically by PETRIFY and by the intervention of the designer are compared in Table 6.2. The types of transformation used to resolve

encoding problems are presented. These types include either concurrency reduction or signal insertion, which may be sequential, concurrent or both. PETRIFY has been used to derive equations with logic decomposition into library gates with at most four literals. The area is estimated by PETRIFY. The latency is measured as the accumulative delay of gate switches between an input and the next output. The negative gate delay depends on its complexity and is estimated as follows. The latency of an inverter is associated with 1.0 unit delay. Gates which have a maximum of two transistors in their transistor stacks are associated with 1.5 units; 3 transistors with 2.0 units; 4 transistors with 2.5 units. These approximations dependency are derived from the analysis of the gates in AMS 0.35 μm library. This method of latency estimation does not claim to be very accurate. However, it takes into account not only the number of gates switched between an input and the next output, but also the complexity of these gates.

The manual interventions in the resolution process of encoding conflicts in Table 6.2 were undertaken to improve the performance of the circuits. The experiments demonstrate this. However, lower output latencies are achieved at the cost of higher area overheads. In the manual approach, additional signal transitions are often inserted, for example, to preserve regular structures, for decomposition or as a result of flip flop insertions. Additionally, the use of concurrent transition insertions contributes to more complex functions, because the reachable state space is increased. For example, in the read cycle of the VME-bus the manual transformation involves the concurrent insertion of an additional signal. This results in lower latency, but the complexity of the functions contributes to a larger area compared with the automatic resolution, which uses sequential insertions. Note that concurrent insertions increase the reachable state space leaving less room for Boolean minimisation. In some cases, this is the reason why concurrent insertions achieve slower circuits compared with sequential insertions, which delay output signals. Flip flop insertion introduces two complementary signals per resolution step, thus contributing to the increased area. However, simple flip flop insertions produce simple gates for the inserted signals. In general, these two properties cancel each other out. Concurrency reduction may result in a smaller area, since some reachable states are made redundant. This is the case in the top level of the AD con-

verter. In the scheduler of the AD converter the manual intervention made it possible to use timing assumptions to decrease the complexity of some functions, further decreasing the latency.

These experiments show that the interactive approach makes it possible to analyse a large range of transformations. The designer is able to achieve a balance between area and latency in order to select a transformation which best suits the design constraints.

Chapter 7

Conclusion

This thesis presents a framework for interactive synthesis of asynchronous circuits from Signal Transition Graphs (STGs). In particular, the fundamental problem of encoding conflicts in the synthesis is tackled. The framework uses partial order in the form of an STG unfolding prefix, which is an acyclic net with simple structures, offering a compact representation of the reachable state space.

7.1 Summary

The synthesis based on STGs involves the following steps: (a) checking sufficient conditions for the implementability of the STG by a logic circuit; (b) modifying, if necessary, the initial STG to make it implementable; and (c) finding appropriate Boolean next-state functions for non-input signals.

A commonly used tool, PETRIFY [18], performs all these steps automatically, after first constructing the reachability graph of the initial STG specification. To gain efficiency, it uses symbolic (BDD-based) techniques to represent the STG's reachable state space. While such an approach is convenient for completely automatic synthesis, it has several drawbacks: state graphs represented explicitly or in the form of BDDs are hard to visualise due to their large sizes and the tendency to obscure causal relationships and concurrency between the events. This prevents efficient interaction with the designer. Moreover, the combinatorial explosion of the state space is a serious issue for highly

concurrent STGs, putting practical limits on the size of synthesisable control circuits. Other tools based on alternative techniques, and in particular those employing Petri net unfoldings, can be applied to avoid the state explosion problem.

The unfolding approach in Chapter 4 has been applied to the implementability analysis in step (a), viz. checking the Complete State Coding (CSC) condition, which requires the detection of encoding conflicts between reachable states of an STG. In Chapter 5 this approach has also been applied to step (b), in particular for enforcing the CSC condition (i.e. for the resolution of CSC conflicts), which is a necessary condition for the implementability of an STG as a circuit. The work in [42] addresses step (c), where an unfolding approach is used to derive equations for logic gates of the circuit. Together with [40] (an alternative approach for detecting CSC conflicts, which is more efficient as the one described in Chapter 4) and with the framework in Chapter 5 they form a complete design flow for complex gate synthesis of asynchronous circuits, based on STG unfoldings rather than state graphs. Figure 7.1 illustrates the application of the interactive resolution of encoding conflicts in syntheses developed in this thesis. It can be applied either with the state-based synthesis used by PETRIFY or with the unfolding-based synthesis used by VERISAT [42].

The detection of CSC conflicts in Chapter 4 is based on [47], which has been extended and examined for efficiency and effectiveness. The experimental results indicate that this approach is impractical due to its refinement procedure, where a partial state space is built for each detected conflict. Moreover, the detected conflicts may include "fake" conflicts due to overestimation, which are found to be proportional to the number of states. Therefore this approach is inefficient in detecting CSC conflict in STGs, where concurrency dominates. However, during this time an unfolding approach in [39] and later in [40] had been proposed, which are based on integer programming and SAT, respectively. These approaches have proved to be efficient for the purposes of identifying encoding conflicts in STG unfolding prefixes and therefore are used for the resolution of encoding conflicts in Chapter 5.

In Chapter 5 a framework for visualisation and resolution of encoding conflicts has been proposed. A novel visualisation technique has been presented which shows

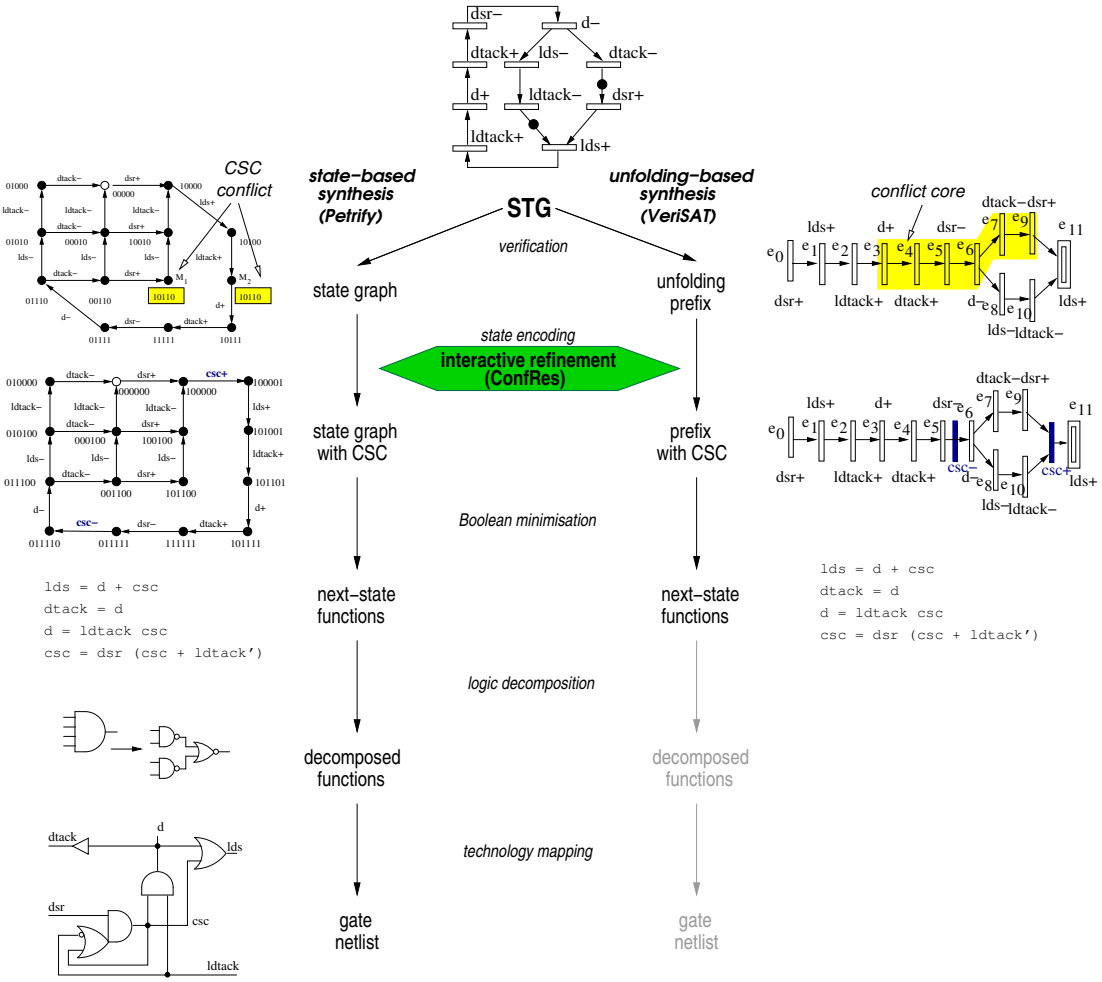


Figure 7.1: Applications of interactive refinement of CSC conflicts

the causes of encoding conflicts in STG unfolding prefixes without enumerating them. Based on the visualisation technique a resolution process has been developed. It employs transformations which are based on conflict cores, and transformation constraints. The transformations include signal insertion and concurrency reduction. The former introduces an additional memory to the system to resolve the conflicts, and the latter introduces additional causal constraints to remove, in particular, conflicting reachable states. Additionally, an alternative signal insertion involving pairs of complementary signals has been applied, which realises flip flop structures. These transformations allow the designer to explore a larger design space.

The resolution procedure employs heuristics for transformations based on the concept of cores and the exploitation of the intersections of cores, resulting in the use of the most essential information about encoding conflicts. It can be applied manually or can be automated, either partially or completely. However, in order to obtain optimal solutions, a semi-automated resolution process should be employed. In this type of resolution, pre-computed transformations are used as guidelines, yet the designer is free to intervene at any stage and choose a transformation according to design constraints.

The visualisation and resolution of encoding conflict has been applied not only to CSC conflicts but also to normalcy violations and CSC_z^X conflicts for a non-input signal z w.r.t. its support X. The CSC property is a necessary condition for the implementability of an STG as a circuit, and thus it is a fundamental problem during the synthesis. However, other encoding conflicts might be of interest in order to alter functions derived during synthesis. In particular, the normalcy property is a necessary condition to implement STGs as logic circuits which are built from monotonic gates. The elimination of CSC_z^X conflicts can be used for decomposition.

Finally, in Chapter 6 the proposed framework for the interactive synthesis has been applied to a number of benchmarks to demonstrate its effectiveness. They show that a combination of the visualisation approach and the intellectual effort of a human designer achieves better solutions than automated ones. The compact representation of encoding conflicts allows the designer to understand the causes of encoding conflicts. This helps the designer to intervene and use their experience to produce optimal solutions according

to design constraints.

7.2 Areas of further research

This transparent approach to synthesis promotes the involvement of designers to obtain optimal solutions by using their experience. Moreover, use of the unfolding-based synthesis in the asynchronous circuit design is potentially high. Therefore, unfolding-based and transparent synthesis should be investigated further.

Resolution process The resolution process involves transformations of an STG using information derived from its unfolding prefix. This requires the construction of the unfolding prefix at each iteration of the resolution process. To improve efficiency, the transformation should be applied directly to the prefix whenever possible. Therefore, alternative transformations of unfolding prefixes should be examined. An algorithm for checking the validity of a concurrency reduction on safe nets should also be developed. Timing assumption could be used to resolve some encoding conflicts. They could be derived from transformations based on concurrency reduction and used to set the firing order of concurrent events. In addition, the cost function should be improved by considering gate level optimisation involving contextual signals and a path level cost. This cost would determine paths between adjacent input signals and their distance would indicate potential worst case delays. The trend in conventional logic synthesis goes towards "physical synthesis", bridging synthesis and place-and-route. Similarly, this could be applied to asynchronous synthesis by incorporating physical level optimisation involving some more detailed physical parameters such as transistor sizes, interconnects and loading parasitics.

Decomposition and technology mapping The problem for technology mapping of asynchronous circuits deals with the implementation of a circuit in a given gate library. It requires an implementation of the circuit using a restricted set of gates. The unfolding-based synthesis only produces complex gates, which are not always mappable into the library elements. Solving this problem requires decomposition

of the Boolean functions into simpler components, preserving speed-independence. The decomposition can often be reformulated as a problem of inserting additional signals into the specification and then resolving the encoding conflicts. Therefore, it is related to the resolution of encoding conflicts discussed in this thesis. In particular, CSC_z^X conflicts can be used for decomposition.

Extension of tool ConfRes The visualisation and resolution of normalcy and CSC_z^X conflicts should be incorporated into this tool. The conflict cores could be derived from conflict sets corresponding to these conflicts, which could be extracted from VeriSAT, an unfolding-based synthesis and verification tool. The proposed resolution process could then be applied. Furthermore, the resolution process could be fully automated in order to be used in the unfolding-based synthesis approach, without involving the designer.

Bibliography

[1] Asynchronous Circuit Design Working Group. http://www.scism.sbu.ac.uk/ccsv/ACiD-WG/, 2005.

[2] The asynchronous logic homepage. http://www.cs.man.ac.uk/async/, 2005.

[3] M. Kishinevsky A. Kondratyev and A. Taubin. Synthesis method in self-timed design. Decompositional approach. In *IEEE International Conference on VLSI and CAD*, pages 16–25, November 1993.

[4] P. Beerel and T.H.-Y. Meng. Automatic Gate-Level Synthesis of Speed-Independent Circuits. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 581–587. IEEE Computer Society Press, November 1992.

[5] C. H. (Kees) van Berkel, Mark B. Josephs, and Steven M. Nowick. Scanning the Technology: Applications of Asynchronous Circuits. *Proceedings of the IEEE*, 87(2):223–233, February 1999.

[6] Kees van Berkel. Beware the Isochronic Fork. *Integration, the VLSI journal*, 13(2):103–128, June 1992.

[7] Kees van Berkel, Joep Kessels, Marly Roncken, Ronald Saeijs, and Frits Schalij. The VLSI-Programming Language Tangram and Its Translation into Handshake Circuits. In *Proc. European Conference on Design Automation (EDAC)*, pages 384–389, 1991.

[8] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[9] Janusz A. Brzozowski and Carl-Johan H. Seger. *Asynchronous Circuits*. Springer-Verlag, 1995.

[10] A. Bystrov, D. Shang, F. Xia, and A. Yakovlev. Self-timed and speed independent latch circuits. In *Proceedings of the 6th UK Asynchronous Forum*. Department of Computing Science, The University of Manchester, Manchester, July 1999.

[11] Josep Carmona, Jordi Cortadella, and Enric Pastor. A structural encoding technique for the synthesis of asynchronous circuits. *Fundamenta Informaticae*, 50(2):135–154, 2002.

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

[13] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Complete State Encoding based on the Theory of Regions. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1996.

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

[15] J. Cortadella, M.Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Automatic handshake expansion and reshuffling using concurrency reduction. In *Proc. of the Workshop Hardware Design and Petri Nets (within the International Conference on Application and Theory of Petri Nets)*, pages 86–110, 1998.

[16] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, Enric Pastor, and Alexandre Yakovlev. Decomposition and Technology Mapping of Speed-Independent Circuits Using Boolean Relations. *IEEE Transactions on Computer-Aided Design*, 18(9), September 1999.

[17] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alex Yakovlev. Hardware and Petri Nets: Application to Asynchronous Circuit Design. *Lecture Notes in Computer Science*, 1825:1–15, 2000.

[18] 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.

[19] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alexandre Yakovlev. A Region-Based Theory for State Assignment in Speed-Independent Circuits. *IEEE Transactions on Computer-Aided Design*, 16(8):793–812, August 1997.

[20] W. J. Dally and J. W. Poulton. *Digital system engineering.* Cambridge University Press, 1998.

[21] Al Davis and Steven M. Nowick. An Introduction to Asynchronous Circuit Design. Technical Report UUCS-97-013, Dept. of Computer Science, University of Utah, September 1997.

[22] David L. Dill. *Trace theory for automatic hierarchical verification of speed-independent circuits.* MIT Press, 1989.

[23] 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.

[24] D. A. Edwards and W. B. Toms. The status of asynchronous design in industry. In *Information society technologies (IST) programme concerted action thematic network contract*, February 2003.

[25] Joost Engelfriet. Branching Processes of Petri Nets. *Acta Informatica*, 28:575–591, 1991. NewsletterInfo: 38, 40.

[26] Javier Esparza, Stefan Römer, and Walter Vogler. An Improvement of McMillan's Unfolding Algorithm. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 87–106, 1996.

[27] Javier Esparza, Stefan Römer, and Walter Vogler. An Improvement of McMillan's Unfolding Algorithm. In *Formal methods in systems design*, pages 285–310, 2002.

[28] S. B. Furber, D. A. Edwards, and J. D. Garside. AMULET3: a 100 MIPS Asynchronous Embedded Processor. In *Proc. International Conf. Computer Design (ICCD)*, September 2000.

[29] Jun Gu and Ruchir Puri. Asynchronous Circuit Synthesis with Boolean Satisfiability. *IEEE Transactions on Computer-Aided Design*, 14(8):961–973, August 1995.

[30] Scott Hauck. Asynchronous Design Methodologies: An Overview. *Proceedings of the IEEE*, 83(1):69–93, January 1995.

[31] K. Heljanko, V. Khomenko, and M. Koutny. Parallelization of the Petri Net Unfolding Algorithm. In *Proc. of international conference on tools and algorithms for the construction and analysis of system (TACAS'2002)*, pages 371–385. Springer Verlag, LNCS 2280, 2002.

[32] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[33] Change-Hee Hwang and Donk-Ik Lee. A Concurrency Characteristic in Petri Net Unfolding. In *IEICE Transactions on Fundamentals of Electronics Communications and Computer Sciences*, volume e81-a, pages 532–539, April 1998.

[34] N. Karaki, T. Nanmoto, H. Ebihara, S. Utsunomiya, S. Inoue, and T. Shimoda. A Flexible 8b Asynchronous Microprocessor based on Low-Temperature Poly-Silicon TFT Technology. In *International Solid State Circuits Conference*, February 2005.

[35] Joep Kessels, Torsten Kramer, Gerrit den Besten, Ad Peeters, and Volker Timm. Applying Asynchronous Circuits in Contactless Smart Cards. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 36–44. IEEE Computer Society Press, April 2000.

[36] V Khomenko. *Clp Documentation and User Guide*, June 2002.

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

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

[39] 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.

[40] 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.

[41] V. Khomenko, M. Koutny, and A. Yakovlev. Logic Synthesis Avoiding State Space Explosion. Technical Report CS-TR-813, School of Computing Science, University of Newcastle upon Tyne, 2003.

[42] V. Khomenko, M. Koutny, and A. Yakovlev. Logic Synthesis for Asynchronous Circuits Based on Petri Net Unfoldings and Incremental SAT. In *Int. Conf. on Application of Concurrency to System Design*, pages 16–25. IEEE Computer Society Press, June 2004.

[43] 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.

[44] Victor Khomenko, Maciej Koutny, and Walter Vogler. Canonical Prefixes of Petri Net Unfoldings. In *Computer Aided Verification, 14th International Conference (CAV 2002), Copenhagen, Denmark, July 27-31, 2002 / E. Brinksma, K. Guldstrand Larsen (Eds.)*, pages 1–582pp. Springer Verlag, LNCS 2404, September 2002.

[45] D. J. Kinniment, B. Gao, A. Yakovlev, and F. Xia. Towards asynchronous A-D conversion. In *Proc. International Symposium on Advanced Research in Asyn-*

chronous Circuits and Systems, pages 206–215. IEEE Computer Society Press, 1998.

[46] Michael Kishinevsky, Alex Kondratyev, Alexander Taubin, and Victor Varshavsky. *Concurrent Hardware: The Theory and Practice of Self-Timed Design*. Series in Parallel Computing. John Wiley & Sons, 1994.

[47] A. Kondratyev, J. Cortadella, M. Kishinevsky, L. Lavagno, A. Taubin, and A. Yakovlev. Identifying State Coding Conflicts in Asynchronous System Specifications Using Petri Net Unfoldings. In *Int. Conf. on Application of Concurrency to System Design*, March 1998.

[48] A. Kondratyev, J. Cortadella, M. Kishinevsky, E. Pastor, O. Roig, and A. Yakovlev. Checking Signal Transition Graph implementability by symbolic BDD traversal. In *Proc. European Design and Test Conference*, pages 325–332, Paris, France, March 1995.

[49] Alex Kondratyev, Jordi Cortadella, Michael Kishinevsky, Luciano Lavagno, and Alexander Yakovlev. Logic Decomposition of Speed-Independent Circuits. *Proceedings of the IEEE*, 87(2):347–362, February 1999.

[50] Alex Kondratyev, Michael Kishinevsky, and Alex Yakovlev. Hazard-Free Implementation of Speed-Independent Circuits. *IEEE Transactions on Computer-Aided Design*, 17(9):749–771, September 1998.

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

[52] L. Lavagno, C. Moon, R. Brayton, and A. Sangiovanni-Vincentelli. Solving the state assignment problem for signal transition graphs. In *Proc. ACM/IEEE Design Automation Conference*, pages 568–572. IEEE Computer Society Press, June 1992.

[53] Luciano Lavagno and Alberto Sangiovanni-Vincentelli. *Algorithms for Synthesis and Testing of Asynchronous Circuits*. Kluwer Academic Publishers, 1993.

[54] Ch. Ykman-Couvreur B. Lin and H. DeMan. *ASSASSIN: A synthesis system for asynchronous control circuits*, 1995.

[55] K.-J. Lin and C.-S. Lin. Automatic Synthesis of Asynchronous Circuits. In *Proc. ACM/IEEE Design Automation Conference*, pages 296–301. IEEE Computer Society Press, 1991.

[56] K.-J. Lin and C.-S. Lin. On the Verification of State-Coding in STGs. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 118–122. IEEE Computer Society Press, November 1992.

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

[58] A Madalinski. *ConfRes Documentation and User Guide.* University of Newcastle upon Tyne, 2003.

[59] 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.

[60] 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.

[61] A. Madalinski, A. Bystrov, V. Khomenko, and A. Yakovlev. Visualization and Resolution of Coding Conflicts in Asynchronous Circuit Design. *IEE Proceedings, Computers and Digital Techniques, Special Issue on Best Papers from DATE03*, 150(5):285–293, 2003.

[62] A. Madalinski, A. Bystrov, and A. Yakovlev. A software tool for State Coding Conflict Detection by Partial Order Techniques. In *1st UK ACM SIGDA Workshop on Design Automation*, Imperial College London, 2001.

[63] A. Madalinski, A. Bystrov, and A. Yakovlev. ICU: A tool for Identifying State Coding Conflicts using STG unfoldings. Technical Report CS-TR 773, Department of Computing Science, University of Newcastle upon Tyne, December 2002.

[64] A. Madalinski, A. Bystrov, and A. Yakovlev. Visualisation of coding conflicts in asynchronous circuit design. Technical Report CS-TR 768, Department of Computing Science, University of Newcastle upon Tyne, April 2002.

[65] A. Madalinski, A. Bystrov, and A. Yakovlev. Visualisation of Coding Conflicts in Asynchronous Circuit Design. In *IWLS-02 IEEE/ACM 11th International Workshop on Logic and Synthesis*, June 2002.

[66] A. Madalinski, A. Bystrov, and A. Yakovlev. Visualisation of Coding Conflicts in Asynchronous Circuit Design. In *Proceedings of the 12th UK Asynchronous Forum.* School of Computing, Information Systems and Maths, South Bank University, London, June 2002.

[67] Alain J. Martin. Compiling Communicating Processes into Delay-Insensitive VLSI Circuits. *Distributed Computing*, 1(4):226–234, 1986.

[68] Alain J. Martin. Programming in VLSI: From Communicating Processes to Delay-Insensitive Circuits. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*, UT Year of Programming Series, pages 1–64. Addison-Wesley, 1990.

[69] K. L. McMillan. *Symbolic Model Checking: an approach to the state explosion problem.* PhD thesis, 1992.

[70] K. L. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *Proc. International Workshop on Computer Aided Verification*, pages 164–177, July 1992.

[71] T. Miyamoto and S. Kumagai. An Efficient Algorithm for Deriving Logic Functions of Asynchronous Circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems.* IEEE Computer Society Press, March 1996.

[72] Charles E. Molnar, Ting-Pien Fang, and Frederick U. Rosenberger. Synthesis of delay-insensitive modules. In Henry Fuchs, editor, *1985 Chapel Hill Conference on Very Large Scale Integration*, pages 67–86. Computer Science Press, 1985.

[73] David E. Muller and W. S. Bartky. A Theory of Asynchronous Circuits. In *Proceedings of an International Symposium on the Theory of Switching*, pages 204–243. Harvard University Press, April 1959.

[74] Tadao Murata. Petri Nets: Properties, Analysis and Applications. In *Proceedings of the IEEE*, pages 541–580, April 1989.

[75] Chris Myers. *Asynchronous Circuit Design.* John Wiley & Sons, 2001.

[76] M. Nielsen, G. Plotkin, and G. Winskel. Petri Nets, Event Structures and Domains, Part I. *Theor. Computer Science*, 13(1):85–108, January 1980.

[77] Steven M. Nowick and David L. Dill. Automatic Synthesis of Locally-Clocked Asynchronous State Machines. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 318–321. IEEE Computer Society Press, November 1991.

[78] E. Pastor, J. Cortadella, O. Roig, and A. Kondratyev. Structural Methods for the Synthesis of Speed-Independent Circuits. In *Proc. European Design and Test Conference*, pages 340–347. IEEE Computer Society Press, March 1996.

[79] Enric Pastor. *Structural Methods for the Synthesis of Asynchronous Circuits from Signal Transition Graphs.* PhD thesis, Univsitat Politècnia de Catalunya, February 1996.

[80] Enric Pastor and Jordi Cortadella. Polynomial Algorithms for Complete State Coding and Synthesis of Hazard-free Circuits from Signal Transition Graphs. Technical report, UPC/DAC Report No RR-93/17, 1993.

[81] Enric Pastor, Jordi Cortadella, Alex Kondratyev, and Oriol Roig. Structural Methods for the Synthesis of Speed-Independent Circuits. *IEEE Transactions on Computer-Aided Design*, 17(11):1108–1129, November 1998.

[82] Enric Pastor, Oriol Roig, Jordi Cortadella, and Rosa M. Badia. Petri net Analysis Using Boolean Manipulation. In *15th International Conference on Application and Theory of Petri Nets*, June 1994.

[83] Ad Peeters. The Asynchronous Bibliography Database. http://www.win.tue.nl/async-bib/, 2003.

[84] Carl Adam Petri. *Kommunikation mit Automaten.* Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962.

[85] L. Pomello, G. Rozenberg, and C. Simone. A survey of equivalence notions for net based systems. *Lecture Notes in Computer Science; Advances in Petri Nets 1992*, 609:410–472, 1992.

[86] W. Reisig and G. Rozenberg. Informal Introduction to Petri Nets. *Lecture Notes in Computer Science: Lectures on Petri Nets I: Basic Models*, 1491, 1998.

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

[88] P. A. Riocreux, L. E. M. Brackenbury, M. Cumpstey, and S. B. Furber. A Low-Power Self-Timed Viterbi Decoder. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 15–24. IEEE Computer Society Press, March 2001.

[89] L. Y. Rosenblum and A. V. Yakovlev. Signal Graphs: from Self-Timed to Timed Ones. In *Proceedings of International Workshop on Timed Petri Nets*, pages 199–207, Torino, Italy, July 1985. IEEE Computer Society Press.

[90] H. Saito, A. Kondratyev, J. Cortadella, L. Lavagno, and A. Yakovlev. What is the cost of delay insensitivity? In *Proc. 2nd Workshop on Hardware Design and Petri Nets (HWPN'99) of the 20th Int. Conf. on Application and Theory of Petri Nets (PN'99), 21 June 1999, Williamsburg, VA*, pages 169–189, 1999.

[91] Charles L. Seitz. Asynchronous machines exhibiting concurrency, 1970. Record of the Project MAC Concurrent Parallel Computation.

[92] Charles L. Seitz. Self-Timed VLSI Systems. In Charles L. Seitz, editor, *Proceedings of the 1st Caltech Conference on Very Large Scale Integration*, pages 345–355, Pasadena, CA, January 1979. Caltech C.S. Dept.

[93] A Semenov. *Verification and Synthesis of Asynchronous Control Circuits using Petri Net Unfolding*. PhD thesis, University of Necastle upon Tyne, 1997.

[94] Alex Semenov, Alexandre Yakovlev, Enric Pastor, Marco Peña, and Jordi Cortadella. Synthesis of Speed-Independent Circuits from STG-unfolding Segment. In *Proc. ACM/IEEE Design Automation Conference*, pages 16–21, 1997.

[95] Alex Semenov, Alexandre Yakovlev, Enric Pastor, Marco Peña, Jordi Cortadella, and Luciano Lavagno. Partial Order Based Approach to Synthesis of Speed-Independent Circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 254–265. IEEE Computer Society Press, April 1997.

[96] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical report, U.C. Berkeley, May 1992.

[97] S. Sokolov and A. Yakovlev. Clock-less circuits and system synthesis. *IEE Proceedings, Computers and Digital Techniques, Special Issue on Embedded Microelectronic Systems*. (To appear).

[98] Jens Sparsø and Steve Furber, editors. *Principles of Asynchronous Circuit Design: A Systems Perspective*. Kluwer Academic Publishers, 2001.

[99] N. Starodoubtsev, S. Bystrov, M.Goncharov, I. Klotchkov, and A. Smirnov. Towards Synthesis of Monotonic Asynchronous Circuits from Signal Transition Graphs. In *Int. Conf. on Application of Concurrency to System Design*, pages 179–188. IEEE Computer Society Press, June 2001.

[100] Ivan E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.

[101] Hiroaki Terada, Souichi Miyata, and Makoto Iwata. DDMP's: Self-Timed Super-Pipelined Data-Driven Multimedia Processors. *Proceedings of the IEEE*, 87(2):282–296, February 1999.

[102] S. H. Unger. *Asynchronous Sequential Switching Circuits.* Wiley-Interscience, John Wiley & Sons, Inc., New York, 1969.

[103] P. Vanbekbergen, F. Catthoor, G. Goossens, and H. De Man. Optimized Synthesis of Asynchronous Control Circuits form Graph-Theoretic Specifications. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 184–187. IEEE Computer Society Press, 1990.

[104] P. Vanbekbergen, B. Lin, G. Goossens, and H. de Man. A Generalized State Assignment Theory for Transformations on Signal Transition Graphs. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 112–117. IEEE Computer Society Press, November 1992.

[105] Victor I. Varshavsky, editor. *Self-Timed Control of Concurrent Processes: The Design of Aperiodic Logical Circuits in Computers and Discrete Systems.* Kluwer Academic Publishers, Dordrecht, The Netherlands, 1990.

[106] Walter Vogler and Ralf Wollowski. Decomposition in asynchronous circuit design. *Concurrency and Hardware Design — Advances in Petri Nets, Volume 2549 of Lecture Notes in Computer Science / J. Cortadella, A. Yakovlev, G. Rozenberg (Eds.)*, pages 152–190, November 2002.

[107] Ingo Wegener. *The complexity of Boolean functions.* John Wiley & Sons, Inc., 1987.

[108] A Yakovlev. *Design and Implementation of Asynchronous Interface Protocols.* PhD thesis, Leningrad Electrical Engineering Institute, 1982. (in Russian).

[109] A. Yakovlev and A. Petrov. Petri nets and Asynchronous Bus Controller Design. Proc. of the International Conference on Application and Theory of Petri Nets, pages 244–263, 1990.

[110] Chantal Ykman-Couvreur and Bill Lin. Optimised State Assignment for Asynchronous Circuit Synthesis. In *Asynchronous Design Methodologies*, pages 118–127. IEEE Computer Society Press, May 1995.

[111] Chantal Ykman-Couvreur, Bill Lin, Gert Goossens, and Hugo De Man. Synthesis and optimization of asynchronous controllers based on extended lock graph theory. In *Proc. European Conference on Design Automation (EDAC)*, pages 512–517. IEEE Computer Society Press, February 1993.