University of Newcastle upon Tyne

Department of Computing Science

Verification and Synthesis of Asynchronous Control Circuits Using Petri Net Unfoldings.

by

Alexei Semenov

PhD thesis

July 1997

Contents

List of Figures

List of Tables

Acknowledgements

I would like to express my gratitude to my supervisor, Alex Yakovlev, for introducing me to the exciting and challenging world of asynchronous circuit design. His enthusiasm and devotion to the asynchronous world kept me going from the very first day when I learned about the existence of asynchronous circuits till the very end. Special mentioning deserves his patience and on-going interest in discussing some of my theories (often not very good ones, in retrospect) and wholehearted participation in the development of most of my papers. He introduced me to top-class researchers in the asynchronous area, for which I am also thankful. He and his family (Maria and Greg) made my life in Newcastle a pleasant experience.

This work would have not been possible had I not met and discussed it with the "mediterranean" research group consisting of Luciano Lavagno, Jordi Cortadella, Enric Pastor, Oriol Roig and Marco Peña. Their comments helped me to develop and shape this work. A special thank you goes to the members of their families who's companies I enjoyed while visiting Berkeley and Barcelona.

Of course, this work would never be completed without constant support from my family and friends, patiently bearing my "coma-like" periods during which I was working towards numerous deadlines. I value greatly the moral input from Olga, Anton and Andrew who once again proved that friendship is indeed delay-insensitive. I would like to thank my officemates from B338, Martin Hesketh and Frode Sandnes for being such a wonderful company; my particular appreciation is to Rob Allen who, in addition, managed to read through the whole thesis.

Last, but not least, I would like to acknowledge financial help from the Research Committee of the University of Newcastle upon Tyne, which provided my Research Studentship grant, and the Committee of Vice-Chancellors and Presidents, which provided the Overseas Research Studentship Award.

Abstract

Design of asynchronous control circuits has traditionally been associated with application of formal methods. Event-based models, such as Petri nets, provide a compact and easy to understand way of specifying asynchronous behaviour. However, analysis of their behavioural properties is often hindered by the problem of exponential growth of reachable state space. This work proposes a new method for analysis of asynchronous circuit models based on Petri nets. The new approach is called PN-*unfolding segment*. It extends and improves existing Petri nets unfolding approaches. In addition, this thesis proposes a new analysis technique for Signal Transition Graphs along with an efficient verification technique which is also based on the Petri net unfolding. The former is called *Full State Graph*, the latter - STG-*unfolding segment*. The boolean logic synthesis is an integral part of the asynchronous circuit design process. In many cases, even if the verification of an asynchronous circuit specification has been performed successfully, it is impossible to obtain its implementation using existing methods because they are based on the reachability analysis. A new approach is proposed here for automated synthesis of speed-independent circuits based on the STG-unfolding segment constructed during the verification of the circuit's specification. Finally, this work presents experimental results showing the need for the new Petri net unfolding techniques and confirming the advantages of application of partial order approach to analysis, verification and synthesis of asynchronous circuits.

Chapter 1

Introduction

Asynchronous (or self-timed) circuits and systems have attracted increasing attention from the research community in recent years. The inherent concurrency in their operation and the absence of the requirement for a pre-determined settling period, the clock cycle, means that these systems reflect more naturally the processes happening in real life.

By making the assumption of a synchronous mode of operation, designers can abstract from the problem of tracking of all intermediate states of the system. It can be safely assumed that the clock period is chosen to be long enough for the signals to settle to their new values. Any feedback is cut off to prevent the changing outputs from affecting the inputs. The arrival of a new clock pulse triggers the transition process to the next state of the system.

In an asynchronous circuit there is no such a "start-stop" mechanism. Any change of signals may cause a transition of the system into the next state. This makes asynchronous circuits harder to design. In addition, the majority of today's designers are used to creating systems and circuits in the synchronous domain.

Since asynchronous circuits are more complex to design, most of the design methodologies assume the use of some formal method. For some time this was an additional obstacle to adopting the self-timing concept. The need for formal verification in circuit design has now been widely recognised in the synchronous domain as well. This places asynchronous circuits on a par with their clocked counterparts.

The asynchronous community has demonstrated that it is possible to design fully functional circuits beyond trivial examples. Several microprocessors have been designed to date. Examples of microprocessor designs can be found in works reported by the Caltech [43], Titech [55] and Manchester [25, 24, 1] research groups. In Manchester, the AMULET group designed an instruction-level compatible asynchronous version of the ARM6 microprocessor whose performance characteristics are comparable to those of the synchronous one. In addition, Philips reported a design of an asynchronous error correction chip [6, 5] which demonstrated 80% savings in the power consumption.

In order to cater for the growing need and complexity of asynchronous circuit design, new methods need to be developed. The conventional approach to circuit verification and synthesis proves to be unable to deal with relatively large examples. The purpose of this work is to introduce a novel technique which will advance the applicability of formal methods. It also serves as the basis for further research.

1.1 Motivation

The use of the asynchronous paradigm in circuit design has been argued to provide certain advantages in circuit performance. The most commonly cited are discussed below.

Average case performance The clock cycle of every synchronous circuit is determined by the longest propagation delay of the circuit. The rate of the clock signal must accommodate the settling times for the longest possible operation. Therefore, during a faster operation some of the parts of the circuit will stay idle while the clock signal is due to switch. To overcome this problem circuit designers need to come up with elaborate scheduling and re-timing schemes.

In an asynchronous circuit, every part works at its own pace. As soon as the data has been processed by one part, the next part is informed and may start working with the data. Thus the overall cycle time, i.e. the average time between the completion of two sequential operations, will be the average of the execution times of all operations.

Absence of clock skew The existence of a propagation delay in the wires of a chip means that the signal change may arrive at two ends of a forking wire at different times. This phenomenon is known as the *clock skew* problem. To guarantee that all operational blocks work synchronously the designer needs to make sure that the clock signal is received by each block at exactly the same time. However, with growing clock rates it becomes increasingly difficult to guarantee the absence of clock skew. In addition, clock wiring has been reported to take up to 60% of all wiring in the chip.

By choosing an asynchronous implementation the designer escapes the clock skew problem and the associated routing problem.

Low power consumption During the operation of a synchronous circuit the clock signal is propagated to every operational block of the circuit even if this block is not used in a particular computation at all. Thus the power is spent on driving the clocked inputs of the gates which do not perform any useful actions.

Each part of an asynchronous circuit operates only when signalled to commence the operation, after the data has been prepared on the inputs of this part. Therefore, until such a request is produced, this part of the circuit does not consume any power at all.

High modularity Synchronous circuits are subject to precise synchronisation between the modules comprising them. Redesigning of any module requires meeting heavy restrictions on the execution times to ensure the correct synchronisation.

Any part of an asynchronous circuit can be redesigned at will. The new module must, of course, conform to the same interface protocol as the module that is being replaced. However, the speed at which the new module operates is irrelevant allowing easy upgrading of asynchronous circuits.

Reduced EMI Electro-magnetic emission generated by synchronous circuits causes interference with other equipment. Much of this interference is attributed to the clock signal which produces a steady peak in the spectrum on the frequency at which the transistors are switched.

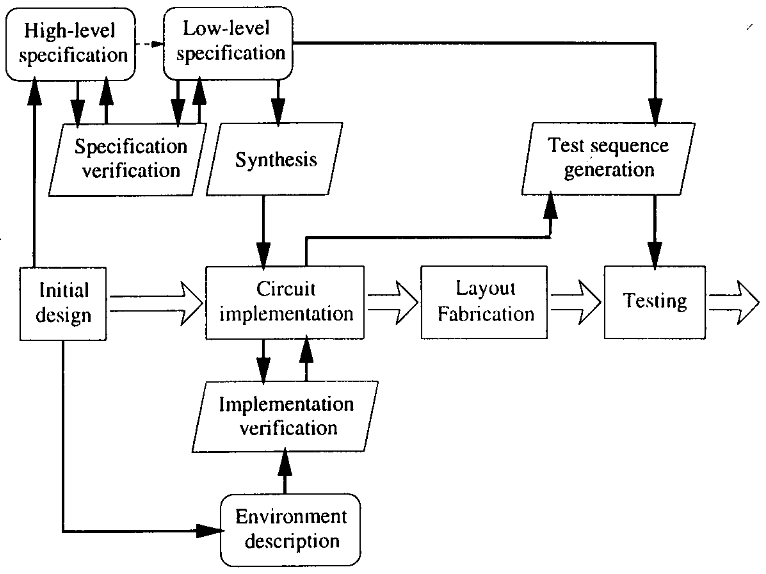


Figure 1.1: Overall design process.

The transistor switching frequency in an asynchronous circuit depends on the data which is being processed by the circuit. Thus the spectrum is smoother and the peak values are lower.

1.2 Design Cycle

The work presented in this thesis follows a well established circuit design path. The overall design process is illustrated in Figure 1.1.

At the first stage of the design process a designer's idea is expressed as a high-level specification of the future circuit. This specification is then checked for correctness, i.e. that the specification behaves according to the designer's requirements.

At the next stage, the high-level description is transformed into a low-level specification of the circuit; this low-level specification now includes signals which will implement the circuit. At this stage the specification is again checked for correctness. This time the requirements also include conditions related to the binary nature of the circuit implementation.

Once verified, the specification is submitted to the synthesis procedure which produces a set of boolean functions, one for each output signal. The synthesis procedure deals with such things as optimisation of the boolean logic functions and the technology mapping (implementing the circuit using a particular type of gates).

If the synthesis procedure uses some formal method and is automated, then there is no need to verify the implementation. However, many designers still use ad hoc techniques to produce an implementation from a high-level "blackboard" specification. In this case, the implementation needs to be verified. To do so, the model of the implementation is composed with the model of the environment obtained from the initial specification. The composed model is then verified for correctness. This process is somewhat similar to the debugging process in software development.

The third stage deals with the actual physical production of the circuit. This stage includes placing and routing of the circuit elements on the actual silicon. A laid out circuit is sent to the manufacturing process.

At the last stage, the manufactured circuits are tested for the absence of faults. The testing procedure usually targets faults which could be introduced during the manufacturing of the chip. In order to do so, test sequences are required which consist of the sequences of input changes along with the outputs' checkpoints. These are generated using the low-level specification and the implementation produced by the synthesis process.

The process outlined above has been widely accepted for the design of asynchronous circuits. It is common to employ some formal model, such as Petri nets or process algebras, in the asynchronous circuit design. A variety of analysis methods can be used for reasoning about the behavioural properties of circuit models. A growing body of current research is aimed at the efficient automation of the design process; this works forms a part of this research.

1.3 Position and Contributions of This Work

The main objective of this work is to demonstrate the application of the partial order approach in the design of asynchronous circuits. Unlike the state graph approach, this method represents the concurrency of an asynchronous system in its true form. Thus in many cases the exponential state explosion, usually associated with the state graph approach, is avoided.

This work tackles three major problems in the asynchronous circuit design:

- *Specification verification*, where the general Petri net (PN) unfolding technique is adapted for the verification of PN and/or Signal Transition Graph (STG) models of the "would-be-circuits".
- *Implementation verification*, where a PN or STG model is constructed for an already designed circuit and then this model is verified along with the model of the circuit's environment.
- *Boolean logic synthesis*, where the boolean logic implementation is obtained from the STG-unfolding segment instead of constructing the state graph.

This work proposes an automated approach which takes a specification of an asynchronous circuit in the form of an STG, verifies the specification and, in the case of successful verification, produces an implementation in the form of boolean logic equations. If the specification failed to pass the verification stage, the offending behaviour is reported and the specification can be corrected. In addition, techniques proposed in this work can be used for verification of existing circuits. The main contributions of this work are as follows:

- The existing PN-unfolding method, suggested by McMillan, is examined and adapted for the verification of relations between transitions of the original PN. The problem of redundancy in the truncated PN-unfolding is approached. A new termination condition is suggested for a wide class of non-autoconcurrent PNs. This condition avoids construction of redundant copies of the transition instances in the unfolding, which results in gains in speed and size of the segment.
- New algorithms, based on the PN-unfolding segment method, are suggested for the verification of the behavioural properties of asynchronous circuits and systems. The

verification of these properties allows the designer to find errors in the design long before the implementation stage is reached.

- A new concept of the Full State Graph (FSG) is introduced, which adequately captures the behaviour of an arbitrary STG. A partial order based approach for the FSG analysis, called the STG-unfolding segment, is suggested. This method is an extension of the PN-unfolding segment approach to STG analysis and takes into account the signal interpretation of transitions in an STG.

- A new method for the automated synthesis of asynchronous circuits from STGs is suggested. This method identifies fragments of the STG-unfolding segment from which the boolean logic implementation is obtained for each signal. In addition, the new synthesis method employs an approximation technique which uses the structural information available from the segment.

- A new algorithm which applies the unfolding technique to contextual nets is proposed. This algorithm takes advantage of contextual dependency and demonstrates significant savings in time and space compared to the analysis of PN models of existing circuits.

- An algorithm is suggested for obtaining a better variable ordering for the analysis and synthesis methods which use Binary Decision Diagrams for the representation of state space.

Results obtained in the course of this research also contributed to works on verification of STGs [76, 77] and synthesis of speed-independent circuits from their STG specifications [80, 81]. In addition, the methods and approaches developed here were used to explore the analysis of timed models [78] and for the analysis of realistic examples of a microprocessor [75] and a communication mechanism [79].

This work leaves out such areas of the asynchronous circuit design process as testing and technology mapping. It does, however, lay the basis for future research in these areas and demonstrates that the partial order methods achieve results which compare favourably to those of existing powerful methods.

1.4 Organisation of Thesis

This thesis is organised as follows:

- Chapter 2 briefly outlines the research work done in the area of asynchronous circuit design.

- Chapter 3 introduces Petri nets (PNs) and describes methods for their analysis. This chapter also introduces Signal Transition Graphs (STGs) which are used for the specification of asynchronous circuits.

- Chapter 4 suggests a new method for the analysis of PNs based on McMillan's truncated unfolding. It presents a comparison between the original version of the PN-unfolding algorithms and the suggested experimental technique and illustrates the performance of the approach on real life examples. The new method is also applied to the analysis of PN models of asynchronous circuits.

- Chapter 5 introduces the Full State Graph (FSG) and proposes the application of the new unfolding technique to the analysis of STGs. The suggested STG-unfolding segment analysis is applied to a set of existing benchmarks and its performance is discussed.
- Chapter 6 describes the application of the STG-unfolding segment to the synthesis of asynchronous circuits. The experimental results demonstrate that the new synthesis method extends the application of automated synthesis procedures.
- Chapter 7 illustrates how the application of the unfolding technique may assist in solving other problems in asynchronous circuit design. The first problem is related to the analysis of PNs with self-loops, which are often found in PN models of circuits, and deals with Contextual nets; the second application of PN-unfolding deals with the variable ordering for Binary Decision Diagrams.
- Chapter 8 concludes the thesis, summarising the results presented in this work and outlining the areas for future research.

Chapter 2

Previous and Related Work

The purpose of this chapter is to give a brief account of the work done to date on asynchronous circuit design.

An asynchronous circuit can be viewed as a set of gates interconnected with wires so that no two outputs can be connected together. Asynchronous circuits assume that there is no clock signal. However, the behaviour of the circuit components cannot be considered without taking the timing domain into account. Every event that takes place in the circuit can be characterised by the time it has taken and the place where this event occurred. In addition, several signalling protocols exist which ensure that the signal levels are interpreted correctly. The interpretation of the time domain and the signalling protocols lead to the two taxonomies that define existing design methodologies:

- Delay taxonomy, which defines the model chosen to represent the duration of every event occurring in the circuit; and
- Protocol taxonomy, which defines the method that is used to pass data from one part of the circuit to another.

Combinations of different types of delays and protocols produced a variety of different design methodologies for asynchronous circuits. Each methodology established certain requirements on the environment in which a circuit operates. It was soon realised that asynchronous circuits were too complex to be designed by hand. This called for the use of formal methods in asynchronous design.

The goal of formal methods application is, on one hand, to provide the designer with a somewhat unified way of describing the desired behaviour. On the other hand, formal methods allow reasoning about the global properties of the behaviour, e.g. establishing if a system ever reaches a certain state. An attempt to find a rarely encountered state by means of conventional simulation may take a long time and cannot be done with 100% certainty.

First this chapter examines the delay and the protocol taxonomies. The discussion about taxonomies is followed by a brief account of the existing methodologies and the formal models and methods used in asynchronous circuit design.

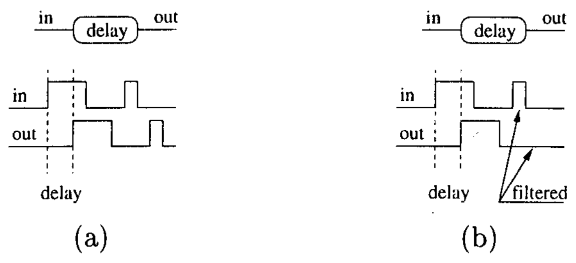


Figure 2.1: Illustration of pure (a) and inertial (b) delay models.

2.1 Asynchronous Circuits Taxonomies

2.1.1 Delay taxonomy

The delay taxonomy draws a distinction between different models of delays taking into account their duration and behaviour.

Bounded and unbounded delays One of the major properties of the delay is its duration. A delay is said to be **bounded** if its upper and lower bounds are known. Alternatively, a delay is called **unbounded** if its upper and lower bounds are unknown, however it is known that the delay is *finite* and positive.

Pure and inertial delays Another characteristic of the delay is its ability to propagate signals. A delay is called **pure** if a change of any length in the input signal causes a change on the output after a certain amount of time. A delay is called **inertial** if the changes in the input signals are not propagated if they are shorter than a certain length. The difference between these two delay models is illustrated in Figure 2.1. The pure delay element in Figure 2.1(a) simply delays the changes in the output signal, whereas the inertial delay element (Figure 2.1(b)) filters out short pulses but delays the the long ones.

Delay cite Any circuit can be viewed as a set of gates interconnected by wires. The cite of the delay in a circuit is crucial to the design methodology. A delay is called **gate delay** if it occurs inside a gate, reflecting the time taken to compute the output signal change. A delay is called **wire delay** if it happens in the wire and reflects the wire propagation times of the signals.

2.1.2 Protocol signalling taxonomy

The protocol signalling taxonomy draws a distinction between the ways in which data is transfered from one part of the circuit to another.

Dual-rail and bundled data signalling The absence of the clock means that there is no way to fix a moment in time at which the signal level can be sampled. For example, registering a low level of some signal twice may mean two sequential 0s. Alternatively, it may be a 0 followed by a 1, but the second value was sampled too early, i.e. before the transition was completed.

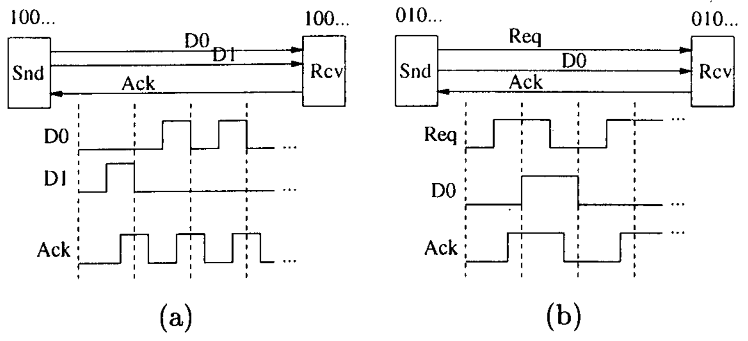


Figure 2.2: Illustration of dual-rail (a) and bundled data (b) signalling.

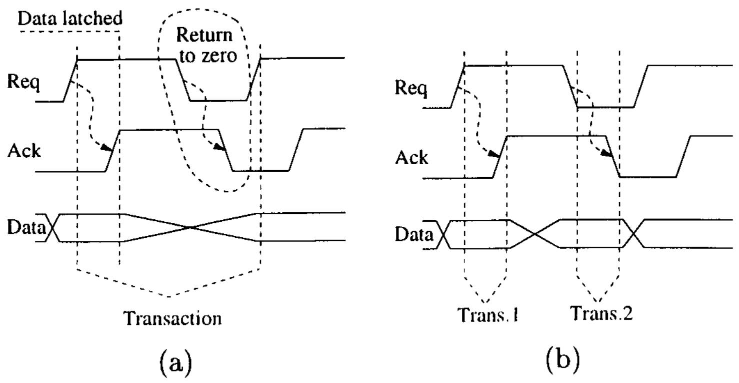


Figure 2.3: Illustration of four-phase (a) and two-phase (b) protocols.

To ensure the correct data transmission between two parts of the system (a sender and a receiver) a **dual-rail** data signalling protocol was introduced, illustrated in Figure 2.2(a). Each bit uses two wires, e.g. $D0$ and $D1$ with the appropriate encoding using combinations of high an low levels of both wires. One combination, e.g. 01 ($D1D0$), represents a "0" and a complimentary one, 10, represents a "1". One of the combinations 00 or 11 is used as a *spacer*, a special symbol which separate sequential bits. The fourth combination is considered to be illegal and must never appear. A single wire is used in the reverse direction to indicate that the receiver has registered the changes on all data wires and is ready to accept the next piece of data.

A **bundled data** signalling protocol assumes that each data bit is represented by one wire, but there exists a pair of request and acknowledgement wires between the sender and the receiver. This protocol is illustrated in Figure 2.2(b). As soon as the sender sets the levels on the data wires, it sends a request signal. Upon receiving the request signal, the receiver processes the data on its inputs and returns an acknowledgement after which the system enters the next cycle. An important assumption in this protocol is that the propagation delay in the request wire must be greater than the longest delay in any of the data wires.

Four-phase and two-phase signalling This distinction comes from the fact that the signals have two levels. A **four-phase** signalling protocol assumes that only one edge (change of the signal from high to low or vice versa) indicates an occurrence of some event. Therefore,

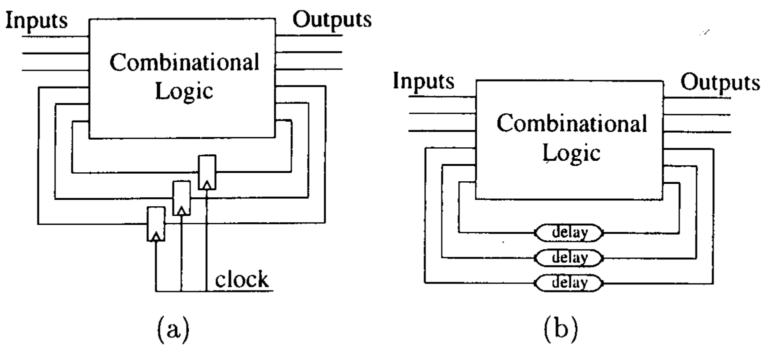


Figure 2.4: Fundamental mode synchronous (a) and asynchronous (b) circuits.

there exists a sequence of changes in the signal levels, called the "return-to-zero" phase, when the signals are reset to their original levels. Since only one level of the signal represents an event, this protocol is also called *level signalling* protocol. This protocol is illustrated in Figure 2.3(a).

In the **two-phase** signalling protocol, every change in the signal level indicates an event. This protocol is illustrated in Figure 2.3(b). In this protocol there is no phase resetting the signals to their original levels.

2.2 Design Methodologies

2.2.1 Huffman Fundamental Mode Circuits

The operation of an asynchronous circuit in the Huffman fundamental mode [31] is similar to the operation of a circuit in the synchronous mode (see Figure 2.4(a)). The Huffman fundamental mode assumes that the circuit consists of a combinational logic block and a set of feedback wires. However, these circuits assume that a bounded delay element is inserted instead of the latches breaking the feedback wires (Figure 2.4(b)). The changes in the outputs appear at the inputs of the circuit separated in time, thus giving the necessary delay for settling of the signals inside the combinational logic block. Huffman fundamental mode circuits also require that only one input changes at a time.

The circuit is specified using a *Finite State Machine* (FSM) flow table [89] which is then binary encoded and implemented. The flow table specification is usually minimised to reduce the complexity of the encoding algorithms and to reduce the number of the variables needed to encode the states of the FSM.

The implementation in the Huffman mode must be free from critical races [90]. A **race** is a simultaneous change of more than one signal during a transition from one state of the system to another. A race is called **critical** if the behaviour of the circuit depends on the order in which the racing signals change.

The operation in the Huffman fundamental mode imposes very strict requirements on the implementation. A number of approaches were suggested to relax these restrictions. They are based on the observation that not all multiple changes in the input signals lead to critical races; hence the other two modes can be considered: *multiple input change* (MIC) mode, where several inputs can change at the same time; and *unrestricted input change* (UIC) mode,

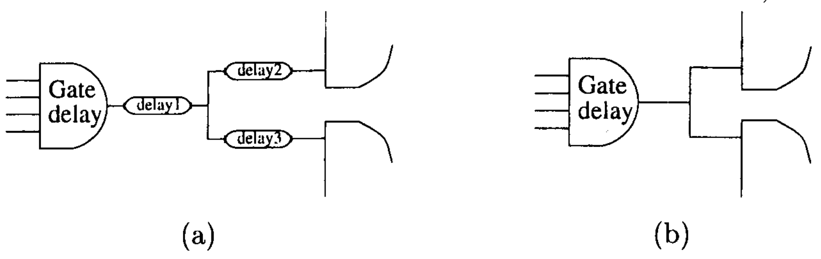


Figure 2.5: Illustration of Delay-Insensitive (a) and Speed-Independent (b) circuits.

where no restrictions are imposed on the input changes whatsoever. However, designing MIC and UIC circuits proved to be complex. A solution was suggested in the form of the *burst mode* circuits [57] which assume that:

- the inputs can change in bursts (sets of simultaneously changing signals);
- no burst can be a subset of another burst.

Furthermore, the burst mode can be extended into the *extended burst mode* [104] which allows *don't-cares* on the inputs and *condition signals*. The former allows an input to choose non-deterministically whether or not its value changes in a particular burst. The latter introduces special signals whose levels determine possible advancements in the behaviour of the FSM. There exists a variety of other methodologies implementing fundamental mode circuits such as locally clocked [59, 60] and 3-D state machines [106, 105].

2.2.2 Delay-Insensitive Circuits

Delay-Insensitive (DI) circuits assume that both gate and wire delays are unbounded. The general idea is shown in Figure 2.5(a). As a result of this assumption these circuits are the most robust implementation with respect to the delay changes. Indeed, a DI circuit is insensitive to the variations of the delay values due to the working conditions of the implementation. If a signal gets stuck permanently at a particular level, so called *stuck-at-fault*, then the circuit will stop functioning rather than producing a spurious result, i.e. it will fail safely.

Very few circuits can be designed so that they are completely DI. Therefore, a DI implementation is usually obtained from modules whose behaviour is considered to be DI on their interfaces.

An implementation is usually obtained from a specification in a high-level programming language such as Communicating Sequential Processes (CSP) [30] and CSP-like HDL [45], Tangram [4], trace theory expressions [20] or DI algebras [33]. The known examples of DI circuits include asynchronous microprocessors [43, 9] and Philips error correction chip [6, 5].

2.2.3 Speed-Independent Circuits

The research into Speed-Independent (SI) circuits was pioneered by Muller [53]. SI circuits assume that the gate delay is unbounded whilst the wire delay is negligible with respect to the date delay. An illustration of the delay cite in an SI circuit is given in Figure 2.5(b). Muller introduced a formal model, called *State Transition Diagram* (STD), for representing the behaviour of asynchronous circuits. Each state of an STD is a binary vector representing the signal values. States are connected by the arcs labelled with signal transitions (only one label per arc is allowed). A signal is said to be *stable* in a particular state if it is equal to the value computed by the corresponding logic function under the values given by the vector; otherwise it is called *excited.* Muller showed that the circuit's behaviour can be equivalently described using STDs.

Speed-independence is closely related to the notion of semi-modularity. A circuit is said to be **semi-modular** if every excited signal becomes stable due to the change of its value, i.e. the signal's excitation cannot be removed by another signal. Muller showed that any semi-modular circuit is speed-independent. The synthesis procedure for SI circuits takes an STD as the specification of the future circuit and produces an implementation in the form of boolean logic equations for the circuit's gates. Varshavsky *et. al.* [93] showed that any semi-modular STD can be implemented as an SI circuit using a restricted set of gates: n-input AND-OR-NOT gates (that are able to implement an arbitrary sum-of-product function) or 2-input NAND and 2-input NOR gates with the fanout limited to two gates. Recent works [3, 2] established the conditions for an SI implementation using n-input NAND gates and a Muller C-element.

Specifying an SI circuit in terms of an STD can sometimes be a problem due to the high degree of concurrency. A number of formal models were suggested to be used for specification of SI circuits, e.g. Trace theory [19], Change Diagrams [35] and Signal (Transition) Graphs [73, 13]. These models specify the behaviour in a compact form and then automatically verify and/or synthesise the SI implementation. A more detailed discussion of the existing methods is given in the next section.

Quasi Delay-Insensitive (QDI) circuits were suggested in [44] and assume the unbounded gate and wire delay models, however, they are enriched with *isochronic forks.* An isochronic fork is a forking wire where the difference between the delays of the branches is negligible. The isochronic fork definition does not require the delays between all destinations to be negligible. Thus it is possible to specify an isochronic fork in which the difference between the delays is negligible only for a subset of destinations. QDI circuits with this form of the isochronic fork are not equivalent to SI circuits. However, if all destinations are isochronic, which is usually the case, QDI circuits are equivalent to SI circuits if the wire delay is considered as a part of the gate delay generating the signal.

2.2.4 Micropipelines

Micropipelines were suggested by I. Sutherland in [88]. The general idea is illustrated in Figure 2.6. The backbone of a micropipeline consists of Muller C-elements and Capture-Pass latches. The sender generates a request when the data is ready to be sent down the pipeline. The first stage is ready to accept new data if the previous piece of data has already been latched by the next stage. If ready, the first stage latches the new data and sends a signal acknowledging this latching to the sender. At the same time the first stage sends a request to the next stage to pass on the portion of data held in the latch. Each stage of the micropipeline operates in a similar way regarding the preceding stage as a sender. At the other end of the micropipeline the receiver accepts the arriving data and acknowledges every

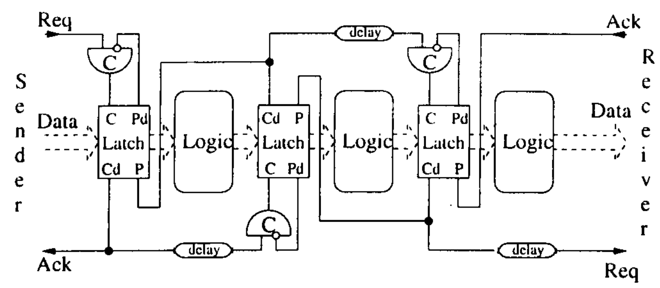


Figure 2.6: Sutherland micropipelines.

portion. In addition, each stage may contain a combinational logic block which performs necessary computations on the data. The control circuitry of the micropipelines is delay insensitive. Introducing the data path requires careful compensation for the propagation time of the signals through the data wires and, where necessary, through the combinational logic.

Micropipelines use bundled data protocol with either two- or four-phase signalling. They have proved to be a powerful design methodology; well-known examples of the micropipelined systems include a FIFO controller [71] and the AMULET microprocessor [24, 23].

2.3 Formal Models

The goal of this section is to give a brief overview of the formal methods employed in asynchronous circuit design methodologies. The motivation behind using formal models in circuit design is to ensure that the implementation is correct. In particular, the implementation must produce the required output signals in response to the input stimuli. Furthermore, an asynchronous circuit must operate without hazards. A *hazard* is an unspecified change of the signal, e.g. a spike. In synchronous design, a spike may occur during the settling period. This spike does not affect the correctness of the implementation but rather its power consumption. In an asynchronous circuit, there is no way to distinguish a spurious spike from a sequence of signal changes. Thus this spike may be registered by a gate and cause the circuit to malfunction. Interested readers are referred to [87, 41] for a thorough review of hazards.

State graph based models In the state graph based methods the specification is given in terms of a finite automaton describing all possible states of the system. If the system has many events that can happen concurrently, then the total number of states in the system may be prohibitively large.

The problem with the size of the specification comes from the fact that any set of concurrent events produces an exponential number of intermediate states, although the state reached at the end is always the same. The use of the burst mode FSM specifications allows a reduction in the size of the specification. In effect, a burst of input and/or output signals captures all interleavings which would be possible had these signals been allowed to change freely. The penalty paid for such a reduction is the requirement for the difference between the moments of signal changes in one burst to be negligible.

The state based models offer a direct route for obtaining the circuit implementations. The states are encoded using binary codes and the truth tables are obtained in a straightforward manner.

Trace based models The trace theory was suggested for the automated verification of SI circuits by Dill [19]. The behaviour of each element of the circuit is described using the trace theory primitives. In addition, the desired behaviour is also specified in terms of trace theory. An element is said to *conform* to its specification if its observable behaviour is equivalent to that of its specification. This suggests the hierarchical verification where an element is substituted with its specification which is often much simpler. This method, however, is more applicable to the verification of the already designed circuits, i.e. the designer must take a trial and error approach if he wishes to implement a particular specification.

Ebergen [20] suggested an approach for the synthesis of DI circuits which is also based on trace theory. This approach uses a top-down design methodology. A future circuit is specified using the trace theory description of its input/output behaviour. The specification is verified for delay insensitivity. Alternatively the specification can be constructed using a restricted grammar which can only produce a DI circuit. Once a circuit is specified it is generated automatically using syntax-directed translation and a predefined table of the implementation primitives.

Josephs [34] takes a similar approach suggesting an algebraic solution to the synthesis of asynchronous circuits. Using a special DI algebra the specification is transformed to the level of the implementation primitives.

The trace based model provides a powerful approach to the automated synthesis of asynchronous circuits. The circuits are hazard free by construction. However, this model does not have provision for the verification of such important properties as a deadlock, i.e. a state from which no further advancement of the system can be made. In addition, implementations produced by a syntax-driven synthesis process are often far from optimal.

High-level description languages High-level description languages specify the system in a similar way to the conventional programming languages. Among the most well-known are Martin's [44] and Brunvand's [10, 8] compilation systems and van Berkel's Tangram language [7, 4]. Most of these methods are based on the theory of Communicating Sequential Processes [30] using a *channel* as the primary communication mechanism between subsystems.

Martin's compilation system used a CSP-like hardware description language whereas van Berkel suggested a completely new language. The approach is, however, similar. The system is specified as a composition of the communicating processes. Once the system is specified, each process is decomposed into simpler processes. At the low level, the communication and synchronisation commands are expanded into a four-phase handshake protocol. The final circuit is obtained after the re-shuffling of transitions and the insertion of state signals to eliminate the ambiguities.

Brunvand's approach is based on a subset of Occam. Similar to the techniques described above, the system is specified as a program. Each statement has a corresponding hardware primitive. The program is directly translated into a set of interconnected primitives. The resulting circuit is often very poor with respect to the area size and the performance. Similar to programming language compilers, this approach uses an optimisation to increase the performance and the area results. The optimisation, called *peephole optimisation*, is based on detecting those parts of the circuit which can be safely substituted by an already optimised fragment with an equivalent behaviour.

Event based models The use of event-based models in asynchronous circuit design was prompted by the difficulties with the state space size for complex behaviours. Instead of the complete enumeration of all states of the system, an event-based formal model specifies events and relations between them. A suitable formal model for this was found in the form of Petri nets (PNs)[1] [65, 67]. PNs provide a simple graphical description of the system with an easy representation of concurrent events or a choice between alternative events. In addition, the set of reachable states can be obtained from a PN using a straightforward algorithm.

PNs do not make any assumptions about the time at which an event occurs. This makes them attractive for asynchronous circuit design. Patil [63] suggested a syntax directed method for the translation of PN specifications of asynchronous systems into implementations.

A number of works [51, 86, 54] use *I-nets* for the specification of asynchronous circuits. An I-net represents the interface behaviour of a circuit using events associated with its inputs and outputs. The initial state of the system is captured by the initial marking. After the specification is completed, an *Interface State Graph* (ISG) is built for this I-net which represents all reachable states of the system. After that an *Encoded Interface State Graph* (EISG) is constructed which takes into account the binary interpretation of the signals on the circuit's interface. The EISG is then used for the generation of truth tables and building an implementation either in the Huffman or burst mode.

Signal Transition Graphs (STGs) were suggested independently in [73] and [13] for the specification, verification and synthesis of self-timed circuits. An STG is a PN where each transition is labelled with a directed signal transition (up or down). An STG specification serves as a low-level description of the future circuit's behaviour. The synthesis process attempts to restore an STD from an STG by building the *reachability graph* representing the set of reachable state of the underlying PN, similar to I-nets. Each state of the obtained reachability graph is assigned with a binary code. Once the binary code assignment is completed the implementation is generated by deriving the truth tables.

A model closely related to the STG model, called *Change Diagrams* (CDs), was suggested in [35]. CDs have two distinctive features. Firstly, they have provision for non-repeatable events using disengagable arcs. Secondly, they allow OR-causality, i.e. they are able to model an event whose happening is induced by any of its causes. A set of algorithms for the verification and automated synthesis of SI circuits was suggested in [35]. A notable feature of these algorithms is that they use the unfolding process to reason about the properties of the specification. Unfortunately, CDs cannot model specifications with non-deterministic choice.

The methods discussed above are geared for the automated verification and generation of implementations from a PN-based specification. A number of works [47, 101] also examined the verification of the already designed circuits. These approaches usually build fragments of PNs for each gate which are then composed together according to the gate net list. The resulting PN is composed with the PN model of the environment and it is verified for errors

[1]See next chapter for definitions from the Petri net theory.

in design and/or hazards.

2.4 Design of Speed-Independent Circuits

SI circuits have a special place in the asynchronous circuit design. Operational conditions for SI circuits require less strict assumptions than those for fundamental mode circuits. The design process of a fundamental mode circuit is similar to the design process of a synchronous circuit. This, on one hand, makes it easier for a synchronous circuit designer to understand the new design methodology. On the other hand, an accurate delay estimation is required for the correct operation of the circuit. The delay estimation must take into account all possible conditions affecting the circuit. SI circuits are built to operate independently of the gate delay. Thus, if a circuit is SI, then this circuit will operate correctly under any environmental conditions that may affect the gate delay. This also makes SI circuits very robust to the manufacturing technology parameters. Therefore, an SI design can be ported to different technologies without major modifications.

SI circuits have a gate-level degree of granularity as opposed to the module level of DI circuits. Only a few elements can be designed to be DI at the gate level. The module-level granularity is very coarse and usually has a negative impact on the performance of a circuit. Furthermore, SI circuits can be built using standard gate libraries, and, therefore, existing layout tools can be used for their low-level design.

It has also been shown that SI circuits are self-checking with respect to stuck-at-faults on gate outputs. That is, if a fault occurs, a circuit would stop rather than produce incorrect or unspecified outputs. Using this property, it is easy to design an autonomous fault-correction mechanism. Thus, if a circuit fails, the fault-correction mechanism will detect the fault place and correct the fault by, for example, replacing the faulty module with a reserved one.

The SI circuit design supports easy decomposition. A complex design can be decomposed into smaller subsystems with a well defined set of interface signals. In many cases, the environment model for a subsystem, which includes only the interface signals, is much smaller than the model for the rest of the system. Thus each subsystem can be designed separately using tools which cannot cope with the whole system. Furthermore, any subsystem can be later re-designed at will without the need for re-design of the rest of the system.

From the very beginning the design of SI circuits was associated with formal methods. Formal verification methods, unlike traditional simulation, can provably show that the circuit's behaviour is correct, or produce a sequence of events leading to the erroneous behaviour.

Behaviours described by PNs have a striking resemblance to asynchronous systems. The fundamental notions of the states and transitions between the states are inherent in PNs. This has prompted their application in SI circuit design. The graphical representation of the behaviour in the form of a PN (or STG) is easier to understand by circuit designers than an algebraic or trace model. The body of existing PN research is enormous; many results from PN theory have been applied to SI circuit design.

2.5 Conclusions

This chapter briefly outlined the main existing methodologies in the asynchronous circuit design. Examples of the reported designs include almost every conceivable combination of delay models, signalling protocols, formal models and their analysis methods. A number of works, e.g. [29, 41], provide a more extensive introduction and comparison of most common approaches. This chapter also described in more detail the pros and cons of SI circuit design which is the main subject of this work.

Chapter 3

Petri Nets and Related Formalisms

This chapter introduces Petri nets (PNs) and their related models, such as Labelled Petri nets (LPNs) and Signal Transition Graphs (STGs). Properties of the behaviour described by each formalism are defined and discussed. These properties are related to the properties of the correct behaviour of asynchronous circuits and systems. The existing methods for the behavioural analysis for each model are also outlined.

3.1 Petri Nets

This section defines a Petri net and introduces the notation used throughout the thesis. Interested readers may wish to refer to Peterson [65] and Reisig [67] for more extensive introductions to Petri net theory.

Definition 3.1.1 A *Petri net* (PN) is tuple $N = \langle P, T, F \rangle$ where

- P is a set of *places*, and
- T is a set of *transitions* such that $P \cap T = \emptyset$; and
- F is a flow relation between places and transitions, $F \subseteq P \times T \cup T \times P$.

Both P and T are assumed to be finite unless stated otherwise. □

Graphically, a PN is usually represented in the form of a graph with two types of vertices: circles, which correspond to places, and bars (or boxes), which correspond to transitions. The flow relation F is represented by directed edges (arcs) of the graph. A *bi-directional* arc is used sometimes as a shorthand for a pair of arcs going in the opposite directions between a particular pair of a place and a transition.

Each element $x \in P \cup T$ of a PN N has a set of input elements (which are connected with x by the arcs going to x) and a set of output elements (which are connected with x by the arcs originating from x). These sets of PN are called *pre-set* and *post-set* of x respectively and are defined as follows:

Definition 3.1.2 The sets $\bullet x$ and $x \bullet$ are called *pre-set and post-set* of $x \in P \cup T$ respectively iff:

- $\bullet x = \{y \in P \cup T | \ (y, x) \in F\}$

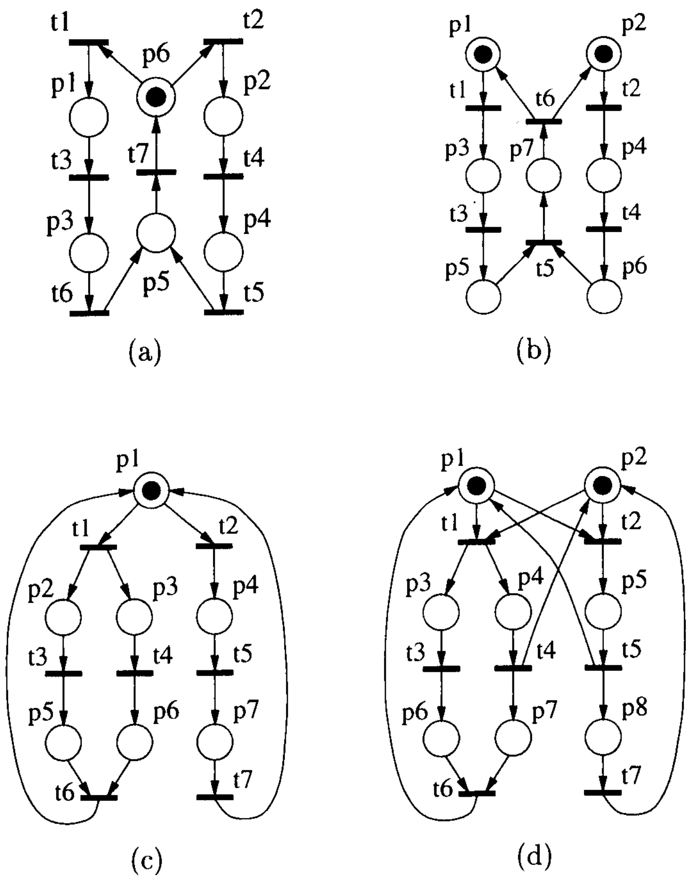


Figure 3.1: Examples of different classes of PNs: (a) SMPN, (b) MGPN, (c) FCPN and EFCPN (d).

– $x\bullet = \{y \in P \cup T |\ (x, y) \in F\}$

The notation $x_1 \bullet x_2$ means that $x_1 \bullet \cap \bullet x_2 \neq \emptyset$. □

In what follows, it is assumed that $\bullet t \neq \emptyset \neq t\bullet$, for every transition $t \in T$.

Structural properties of PNs define structural classes of PNs; these classes are identified below.

Definition 3.1.3 A *state machine* PN (SMPN) is a PN N such that $\forall t_i \in T : |\bullet t_i| = 1$ and $|t_i \bullet| = 1$. □

In other words, every transition in a SMPN has one input and one output place. An example of a SMPN is shown in Figure 3.1(a).

Definition 3.1.4 A *marked graph* PN (MGPN) is a PN N such that $\forall p_i \in P : |\bullet p_i| = 1$ and $|p_i \bullet| = 1$. □

Each place in a MGPN may have at most one input and one output transition. An example of a MGPN is shown in Figure 3.1(b).

A place p_i such that $|p_i \bullet| \geq 2$ is called a *conflict place* and the transitions that are in $p_i \bullet$ are said to be in *structural conflict*. This is defined below:

Definition 3.1.5 Two transitions $t_i \in T$ and $t_j \in T$ of a PN N are said to be in *structural conflict* iff $\bullet t_i \cap \bullet t_j \neq \emptyset$. □

The structural conflict between two different transitions t_i and t_j is denoted as $t_i \# t_j$. No two transitions of MGPN can be in structural conflict.

Definition 3.1.6 A *free choice* PN (FCPN) is a PN N such that for any $p_i \in P$ with $|p_i \bullet| \geq 2$ the following is true: $\forall t_i \in p_i \bullet : |\bullet t_i| = 1$. □

Definition 3.1.7 An *extended free choice* PN (EFCPN) is a PN N such that for any $p_i \in P$ the following is true: $\forall t_i, t_j \in p_i \bullet : \bullet t_i = \bullet t_j$. □

Any two conflicting transitions in FCPN have only one input place. EFCPNs are an extension of FCPNs allowing the conflicting transitions to have more than one input place but, at the same time, requiring that this set of input places is identical for these transitions. Examples of FCPN and EFCPN are shown in Figures 3.1(c) and 3.1(d) respectively.

In order to convey the dynamic properties of the system a notion of PN *marking* is used. A subset of places P may be *marked* which is denoted on the graph by placing *tokens* (thick black dots) into the places. Formally, a marking is defined below.

Definition 3.1.8 A *marking* of a PN N is a multiset M defined on P, i.e. it is a function $M : P \rightarrow \{0, 1, 2, \ldots\}$. □

Definition 3.1.9 A transition t_i is said to be *enabled* at a marking M iff $\bullet t_i \subseteq M$. □

A transition which does not have all of its input places marked at a marking M is said to be *disabled* at this marking. An enabled transition may **fire**, changing the current marking of the PN. The new marking is calculated as follows:

$$M' = M \setminus \bullet t_i + t_i \bullet.$$

This rule is called the PN **firing rule**. The firing of a transition t_i is denoted as:

$$M \xrightarrow{t_i} M'.$$

Thus a transition of a PN can be associated with some event and its input and output places with pre- and post-conditions. When all its pre-conditions are fulfilled, the event occurs (transition fires) changing the state of the system by setting its post-conditions to TRUE. A dynamic system is usually described by its structure and some initial state from which the system progresses. In terms of PNs this is defined as a marked PN.

Definition 3.1.10 A *marked* PN is a tuple $N = \langle P, T, F, M_0 \rangle$ where M_0 is an *initial marking* of the PN N. □

From now on any PN in this thesis will be treated as a marked PN unless stated otherwise.

Transitions of a PN start firing from its initial marking M_0 and their firing may continue while there exists at least one enabled transition. A sequence of transitions such that: $\sigma = M_1 \xrightarrow{t_1} M_2 \xrightarrow{t_2} M_3 \cdots$ is called *a firing sequence* from M_1. Obviously a transition t_i may

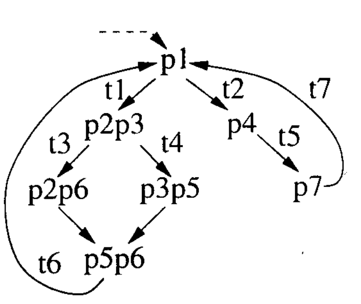


Figure 3.2: Example of an RG of a PN.

be included several times in one firing sequence. Each firing of this transition is called an *instance*[1]. Given a marking M_1 and a sequence σ it is easy to restore all visited markings by firing the transitions in the order of their instances in σ.

Definition 3.1.11 A marking M_m is said to be *reachable* in a PN N from M_1 iff there exists at least one firing sequence $\sigma = M_1 \xrightarrow{t_1} M_2 \xrightarrow{t_2} \cdots \xrightarrow{t_{m-1}} M_m$.

This is also denoted as: $M_1 \xrightarrow{\sigma} M_m$. □

The set of all makings which are reachable from M_0 is called the *reachability set* of a PN N. It is defined formally as follows:

Definition 3.1.12 The set $R = \{M_i \mid \exists\sigma : M_0 \xrightarrow{\sigma} M_i\}$ of markings of a PN N is called the *reachability set* of N. □

Often, the reachability set of a PN is represented as a directed graph where vertices are labelled with reachable markings and the edges are labelled with transitions which change one marking to another. This graph is often referred in the literature (e.g. [65]) as the *reachability graph* (RG). An example of the RG for the PN from Figure 3.1(c) is shown in Figure 3.2. In order to avoid cluttering in the figures from here on, obvious labels of arcs which represent the same transition from different markings will be omitted, e.g. arcs between (p_2,p_3) and (p_3,p_5) and between (p_2,p_6) and (p_5,p_6) correspond to the firing of transition t_4. These arcs are drawn parallel to each other. The initial marking is indicated by a broken arrow.

The dynamic behaviour of a PN allows some transitions **i)** to fire in parallel or **ii)** to prevent each other from firing. This is captured in the notions of **i)** concurrency and **ii)** (dynamic) conflict.

Definition 3.1.13 Two transitions t_i and t_j of a PN N are said to be *concurrent* iff there exists a reachable marking M at which both transitions are enabled and M contains the multiset $\bullet t_i$ and $\bullet t_j$; i.e. $\bullet t_i + \bullet t_j \subseteq M$. □

The fact that a particular marking M contains the sum of input places of both enabled transitions t_i and t_j means that transitions may fire *simultaneously* (as illustrated in Figure 3.3(a)) consuming tokens from their input places and producing tokens into their output places. Note that t_i and t_j may be in a structural conflict but they could still fire independently. If the number of tokens in the conflicting place(s) at M is sufficient for firing both

[1]This notion will also be used later to refer to transitions and places in an unfolding of PN

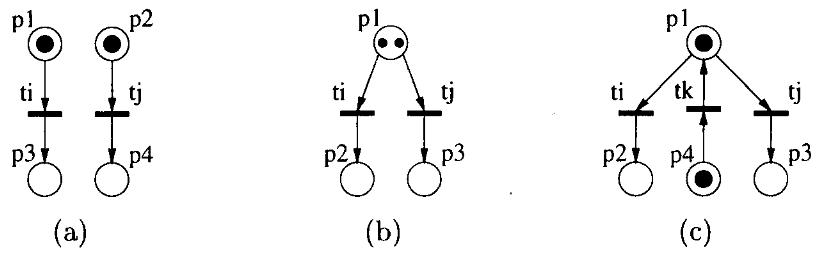


Figure 3.3: Illustration of relations between transitions: (a) concurrent, (b) structural conflict with concurrent transitions and (c) dynamic conflict.

t_i and t_j simultaneously, then these two transitions are concurrent (see Figure 3.3(b)). Two concurrent transitions are denoted as: $t_i || t_j$.

The notion of concurrency can be further extended to represent the relation between places and places and transitions. Two places p_1 and p_2 of a PN are said to be concurrent if there exists a reachable marking M such that $\{p_1, p_2\} \subseteq M$, i.e. p_1 and p_2 can be simultaneously marked at some reachable marking M. Lastly, a place p is said to be concurrent to a transition t if there exists a reachable marking M such that $\{p\} + \bullet t \subseteq M$, i.e. token in place p remains untouched while transition t fires at M.

A special case is when $\bullet t + \bullet t \subseteq M$; then transition t may fire simultaneously more than once. Such a transition is called **autoconcurrent**. For example, both t_i and t_j in Figure 3.3(b) are autoconcurrent.

Definition 3.1.14 A PN N is said to be *non-autoconcurrent* if no transition is autoconcurrent at any reachable marking of N. □

Non-autoconcurrency of events is one of the main requirements in asynchronous circuit specifications. Therefore, unless it is necessary to distinguish explicitly, all PNs in this work will be considered to be non-autoconcurrent.

Definition 3.1.15 Transition t_i of a PN N is said to be *in dynamic conflict* with another transition t_j at a marking M iff both transitions are enabled at M and the firing of t_i disables t_j. □

The notion of dynamic conflict is asymmetric, i.e. transition t_i may be disabling t_j whereas the firing of t_j does not disable t_i. A transition t_i in conflict with t_j is denoted as $t_i \vec{\#} t_j$. If the dynamic conflict is symmetric then, abusing the notation, it is denoted as $t_i \# t_j$. A symmetric conflict is illustrated in Figure 3.3(c), transitions t_i and t_j are in conflict at the marking (p_1, p_4). Furthermore, t_i and t_j may be in dynamic conflict at M but the firing of a third transition t_k may change the marking to M' at which t_i and t_j will be concurrent, e.g. transitions t_i and t_j at the marking (p_1, p_1) in Figure 3.3(c).

Recall the classification of PNs according to their structural properties. It follows from the definition of MGPN that no two transitions in a MGPN can be in dynamic or structural conflict. The notion of dynamic conflict is stronger than the notion of structural conflict. If two transitions are in dynamic conflict, then they are always in structural conflict. The notion of dynamic conflict is used to identify another class of PNs.

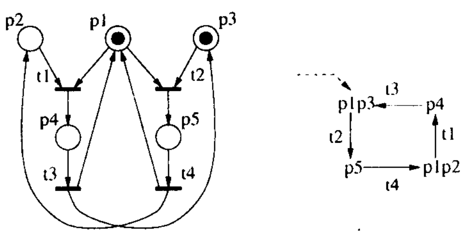


Figure 3.4: Illustration of UCPN.

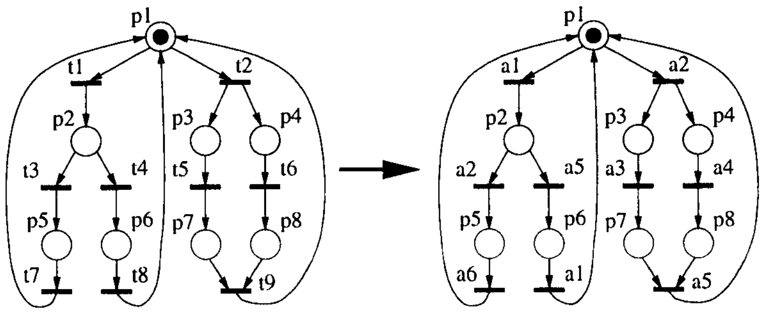


Figure 3.5: Example of an LPN.

Definition 3.1.16 A *unique choice* PN (UCPN) is a PN N such that for any two transitions t_i and t_j in dynamic conflict the following is true: $\bullet t_i = \bullet t_j$. □

An example of a UCPN is shown in Figure 3.4 with its reachability graph on the right. As it can be seen, transitions t_1 and t_2 are never enabled together although they are in structural conflict.

Each transition in a PN is unique. However, it is often impossible to describe a system with only one transition corresponding to each action of the system. Therefore, the notion of a *Labelled* PN is introduced below.

Definition 3.1.17 A *Labelled* PN (LPN) is a tuple $N^L = \langle N, A, L\rangle$ where

- N is a marked PN,
- A is a set of actions, and
- $L : T \to A$ is a *labelling function* which associates each transition of the PN N with some action from A.

□

Henceforth, N will be used instead of N^L to represent an LPN unless it causes confusion.

It is sometimes convenient to allow A to include a special action γ which is called *silent action* and does not cause any visible effect. An example of an LPN is given in Figure 3.5.

It is also possible to define the notions of concurrency and conflict between actions of an LPN.

Definition 3.1.18 Two actions a_i and a_j of an LPN N are said to be concurrent if there exists a reachable marking M at which two concurrent transitions $t_l : L(t_l) = a_i$ and $t_m : L(t_m) = a_j$ are enabled. □

Definition 3.1.19 An action a_i of an LPN N is said to be in dynamic conflict with another action a_j at a reachable marking M iff there exist two transitions $t_l : L(t_l) = a_i$ and $t_m : L(t_m) = a_j$ enabled at M and no transition $t_k : L(t_k) = a_j$ is enabled at the marking reached by firing t_l. □

The dynamic conflict between actions always requires two transitions to be in dynamic conflict. An example of a dynamic conflict between two actions is shown in Figure 3.5. Although transitions t_1 and t_2 are in dynamic conflict, the action a_1 is not disabling the action a_2. Such a behaviour is also sometimes described by the term *fake conflict.* However, the firing of a_2 leads to a marking (p_3, p_4) at which no transition labelled with a_1 is enabled.

Definition 3.1.20 A *deadlock* is a marking at which no transition is enabled. □

Obviously a deadlock represents a state of the system from which no further progress can be made. Presence of deadlocks is regarded as an error in a system which operates in cycles. A deadlock can be found while traversing the RG as a node with no outgoing arcs.

Another notion, closely related to the correct functioning of the system is boundedness.

Definition 3.1.21 A PN is said to be *k-bounded* iff the number of tokens in any place at any reachable marking does not exceed k. □

When modelling asynchronous systems with PNs, tokens often represent resources or signals stored in some part of the system. Obviously, the number of available resources or latches to store signals is bounded and, therefore, this property is a fundamental one. The RG of an unbounded PN is infinite[2]. Therefore, the boundedness of a PN is checked while the RG is constructed. If for a newly generated marking $M_j : M_i \xrightarrow{\sigma} M_j$ the following is true: $M_i \subset M_j$, then the sequence σ can be fired from the places marked at M_i repeatedly increasing the number of tokens in at least one place of M_j. Hence such a PN is unbounded.

A 1-bounded net is called **safe** net. The safeness of a PN can be checked during the construction of its RG by simply checking each newly generated marking.

Definition 3.1.22 A transition $t \in T$ of a PN N is called *live* at a marking M if there exists a firing sequence $\sigma : M \xrightarrow{\sigma} \cdots$ such that $t \in \sigma$. □

Definition 3.1.23 A PN N is called *strongly live* if every transition from T is live at every reachable marking.

A PN N is called *weakly live* if every transition from T is live at M_0. □

A transition which is not live usually indicates that some operation of the designed system can never be performed. A *live action* of an LPN is defined in a similar way.

Definition 3.1.24 An action $a_i \in A$ of an LPN N is called *live* at a marking M if there exists a firing sequence $\sigma : M \xrightarrow{\sigma} \cdots$ such that $\exists t \in \sigma : L(t) = a_i$. □

Another important notion is persistency, i.e. the ability of transitions and actions to stay enabled while other transitions are firing.

[2]In [65] a representation of infinite RGs is considered using ω-sequences. These are not considered in this work.

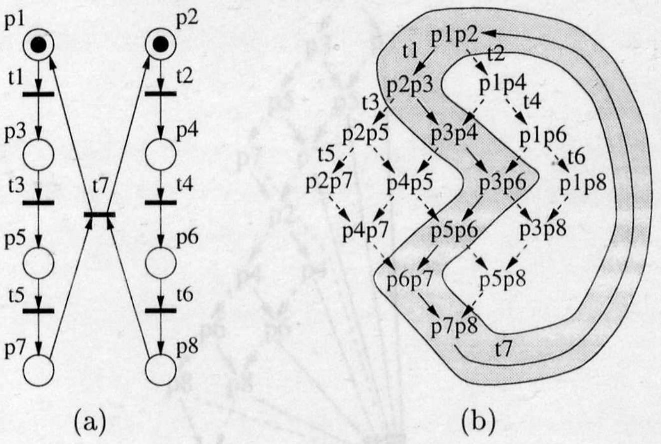


Figure 3.6: Example of a PN (a) and a reduced RG (b) built using stubborn sets method.

Definition 3.1.25 A transition t_i of a PN N is said to be *persistent* with respect to another transition t_j if t_j is not in dynamic conflict with t_i at any reachable marking M enabling both t_i and t_j. □

Definition 3.1.26 An action a_i of an LPN N is said to be *persistent* with respect to another action a_j if a_j is not in dynamic conflict with a_i at any reachable marking M which enables transitions labelled with both actions. □

For example, in Figure 3.5 action a_2 is persistent with respect to action a_1. As it will be shown later, the persistency of actions of an LPN is closely related to the correct functioning of an asynchronous circuit whose behaviour has been specified by the LPN.

3.2 Analysis of PN Behaviour by RG Methods

PN properties can be verified by traversing its RG. However, the size of the RG can be exponential in the number of transitions of the PN. It is often impossible to construct explicitly the whole RG even for a moderately sized PN. Several methods have been suggested for overcoming this problem. Amongst the most efficient are *stubborn set* methods [91] (a closely related method based on *persistent sets* was introduced in [28]) and PN *symbolic traversal* [61, 36]. Here these methods are only briefly outlined; an interested reader is referred to the above mentioned literature for more details.

Stubborn sets The objective of the stubborn set method [91] is to analyse a given PN for deadlock freedom. This method is based on the fact that different interleavings of concurrent transitions will lead to the same marking. Therefore, only one interleaving needs to be explored to find out if this marking is a deadlock marking. Thus only a subset of all reachable markings for a given PN is constructed. A stubborn set is a set of transitions whose ability to become enabled can not be affected by firing any transitions outside this set. The stubborn set is calculated using the structural properties of the PN graph. Therefore, the complexity of finding a stubborn set for a given marking depends on the size of the original PN. An example

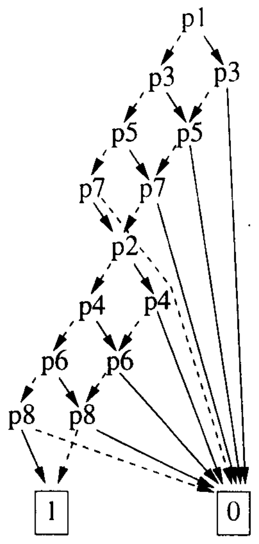


Figure 3.7: Example of a BDD representation of the RG in example from Figure 3.6(a).

of a PN and its reduced RG are shown in Figure 3.6. The shaded area shows one of the possible interleavings which may be explored by the algorithm.

PN symbolic traversal [61, 36] This method uses symbolic representation of the RG in the form of a characteristic boolean function[3]. This function is represented in a canonical graphical form called *Binary Decision Diagram* (BDD) [11]. Contrary to the stubborn set methods, it represents implicitly the whole RG. Each boolean variable is associated with a place of the analysed PN. If a PN is safe, each place can either be marked or unmarked, i.e. the corresponding variable is TRUE whenever a place is marked and FALSE otherwise. A marking is a boolean function which evaluates to TRUE when all places that belong to it are marked. The RG of a PN is represented as a BDD for a disjunction of boolean functions for all reachable markings. An example of a BDD representing the RG for the PN from Figure 3.6 is shown in Figure 3.7; the dashed vertices correspond to the FALSE value of each variable. This method has shown to be efficient for verification of state-based properties, i.e. the properties that can be determined from the RG vertices only.

Methods based on partial exploration of the RG can efficiently detect deadlocks in a PN. The verification of other properties by means of stubborn sets requires exploring additional states which reduces their efficiency. The PN symbolic traversal suffers from the *variable ordering* problem[4]. However, this method proved to be popular and is currently the subject of rigorous research.

3.3 Analysis of PN Behaviour by PN-unfolding

Another approach, based on partial orders, is to obtain an implicit representation of the RG for a PN by preserving the concurrency relation between instances of transitions (i.e. possible firings of transitions). Such a method, known as PN-unfolding, was introduced in [46, 47].

[3]This method is described in more detail later in Chapter 7.

[4]See also Chapter 7.

The formal definition of the PN-unfolding and other necessary notions are introduced below.

A PN-unfolding is an occurrence net [56] which is defined as a (P, T) labelled occurrence net in [47].

Definition 3.3.1 A (P, T) *labelled occurrence net* (possibly infinite) obtained from a PN N is a tuple $\langle N', L' \rangle$, where N' is a PN $N' = \langle P', T', F' \rangle$ and L' is a labelling function which maps P' and T' onto P and T respectively such that:

- F' is acyclic, i.e. the (irreflexive) transitive closure of F' (later denoted by F'^*) is a partial order;
- $\forall p' \in P', p' \in t'_1\bullet$ and $p' \in t'_2\bullet : t'_1 = t'_2$, i.e. there are no *backward conflicts*;
- $\nexists t' \in T' : t' \tilde{\#} t'$, i.e. no transition is in self-conflict, where $x' \tilde{\#} y'$ if there are t'_1, t'_2 such that $t'_1 \neq t'_2$ and $\bullet t'_1 \cap \bullet t'_2 \neq \emptyset$ and $(t'_1, x') \in F'^*$ and $(t'_2, y') \in F'^*$;
- $\forall t'_1, t'_2 \in T',\ L'(t'_1) = L'(t'_2),\ \bullet t'_1 = \bullet t'_2 :\ t'_1 = t'_2$;
- N' is finitely preceded, i.e. for every $x' \in P' \cup T'$, $\{y' | (y', x') \in F'^*\}$ is finite.

□

Note that the elements of P' and T' will be called *instances*; two elements satisfying $x' \tilde{\#} y'$ will be called to be in *conflict*.

A (P, T) labelled occurrence net is an acyclic PN which starts from a set of minimal places. Traversing such a PN visits every transition and place only once. Furthermore, if tokens are placed into the set of minimal places, then each transition can fire once and only once.

An unfolding construction algorithm is based on the temporal relations between instances of transitions and places. Similar to PNs, there are two types of temporal relations: conflict (already defined in Definition 3.3.1) and concurrency and, in addition, the precedence relation. The precedence and concurrency relations are defined below for instances of the occurrence net.

Definition 3.3.2 An element x'_1 of a (P, T) labelled occurrence net is said to be *preceding* another element x'_2 of this net iff there exists a set $Z \subseteq P' \cup T'$, $Z = \{z'_1, z'_2 \ldots z'_n\}$ such that $x'_1 \bullet z'_1 \bullet \ldots \bullet z'_n \bullet x'_2$.

Elements x'_1 and x'_2 are also said to be in the *sequential* relation; it is denoted as $x'_1 \prec x'_2$. The notation $x'_1 \preceq x'_2$ means that $x'_1 = x'_2$ or $x'_1 \prec x'_2$. □

Definition 3.3.3 Two instances x'_1 and x'_2 of the occurrence net N' are said to be *concurrent*, denoted as $x'_1 \| x'_2$, iff they are neither preceding each other nor in conflict. □

Note that the concurrency relation in the occurrence net is generalisable, i.e. given a set of elements $X' \subseteq P' \cup T'$ such that $\forall x'_1, x'_2 \in X', x'_1 \neq x'_2 : x'_1 \| x'_2$ all elements in X' can be marked and/or fired simultaneously if the execution is started from one token in each of the minimal places. If $X' \cap T' = \emptyset$, then X' is a set of instances of places which can *all* be marked at the same time. If $X' \cap P' = \emptyset$, then X' is a set of transition instances which can *all* fire simultaneously.

```
proc Build unfolding($N = \langle P, T, F, C, M_0 \rangle$)
    Initialise $N'$ with instances of places in $M_0$
    repeat
        for each $t$ in $T$ do
            Find unused set of mutually concurrent instances of places in $\bullet t$
            if such set exists then do
                Add instance of $t$ and $t\bullet$ to $N'$
            end do
        end do
    until no new instance can be added
    return $N'$
end proc
```

Figure 3.8: Algorithm for building PN-unfolding.

Definition 3.3.4 A PN-*unfolding* N' built from the PN N is the maximal (P, T) labelled occurrence net (up to isomorphism) satisfying the following:

- $\forall t' \in T'$: L' restricted to $\bullet t'$ and $\bullet L'(t')$ is a bijection, and L' restricted to $t'\bullet$ and $L'(t')\bullet$ is a bijection;
- L' restricted to the set P'_{min} of minimal places in P' is a bijection between P'_{min} and M.

□

For any element of a PN N the corresponding element of the PN-unfolding is referred to by adding an apostrophe to its name. The number of apostrophes denotes the occurrence number of this element. When the occurrence number is too large, it is denoted by a number itself.

The pseudo-code of the algorithm for building the PN-unfolding (similar to that of [47]) is given in Figure 3.8. The algorithm first initialises the unfolding by adding the instances of places which are marked at the initial marking M_0. It then checks each transition in order to find at least one transition whose places in $\bullet t$ have mutually concurrent instances in the unfolding. If such a set is found and this transition has not yet been instantiated with this set of inputs, then instances of t and places in $t\bullet$ are added to the unfolding. If no new instance can be added to the unfolding, the algorithm terminates. An illustration of the output of the PN-unfolding construction algorithm for the PN from Figure 3.6 is given in Figure 3.9. It illustrates three steps of the algorithm's work: the PN-unfolding initialised with instances of places marked at the initial marking (Figure 3.9(a)), the PN-unfolding after adding instances of t_1 and t_2 (Figure 3.9(b)) and after adding one instance of each transition of the original PN (Figure 3.9(c)).

Definition 3.3.5 The *initial transition* of a PN-unfolding is denoted as $\perp$ and is defined so that $\bullet \perp = \emptyset$ and $\perp \bullet = P'_{min}$. □

Although $\perp$ does not have pre-set places, this is harmless. It can be viewed as a "phantom" transition which initialises the PN by putting tokens into the places of initial marking, and

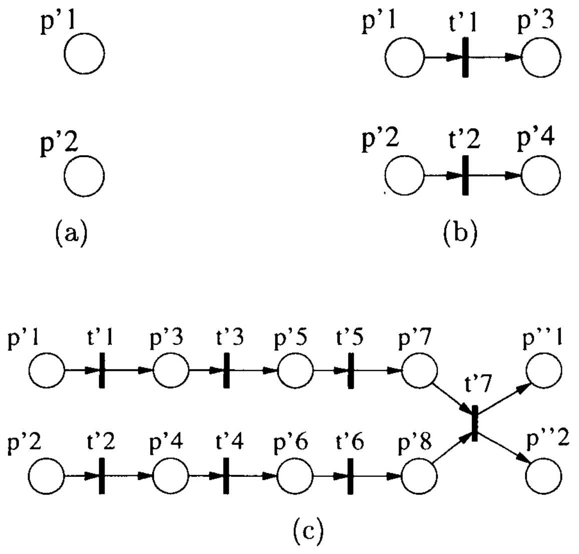


Figure 3.9: Steps of the PN-unfolding algorithm (a) initialised N', (b) after adding two instances, (c) after adding one instance of each transition.

is needed only to make some notation simpler. This transition is never fired more than once. In discussions about reachability it should be borne in mind that the initial marking is $\perp \bullet$.

Definition 3.3.6 A *configuration* C of an unfolding N' is a non-empty subset of T' such that:

- $\forall t'_1, t'_2, t'_1 \bullet t'_2, t'_2 \in C : t'_1 \in C$;
- $\forall t'_1, t'_2 \in C, t'_1 \neq t'_2 : \bullet t'_1 \cap \bullet t'_2 = \emptyset$.

$\square$

Property 3.3.1 No two instances which belong to a configuration C are in conflict. $\square$

A configuration is a set of transition instances which is conflict-free and backwards closed with respect to the precedence relation.

Definition 3.3.7 The *local configuration* of a transition instance t', denoted as $\lceil t' \rceil$, is the least backwards closed subset of T' with respect to F' containing t'. $\square$

Clearly a local configuration is a configuration. From definitions also follows that $t'_1 \prec t'_2$ iff $t'_1 \in \lceil t'_2 \rceil$.

Proposition 3.3.1 For any two transitions $t'_1, t'_2 \in T'$ the following is true: $t'_1 \preceq t'_2$ iff $\lceil t'_1 \rceil \subseteq \lceil t'_2 \rceil$. $\square$

For any transition t' in the unfolding the following is true: $\lceil \perp \rceil \subseteq \lceil t' \rceil$. The local configuration of $\perp$ is the initial transition itself.

Definition 3.3.8 The set $C\bullet \subseteq P'$ is called the *post-set* of a configuration C and is calculated as: $C\bullet = \{t'\bullet \ : t' \in C\} \setminus \{\bullet t' : t' \in C\}$. $\square$

Definition 3.3.9 The multiset of places $F_s(C) = L'(C\bullet)$ (i.e. if $C\bullet = \{p'_1, p'_2, \ldots\}$, then $F_s(C) = L'(p'_1) + L'(p'_2) + \cdots$) of the PN N is called the *final state* of configuration C in the PN-unfolding built from N. □

The set of maximal elements of a configuration, called the *max-set of configuration*, is defined as $Max(C) = \{t' \in C : \nexists t'' \in C, t' \prec t''\}$. The max-set is unique for each configuration. For any instance in the max-set of configuration the following is true: $t'\bullet \subseteq C\bullet$.

Property 3.3.2 For any finite configuration C the following is true:

$$C = \bigcup_i \lceil t'_i \rceil \, : \, t'_i \in Max(C)$$

□

That is, any finite configuration can be found from the local configurations of the instances in its max-set.

The next property follows from the definitions of conflict relation and configuration.

Property 3.3.3 For any conflict free set of instances $T' = \{t'_1 \ldots t'_n\}$ the following is true: $\lceil t'_1 \rceil \cup \ldots \cup \lceil t'_n \rceil$ is a configuration. □

Lemma 3.3.1 Let N' be the PN-unfolding of a PN N. If for two instances $t'_1, t'_2 \in T'$ the following is true: $t'_1 \| t'_2$, then two transitions $t_1 = L'(t'_1)$ and $t_2 = L'(t'_2)$ are concurrent.

Proof: Since t'_1 and t'_2 are concurrent, then no pair of instances in their local configurations may be in conflict. From Definition 3.3.6 it follows that there exist a configuration $C = \lceil t'_1 \rceil \cup \lceil t'_2 \rceil$. Furthermore, there exists a configuration $C' = C \setminus \{t'_1, t'_2\}$ for which the following is true: $\bullet t'_1 \subseteq C'\bullet$ and $\bullet t'_2 \subseteq C'\bullet$ and $\bullet t'_1 \cap \bullet t'_2 = \emptyset$. Hence there exists a reachable marking $M = F_s(C')$ such that $(L'(\bullet t'_1) + L'(\bullet t'_2)) \subseteq M$, i.e. t_1 and t_2 are concurrent.

□

It was proved in [47] that for any reachable marking M there exists a finite configuration C such that $M = F_s(C)$ and vice versa.

Lemma 3.3.2 For a transition $t \in T$ of a PN N which is live at M_0 there exists a corresponding instance $t' \in T'$ of N' such that: $L'(t') = t$.

Proof: Since every reachable marking is represented as a post-set of some finite configuration, then the marking M enabling t will have a corresponding configuration $C : F_s(C) = M$. Thus a set of mutually concurrent instances of $\bullet t$ exists in the unfolding and, hence, t' will be instantiated.

□

Theorem 3.3.1 Two transitions $t_1, t_2 \in T$ are concurrent in PN N iff there exists a pair of their instances $t'_1, t'_2 \in T'$ in the PN-unfolding N' such that $t'_1 \| t'_2$.

```
proc Build truncated unfolding($N = \langle P, T, F, M_0 \rangle$)
    Initialise $N'$ with instances of places in $M_0$
    Initialise QUEUE with t enabled at $M_0$
    while QUEUE not empty do
        Pull $t$ from QUEUE
        if $t'$ is a not cutoff then do
            Add $t'$ and $t'\bullet$ to $N'$
        end do
        for each $t$ in $T$ do
            Find unused set of mutually concurrent instances of places in $\bullet t$
            if such set exists then do
                Add $t$ to QUEUE in order if its $|\lceil t' \rceil|$
            end do
        end do
    end do
    return $N'$
end proc
```

Figure 3.10: Algorithm for truncated PN-unfolding.

Proof: [*if*] Follows from Lemma 3.3.1.

[*only if*] Any reachable marking M such that $(\bullet t_1 + \bullet t_2) \subseteq M$ is represented as a final state of some configuration C [47]. This configuration has the post-set $C\bullet$ which includes disjoint instances of $\bullet t_1$ and $\bullet t_2$. Therefore, t'_1 and t'_2 will be instantiated in N' with these instances of $\bullet t'_1$ and $\bullet t'_2$. The two instances t'_1 and t'_2 cannot be in conflict with any instances in C, nor can they be preceding each other. Hence, $t'_1 \| t'_2$.

□

The *size of configuration* C is defined as the number of instances in C and is denoted as $|C|$.

The complete unfolding of a cyclic PN can be infinite. Thus a characteristic fragment of the unfolding is required which will be finite but as informative as the full unfolding.

A method of truncating the PN-unfolding was introduced in [47]. This method is based on the cutoff condition as follows:

Definition 3.3.10 (Cutoff condition [47]) A newly built instance t'_c of the PN- unfolding is a cutoff point if there exists another instance t' such that:

$$F_s(\lceil t' \rceil) = F_s(\lceil t'_c \rceil) \text{ and } |\lceil t' \rceil| < |\lceil t'_c \rceil|.$$

□

In other words, an instance t'_c is a cutoff point if there exists another instance t' in the already built portion of the PN-unfolding such that the firing of the transitions whose instances are $\lceil t' \rceil$ and $\lceil t'_c \rceil$ leads to the same PN marking and the local configuration of the existing transition is smaller.

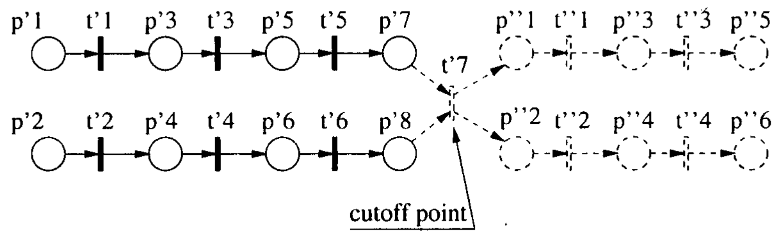


Figure 3.11: The truncated unfolding for PN from Figure 3.6(a).

A somewhat simplified version of the McMillan's algorithm is shown in Figure 3.10. The construction of a truncated unfolding starts from a set of places which are labelled as occurrences of places in the initial marking of a PN. Contrary to the algorithm in Figure 3.8, it checks each added transition for a cutoff condition. Furthermore, the truncated PN-unfolding algorithm uses QUEUE to sort the newly generated instances so that an instance is added only after all instances with smaller local configurations are considered. The truncated PN-unfolding for the PN in Figure 3.6 is shown in Figure 3.11. Instance t'_7 is a cutoff point and, therefore, no instance depicted with a dashed line will be added to the truncated PN-unfolding. The construction stops for a bounded PN [47] when there are no more transitions that could be added into the PN-unfolding.

Theorem 3.3.2 [47] A marking M of a bounded PN N is reachable iff there exists a configuration C in the truncated unfolding N' built from N such that $F_s(C) = M$. □

From this theorem it follows that instead of constructing the reachability graph, a truncated PN-unfolding can be constructed to represent all reachable markings of a bounded PN.

3.4 Signal Transition Graphs

Signal Transition Graphs (STGs) have been introduced independently by [73] and [13] for the low level modelling of asynchronous circuits. They became popular because of their close relationship to PNs, which provides a powerful theoretical background for the specification and verification of asynchronous circuits. In addition, an asynchronous circuit can be synthesised from its STG specification if it satisfies certain criteria of implementability. In this section STGs are formally defined; the correctness of a behaviour described by an STG is defined along with the corresponding properties.

Definition 3.4.1 A *Signal Transition Graph* (STG) is a tuple $G = \langle N, A, v_0, \Lambda \rangle$ where

- N is a marked PN,
- A is a set of signals,
- v_0 is an initial state of the STG, which is a binary vector of dimension $|A| : v_0 \in \{0,1\}^{|A|}$,
- Λ is a labelling function which labels every transition of N with a signal transition $+a$ or $-a$ where $a \in A$.

□

It can be observed from the definition that STGs are a particular case of LPNs where the set of actions is restricted to signal transitions, i.e. $+a(-a)$ represents the change of the value of signal a from logical 0 to logical 1 (from 1 to 0). The set of signal transitions on A is defined as $*A = \{+,-\} \times A$ so that $*a \in \{+a,-a\}$ and $|*a|$ denotes the signal itself, i.e. $|*a| = a$. There also exists a less strict definition of the STG which implies that some of the transitions of the STG can be *dummy* transitions, i.e. they do not change values of any signal in the STG.

STGs were introduced as a formal model for the specification of asynchronous circuits. Each transition is associated with some signal transition of the circuit or its environment. Therefore, the set of signals of an STG is usually divided into two subsets: a *set of input signals* and a *set of output signals.* Obviously, not every behaviour can be regarded as a correct one for circuit implementation. The notion of correctness of an STG is defined on the firing sequences that it can generate. First, a valid firing sequence is defined.

Definition 3.4.2 A firing sequence $\sigma : M_0 \xrightarrow{\sigma} M$ is *valid* iff for every signal $a : \exists t \in \sigma : \Lambda(t) = *a$ the following is true:

- the next possible change of signal a after $+a(-a)$ can only be $-a(+a)$,
- the first change of signal a is consistent with the initial state of the STG, i.e. if the value of a is $0(1)$ in the initial state, then $+a(-a)$ is the first change of a in any firing sequence.

□

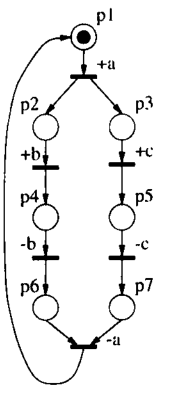
The first condition is known as *switchover correctness* [35]. The second condition is known as *stability of the initial marking*, also due to [35].

Definition 3.4.3 An STG is called *valid (correct)* iff the underlying PN is finite, bounded and every feasible sequence in it is valid. □

Property 3.4.1 An STG is invalid if there exist two concurrent transitions labelled with signal transitions of one signal a. □

The above property proves to be very important later in the analysis of STGs using the PN-unfolding method. Its correctness follows from the fact that concurrent transitions are enabled together at at least one reachable marking. Hence, further advancement of the system from this marking will violate at least one of the conditions of a valid sequence.

Example. An STG shown in Figure 3.12(a) is valid if the initial state is $\{000\}$ where the order of signals in the state vector is abc. An example of an invalid STG is shown in Figure 3.12(b). If the initial state is $\{10\}$ or $\{11\}$ (the order of signals is ab) then the firing of $+a$ will be inconsistent with the initial state. If the initial state is $\{01\}$, then for signal b, a change $-b$ can occur after $-b$ without an intermediate $+b$ in the sequence $p_1 \xrightarrow{+a} p_3 \xrightarrow{-a} p_5 \xrightarrow{-b} p_1 \xrightarrow{+a} p_3 \xrightarrow{-a} p_5 \xrightarrow{-b}$ (a shorter sequence $p_1 \xrightarrow{+a} p_3 \xrightarrow{-a} p_4 \xrightarrow{+a} p_2 \xrightarrow{+b} p_3$ is also inconsistent with the initial state). ■



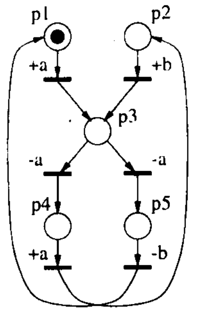


Figure 3.12: Examples of a valid (a) and invalid (b) STGs.

A *single-run* STG is an STG whose RG contains no cycles. Also, a *cyclic* STG is an STG whose RG is strongly connected. Often practically useful STGs consist of a combination of single-run and cyclic segments, e.g. an STG with a segment which is executed once before it enters its cyclic segment. Such STGs model the behaviour of circuits or signalling protocols with initialisation.

STGs are a particular subclass of LPNs. Thus they have the same properties as LPNs defined in the previous section. An additional important property of STGs, related to the correctness of the circuit functioning is *output signal persistency* which is defined below.

Definition 3.4.4 A signal a_i of an STG G is said to be *persistent* with respect to another signal a_j if a_j is not in dynamic conflict with a_i at any reachable marking M which enables transitions labelled with both actions. □

Since the set of signals is divided into sets of input and output signals, the signal persistency can be defined with respect to a set of signals.

Definition 3.4.5 An STG G is called persistent with respect to a set of signals $A' \subseteq A$ if every signal $a_i \in A'$ is persistent with respect to any other signal $a_j \in A$ at any reachable marking M which enables transition labelled with a_i. □

An STG where $A' = A$ is simply called *persistent* STG. The output signal persistency has an important practical meaning.

Definition 3.4.6 An STG is called *output signal persistent* if it is persistent with respect to its output signals. □

Output signal persistency is closely related to the correct operation of the circuit. It guarantees that the outputs of the circuit cannot change non-deterministically. Thus, for the observer in the environment, the circuit always reacts deterministically to any input stimuli.

3.5 Analysis of STG Behaviour

This section studies the conventional method of STG analysis. This method is based on the reachability analysis of the underlying PN.

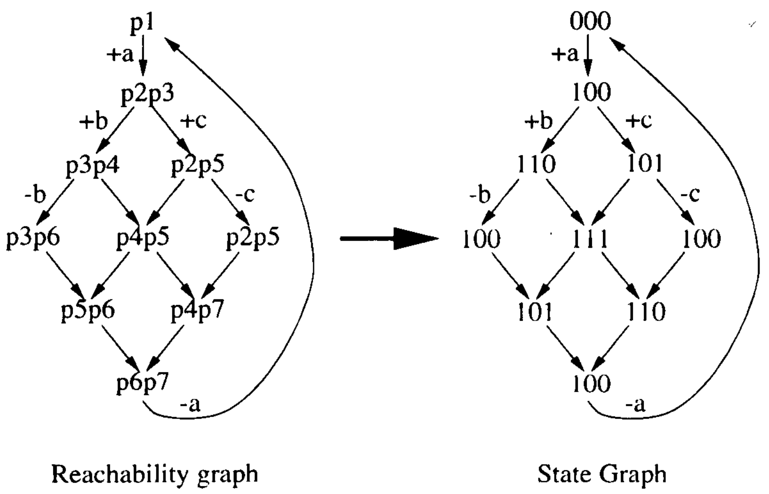


Figure 3.13: Illustration of the RG and SG built for the STG example from Figure 3.12(a).

The current state of an STG is captured using the notion of the *state vector*. A state vector v of an STG G is a binary vector of size $|A|$ such that each of its elements correspond to one and only one signal. The element $v[i]$, corresponding to a signal a_i, represents the state ("high" or "low") of this signal. It is assumed that 1 corresponds to the "high" value of the signal and 0 corresponds to the "low" value of the signal. Whenever an STG changes its marking by a signal transition of a_i the state vector changes accordingly.

The behaviour described by an STG is analysed by means of constructing its State Graph.

Definition 3.5.1 The *State Graph* (SG) constructed from an STG $G = \langle N, A, v_0, \Lambda\rangle$ is a tuple $S = \langle V, E\rangle$ where for each reachable marking M_i of N there exists a vertex $s_i = (M_i, v_i) \in V$ assigned with a state vector v_i. For each transition $t \in T : M_i \xrightarrow{t} M_j$ there exists an edge $e \in E$ connecting two corresponding vertices and labelled with $\Lambda(t)$.

Each vertex of the SG is called a *state*. □

Figure 3.13 shows an example of the RG and the SG obtained from the STG in Figure 3.12(a).

The validity of an STG is checked on the SG level through the notion of *consistent binary vector assignment.*

Definition 3.5.2 An SG labelling is called *consistent* if for all edges e the following is true:

- if $M_i \xrightarrow{+a_j} M_k$ then $v_i[j] = 0$ and $v_k[j] = 1$;
- if $M_i \xrightarrow{-a_j} M_k$ then $v_i[j] = 1$ and $v_k[j] = 0$;
- $v_i[j] = v_k[j]$ otherwise

□

It has been shown in [41] that an STG is valid if its SG is finite and its binary vector assignment is consistent. The SG of an STG is finite if the underlying PN is bounded. Thus, the validity of an STG can be verified through the construction of its SG instead of examining all its feasible sequences.

If an STG specifies a desired asynchronous circuit, then its validity only indicates that this STG *may be* implemented as an asynchronous circuit. If the obtained SG does not have a consistent state vector assignment, then the STG is invalid and should be redesigned. However, the consistent state vector assignment does not immediately guarantee its implementability. The behaviour must satisfy other implementability properties such as *output signal persistency* and *Complete State Coding.*

The circuit implementation should be free from hazards – unspecified changes of signals. Several works, e.g. [41, 35], define *semi-modularity* of SI circuits as the criterion of its hazard freedom.

Suppose that a circuit is given in the form of a gate net-list together with the model of the environment in which this circuit operates. A circuit is said to be *semi-modular* if every its signal becomes stable only through changing its value. Informally, a semi-modular circuit is hazard-free since no signal changes are allowed which may cause a glitch on the output of any gate.

The *State Transition Diagram* (STD) is constructed for a circuit which represents all states of the circuit as a graph with vertices corresponding to the states of the circuit and edges representing transitions between the states. Every vertex is associated with a binary vector whose elements represent the states of signals. A signal is said to be stable at some state if the value of the corresponding element of the binary vector is equal to the value calculated by the corresponding logic function under the input values given by the vector; and it is called excited otherwise. An STD is called semi-modular if there are no edges connecting two states so that the corresponding transition removes excitation from some signal other than the one changing in the transition.

The notions of STD and SG are very close. It is possible to define an isomorphism between an STD and an SG obtained for some STG. In this case the STG is said to be an event-based representation of the circuit. It was shown in [41] that a circuit is semi-modular (hazard free) if the STG is valid, output signal persistent and it has an SG isomorphic to the STD of the circuit. Interested reader is referred to [99, 41, 35] for a detailed explanation of the relationship between circuit semi-modularity and its hazard freedom.

An important observation from the above discussion for this work is that STG validity and output signal persistency are necessary conditions for an SI circuit to be hazard free. This means that if for a given STG it is hypothesised that there exists a hazard free SI circuit implementing this behaviour, then the following properties of this STG must be checked in order to guarantee its hazard free implementation as an SI circuit:

- STG boundedness;
- STG validity;
- signal output persistency.

The process of obtaining an implementation from an STG specification is called *synthesis.* During the synthesis process, truth tables for future logic gates are derived from the binary vectors assigned to states of the built SG. Each vector is considered as a point in the domain of a boolean function whose value is the implied value of some output signal. However, if two different states of the SG imply the same set of values for the output signals, then these two states cannot be distinguished using only these output signals. Therefore, another necessary condition, the Complete State Coding (CSC) [13] requirement, must be satisfied if the circuit is to be synthesised from an STG by constructing the corresponding SG.

Definition 3.5.3 An STG is said to have *Complete State Coding* (CSC) if for any two states s_1 and s_2 in its SG such that $v_1 = v_2$ the set of output signals excited (defined in Section 2.2.3) in both states is equal. □

At the circuit level all information about the state of the system is kept in the values of the signals, i.e. binary codes. Satisfying the CSC ensures that each state of the synthesised circuit is unique at the circuit level.

The CSC condition distinguishes between input and output signals. It does, however, allow an SG to have two states with different marking components but equal binary vectors assigned to them. A stricter condition was also introduced in [13] which requires every state of the SG to have a unique binary vector assigned to it. This property is called *Unique State Coding* and is defined below.

Definition 3.5.4 An STG is said to have *Unique State Coding* (USC) property iff for any two states s_1 and s_2 in its SG such that $M_1 = M_2$ the following is true: $v_1 = v_2$. □

Further discussion about these properties in relation to the synthesis of SI circuits can be found in Chapter 6.

3.6 Conclusions

In this chapter PNs and LPNs were introduced. Structural and behavioural properties of each formalism were defined. This chapter also reviewed the main two methods of the PN behavioural analysis based on constructing a reduced reachability graph and symbolic representation of the reachability graph in the form of BDD.

Section 3.3 introduced PN-unfolding as the means for analysis of PN behaviour. This method is based on the partial order approach which constructs a labelled occurrence net. The algorithms for building the unfolding and its truncated fragment are reproduced. The truncated PN-unfolding represents all reachable states of the original bounded and finite PN in a finite fragment of the PN-unfolding. The truncation uses the cutoff condition suggested in [47]. The following chapter demonstrates the potential inefficiency of this cutoff condition and introduces a PN-unfolding segment and a new cutoff condition which avoids this drawback.

The notion of STGs as a particular case of LPNs was introduced. It allows specification of asynchronous circuits at the signal level. The STG analysis method based on the construction of its SG was presented. The properties of STGs which are crucial to the correct implementation of an STG specification as an SI circuit are defined. It will be shown later that these properties can be efficiently verified using the STG-unfolding segment, a new concept introduced in Chapter 5.

Chapter 4

Analysis of PN Models

This chapter describes a new concept of the PN-unfolding segment which is based on the truncated PN-unfolding. First, the existing truncated PN-unfolding is adapted to enable the analysis of the temporal relations between transitions of the original PN. Then the redundancy of the modified truncated PN-unfolding is explained. This chapter then presents a cutoff condition for safe PNs due to Esparza *et. al.* [22], which attempts to avoid this redundancy. In addition, an experimental technique of truncation is presented which is applied to truncated unfoldings of unsafe PNs and demonstrates great potential. The fragment of the unfolding constructed with the new cutoff condition is called the PN-*unfolding segment.* Experimental results illustrate the reduction in size of the fragment which is required to represent the reachable state space for two realistic PN models. The PN-unfolding segment is applied to the verification of asynchronous circuits.

4.1 Adapting Truncated PN-unfolding

The algorithm for obtaining a truncated PN-unfolding [47] constructs a representation of all reachable markings. However, this is not enough if the relations between transitions need to be analysed. According to the original McMillan's algorithm, cutoff point transitions are not included in the truncated unfolding. Thus some of the live transitions t of the original PN may have no corresponding instances t' in the truncated PN-unfolding.

Consider the truncated PN-unfolding in Figure 4.1(a) reproduced from Figure 3.9. Since instance t'_7 is a cutoff point, this instance will not be added to the truncated PN-unfolding.

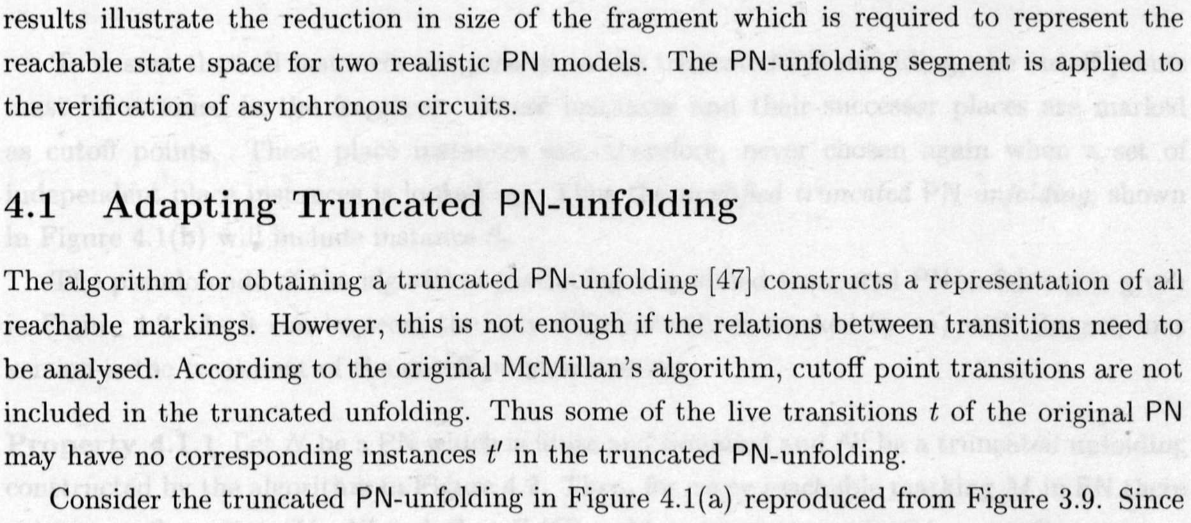


Figure 4.1: Example of truncated (a) and modified truncated (b) PN-unfolding.

```
proc Build modified truncated unfolding($N = \langle P, T, F, C, M_0 \rangle$)
   Initialise $N'$ with instances of places in $M_0$
   Initialise QUEUE with $t$ enabled at $M_0$
   while QUEUE not empty do
      Pull $t$ from QUEUE
**       Add $t'$ and $t'\bullet$ to $N'$
**       if $t'$ is a cutoff then do
**          Mark $t'$ and $t'\bullet$ as cutoff points
**       end do
      for each $t$ in $T$ do
**       Find unused set of concurrent instances of places in $\bullet t$
**          which are not successors of a cut-off transition
         if such set exists then do
            Add $t$ to QUEUE in order if its $|\lceil t' \rceil|$
         end do
      end do
   end do
   return $N'$
end proc
```

Figure 4.2: Algorithm for modified truncated PN-unfolding.

To ensure that all instances are present in the truncated PN-unfolding, the cutoff points must be retained in the fragment. These instances and their successor places are marked as cutoff points. These place instances are, therefore, never chosen again when a set of independent place instances is looked up. Thus the *modified truncated* PN-*unfolding*, shown in Figure 4.1(b) will include instance t'_7.

The pseudo-code of the algorithm producing a modified truncated PN-unfolding is given in Figure 4.2. As it can be seen, the only difference (lines marked by $**$) with the previous version is the treatment of the cutoff point transitions.

Property 4.1.1 Let N be a PN which is finite and bounded and N' be a truncated unfolding constructed by the algorithm in Figure 4.2. Then, for every reachable marking M in PN there exists a configuration C in N' such that $F_s(C) = M$ and no instance in C is a cutoff transition.

□

Proposition 4.1.1 For any transition t which is live from M_0 in a finite and bounded PN N there exists an instance t' in the modified truncated PN-unfolding built for N.

Proof: By Property 4.1.1, for every reachable marking M there exists a configuration C in the truncated PN-unfolding N' such that $F_s(C) = M$ and no instance in C is a cutoff transition. Hence, no instance in $C\bullet$ is a successor of a cutoff transition; and, therefore, if a transition is enabled at M, then its instance t' will be instantiated in N' unless t' is a cutoff transition.

□

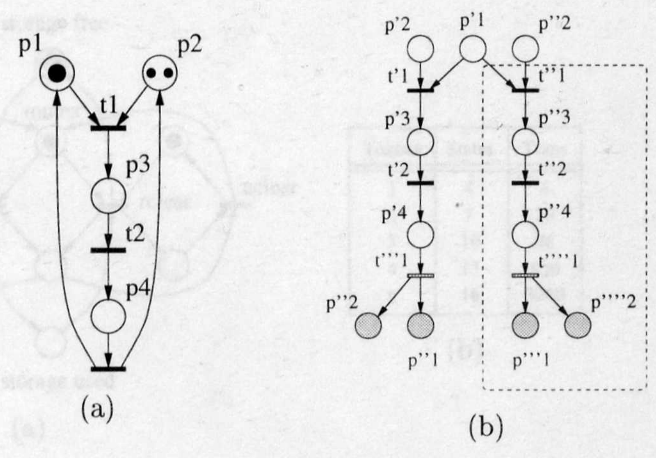


Figure 4.3: Example of truncated PN-unfolding redundancy

Proposition 4.1.2 For all transitions $t_1 \dots t_n$ enabled at a reachable marking M of a finite and bounded PN N there exist a configuration C and instances $t'_1 \dots t'_n$ in the modified truncated PN-unfolding such that $\forall t'_i : \bullet t'_i \subseteq C\bullet$ and $F_s(C) = M$ and no place in $C\bullet$ is a successor of a cutoff transition. □

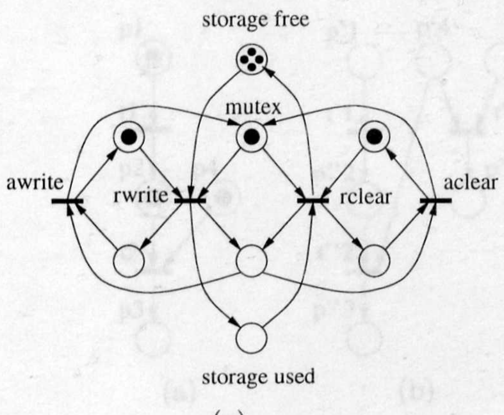
The result above is very important as it shows that the relations between transitions can be analysed in the modified truncated PN-unfolding. The next corollary proves to be useful for establishing relations between elements of the original PN from its modified truncated PN-unfolding.

Corollary 4.1.1 Two elements $x_1, x_2 \in P \cup T$ of a PN N are concurrent iff there exist two instances x'_1 and x'_2 in the modified truncated unfolding segment such that $x'_1 \| x'_2$. □

Since the addition of cutoff points cannot lead to the instantiation of new instances, all statements proved in the previous chapter and [47] hold for the modified truncated PN-unfolding. The size of the unfolding is increased only by the number of cutoff transitions and their immediate successors. Henceforth, the modified truncated PN-unfolding will be referred to as a truncated PN-unfolding.

4.2 Avoiding Redundancy in PN-unfolding

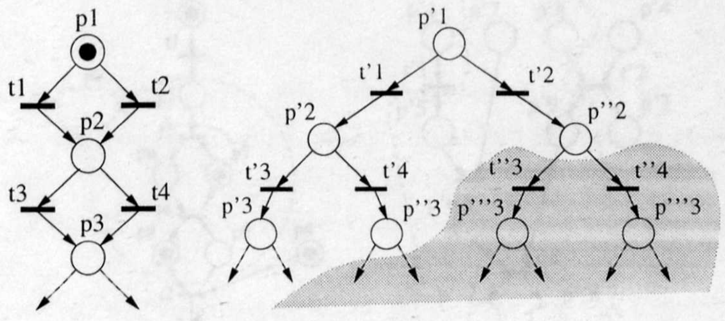
The original McMillan's algorithm may produce a truncated PN-unfolding which includes redundant instances of transitions. A redundant instance t' is a non-cutoff transition of the constructed truncated PN-unfolding which does not add new information about the PN behaviour, i.e. the final state of any configuration C which includes t' is equal to the final state of some other configuration C' of the truncated PN-unfolding. Note that the cutoff transitions are not considered to be redundant as their presence ensures the correct representation of events in the unfolding. This section examines the redundancy of the truncated PN-unfolding, explains an existing method of battling it for safe PNs and suggests a truncation technique for bounded PNs.



(a)

Tokens	States	Trans
1	4	4
2	7	20
3	10	96
4	13	520
5	16	3260

(b)



(c) (d)

Figure 4.4: Examples of truncated PN-unfolding growth

4.2.1 Redundancy of Truncated PN-unfolding

The truncated PN-unfolding has the power to distinguish tokens in a place. If a place p becomes unsafe, then the number of instances of this place will correspond to the number of tokens in it. The PN firing rule does not distinguish which of the tokens should be removed by the firing of a transition t which has p in $\bullet t$. The marking reached after the firing of t is the same whichever token is used. However, in the unfolding, the structural distinction between place instances may cause the construction of several instances of t and their successors, increasing the size of the unfolding.

Example. Consider a PN and its truncated unfolding shown in Figure 4.3. Since place p_2 is unsafe, there will be two instances of p_2 in the PN-unfolding. Thus a single token in p_1 will be "paired" with each of the tokens in p_2 and two instances of transition t_1 will be constructed. If the order of the tokens in the unsafe place does not matter, as it is true for most systems, then the second instance of t_1 and all its successors are redundant.

Another example, shown in Figure 4.4(a) is taken from Nowick and Dill's work [58] and represents a storage unit. The number of tokens in an unsafe place represents the capacity of the storage unit. The truncated PN-unfolding grows exponentially in the number of tokens in this example as illustrated in the Table in Figure 4.4(b). ■

An overly strong cutoff condition is another cause of redundancy; it may lead to the creation of redundant transitions even in the case of a safe PN. It is easy to come up with

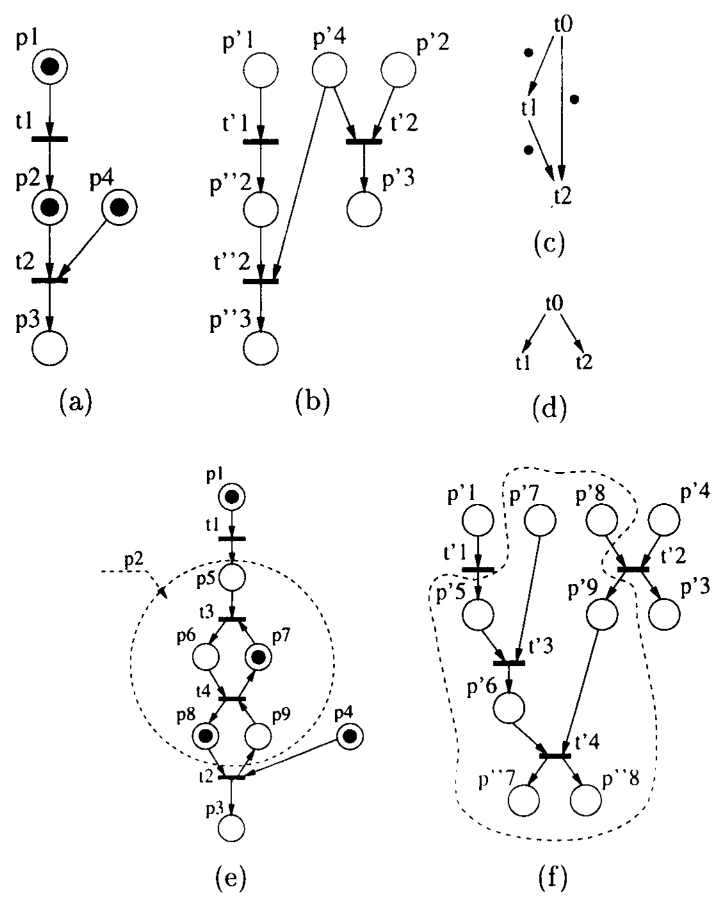


Figure 4.5: Illustration of FIFO transformation of an unsafe PN.

an extreme example (e.g. example in Figure 4.4(c) taken from [39]) which is a safe SMPN. Because of the strict condition on the size of the local configuration for the cutoff transition, the size of the truncated PN-unfolding of this PN, shown in Figure 4.4(d), will be exponential. However, only a fraction of the instances, linear in the number of transitions in the original PN, will be sufficient to represent all reachable markings. The redundant copies of instances are shaded out in Figure 4.4(d). In this case the truncated unfolding distinguishes tokens arriving in a place through different paths.

An interesting approach was described in [35]. The CD-unfolding, suggested there, imposes a strict FIFO order on the consumption of the tokens from an arc with the activity more than 1 (a "CD equivalent" of an unsafe place). Under this assumption the tokens are consumed in the order of their arrival onto this arc and, therefore, the transition can consume only one token. Thus only one instance of t will be created in the CD-unfolding. It is possible to simulate such behaviour in a PN by substituting any unsafe but bounded place with a pipeline structure which is capable of accumulating a bounded number of tokens; this structure will release tokens in the order of their arrival.

Consider the example shown in Figure 4.5(a). The PN-unfolding for this PN is shown in Figure 4.5(b). As it can be seen, there are two instances of t_2, one of which, t''_2, is redundant as the final state of its configuration is already represented by t'_2. This behaviour can be equivalently described by a CD shown in Figure 4.5(c) whose CD unfolding is shown in

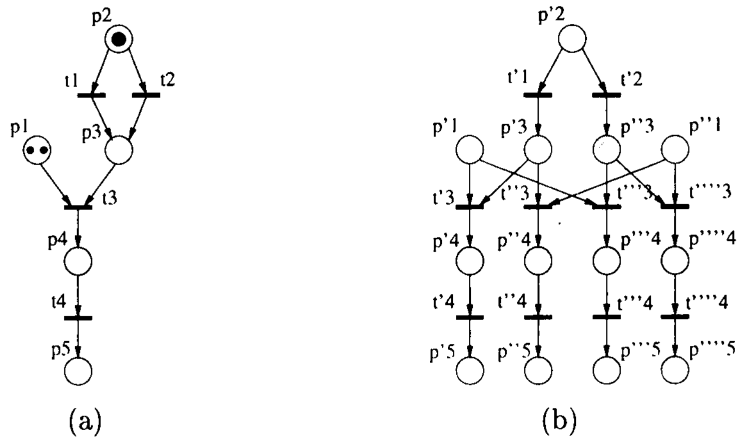


Figure 4.6: Example of a PN and its truncated PN-unfolding

Figure 4.5(d). The CD unfolding has only one instance of t_2 and thus does not have this redundancy. However, the unsafe place p_2, which causes redundancy, can be substituted by a pipeline-like structure as shown in Figure 4.5(e), and, hence, the FIFO order is imposed on the token consumption. The PN-unfolding of the transformed PN is given in Figure 4.5(f), it has only one instance of t_2. The part of the PN-unfolding of the transformed PN corresponding to place p_2 of the unsafe PN is highlighted by a dashed line.

Note that the result of such transformation is a safe PN. However, the FIFO discipline on the order of token consumption in CDs can be assumed due to the fact that they are choice-free. Therefore, the transformation illustrated above can only be applied to choice-free specifications, e.g. MGPN.

Several attempts were made to enhance the truncated PN-unfolding condition. In [39] a seemingly simple cutoff condition was suggested which allows a transition with the equal size of its local configuration. It has, however, been shown to be incorrect first in [76] for bounded PNs and then in [22] for safe PNs. Esparza et. al. [22] suggested an efficient algorithm for constructing a finite prefix of PN-unfolding from safe PNs. In this work it was noted that a total order can be imposed between configurations in the unfolding of a safe PN. This total order corresponds to the total order in which instances of transitions in the unfolding are constructed. Thus it can be used to decide which of the instances is a cutoff transition.

The following section presents the result of [22] and then suggests a truncation technique to overcome the redundancy associated with unsafe places. A segment of the PN-unfolding which does not contain the redundant copies of transitions is called a PN-**unfolding segment**.

4.2.2 PN-unfolding Segment

Consider an example of a PN in Figure 4.6(a) with its truncated PN-unfolding shown in Figure 4.6(b). As it can be seen, the truncated unfolding contains redundant instances of transitions t_3 and t_4, e.g. instances t'_3 and t''_3 have equal final state of their local configurations and, therefore represent the same reachable marking (p_1,p_4). Consider first the redundancy associated with arrival of tokens into one place through different paths.

Let there exist an arbitrary total order $\ll$ which is imposed on the transitions of a PN N. The order $\ll$ is extended to arbitrary connected partially ordered sets of instances of the

PN-unfolding as follows. For a set of events T'', let $\varphi(T'')$ be that sequence of transitions which is ordered according to $\ll$ and contains each transition t as often as there are events in T'' with the label t. $\varphi(T')$ is called the *signature of* T''. Now, it is said that $T_1'' \ll T_2''$ if $\varphi(T_1'')$ is shorter than $\varphi(T_2'')$, or if they are the same length but $\varphi(T_1'')$ is lexicographically smaller than $\varphi(T_2'')$. Note that T_1'' and T_2'' are incomparable with respect to $\ll$ if $\varphi(T_1'') = \varphi(T_2'')$. In particular, if T_1'' and T_2'' are incomparable with respect to $\ll$, then $|T_1''| = |T_2''|$.

Esparza *et. al.* [22] also introduced an isomorphism $I_{C_2}^{C_1}$ between two finite configurations. Let two finite configurations C_1 and C_2 be such that $F_s(C_1) = F_s(C_2)$ and $C_1 \subset C_2$. From definitions it follows that since $F_s(C_1) = F_s(C_2)$, then the extension of the unfolding from C_1 is isomorphic to the unfolding of the original PN where $M_0 = F_s(C_2)$; and therefore the unfolding extension from C_1 is isomorphic to the unfolding extension from C_2. Moreover, there is an isomorphism $I_{C_2}^{C_1}$ from the extensions from C_1 to the extensions from C_2. This isomorphism introduces a mapping from the finite extensions of C_1 onto the extensions of C_2; it maps $C_1 \cup T''$ onto $C_2 \cup I_{C_2}^{C_1}(T'')$. A finite extension T'' of some configuration C is also called a *suffix* to C. In [22] a notion of adequate order was introduced as follows.

Definition 4.2.1 [22] A partial order $\Subset$ on the finite configurations of PN-unfolding is called *adequate order* iff:

- $\Subset$ is well-founded, i.e. $\Subset$ is finite,
- $\Subset$ refines $\subset$, i.e. $C_1 \subset C_2$ implies $C_1 \Subset C_2$, and
- $\Subset$ is preserved by finite extensions, meaning that if $C_1 \Subset C_2$ and $F_s(C_1) = F_s(C_2)$, then $C_1 \cup T'' \Subset C_2 \cup I_{C_2}^{C_1}(T'')$, where T'' is a suffix to C_1.

□

It was observed in [22] that the algorithm and the cutoff condition can be presented in parametrised form where the parameter is an adequate order $\Subset$, i.e. a newly generated instance t' is a cutoff transition iff

$$\exists t_c' \in T' : F_s(\lceil t_c' \rceil) = F_s(\lceil t \rceil) \text{ and } \lceil t_c' \rceil \Subset \lceil t \rceil.$$

The parametrised version of the algorithm is given in Figure 4.7; the difference from the algorithm in Figure 3.10 is in the ordering of the newly generated instances in the QUEUE (a line marked with "**").

It was also noted that McMillan's order corresponds to a particular case of $\Subset$, i.e. the order

$$C_1 \Subset_M C_2 \Leftrightarrow |C_1| < |C_2|$$

which can be easily shown to be adequate. Furthermore, Esparza *et. al.* [22] introduced the following total order on suffixes of configurations.

Definition 4.2.2 [22] Let T_1'' and T_2'' be two suffixes of configurations of a PN-unfolding and let $Min(T_1'')$ and $Min(T_2'')$ be the sets of minimal elements of T_1'' and T_2'' with respect to the causal relation. Then $T_1'' \Subset_E T_2''$ if:

- $T_1'' \ll T_2''$, or

```
proc Build truncated unfolding($N = \langle P, T, F, M_0 \rangle$)
   Initialise $N'$ with instances of places in $M_0$
   Initialise QUEUE with $t$ enabled at $M_0$
   while QUEUE not empty do
      Pull $t$ from QUEUE
      Add $t'$ and $t'\bullet$ to $N'$
      if $t'$ is a cutoff then do
         Mark $t'$ and $t'\bullet$ as cutoff points
      end do
      for each $t$ in $T$ do
         Find unused set of mutually concurrent instances of places in $\bullet t$
            which are not successors of a cut-off transition
         if such set exists then do
**             Add $t$ to QUEUE according to the order $\Subset$
         end do
      end do
   end do
   return $N'$
end proc
```

Figure 4.7: Parametrised algorithm for truncated PN-unfolding.

- $\varphi(T_1'') = \varphi(T_2'')$ and
  - $Min(T_1'') \ll Min(T_2'')$, or
  - $\varphi(Min(T_1'')) = \varphi(Min(T_2''))$ and $T_1'' \setminus Min(T_1'') \Subset_E T_2'' \setminus Min(T_2'')$.

□

The following result is due to Esparza *et. al.* [22].

Theorem 4.2.1 [22] $\Subset_E$ is an adequate total order on the configurations of a PN-unfolding N' constructed for a safe PN N.

The proof of this theorem is done in [22] by proving the following statements:

- $\Subset_E$ is a partial order,
- $\Subset_E$ is total on configurations of N',
- $\Subset_E$ is well-founded,
- $\Subset_E$ refines $\subset$,
- $\Subset_E$ is preserved by finite extensions.

□

The interested reader is advised to consult the original proof for details. From the above theorem it follows that using the signature defined as above for every instance in the PN-unfolding it is possible to construct a finite prefix for safe PNs such that no two instances have equal final states of their local configurations. The cutoff condition is defined as follows.

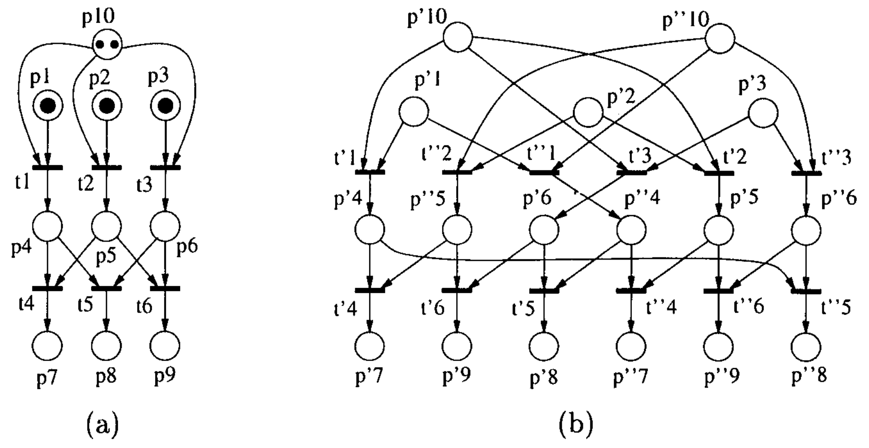


Figure 4.8: Example of a PN and its truncated PN-unfolding

Definition 4.2.3 Safe PN cutoff condition
A newly generated instance t'_c is called a *cutoff transition* iff there exists another instance t' such that

$$F_s(\lceil t' \rceil) = F_s(\lceil t'_c \rceil) \text{ and } \lceil t' \rceil \Subset_E \lceil t'_c \rceil$$

. □

The result of [22] is based on the fact that all configurations in a PN-unfolding built for a safe PN are totally ordered. It has demonstrated significant savings in computational time and space when applied to analysis of safe PNs. However, this method is restricted to safe PNs only, as in an unsafe PN the order between two signatures ceases to be preserved by finite extensions. Nevertheless, the analysis of bounded PNs can be helped by means of pruning.

Consider another PN shown in Figure 4.8(a). The PN-unfolding built from it is shown in Figure 4.8(b). Obviously, some of the transition instances are redundant. However, consider a truncated PN-unfolding constructed for the same PN and shown in Figure 4.9. The order of instances added to the unfolding was $t'_1, t''_1, t'_2, t''_2, t''_3, t'_3, t'_5, t''_6$. If the instances were made cutoffs as shown, then this PN-unfolding does not represent one of the reachable markings, namely p_3, p_7 reached through the firing of transition t_4. Note, that whatever order of instances was chosen, at least one reachable marking of the original PN will not be represented in the PN-unfolding. In fact, the instance t''_2 (not t'_2) and both instances of t_3 must be kept in order to have all reachable markings represented in the unfolding[1].

The above example demonstrates that further decision about an instance being a cutoff in the unfolding of a bounded non-autoconcurrent PN can only be made after taking into account the fact that this instance may produce new instances in the future. Intuitively, this may be illustrated as follows. Suppose there are two instances t'_1 and t''_1 which have equal final states of their local configurations and are in conflict. Suppose also that there exists a third instance t'_2 which is concurrent to t'_1 but there is no instance of t_2 concurrent to t''_1. In this case, if t'_1 is made a cutoff point, then no instances which include t'_1 and t'_2 into their local configurations could be constructed. At the same time, since no instance of t_2,

[1]Obviously the choice of kept instances depends on the order of added instances in the unfolding.

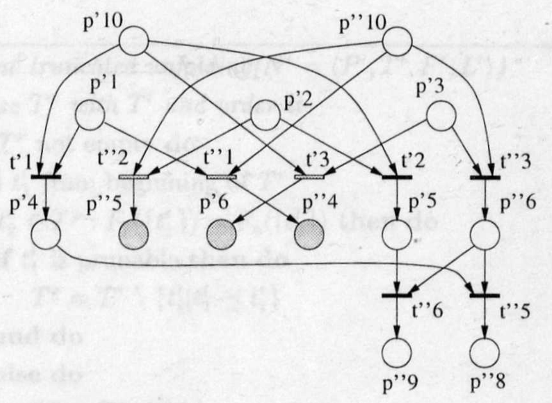


Figure 4.9: Truncated PN-unfolding with wrong cutoffs.

which is concurrent to t_1'', is present in the unfolding, no instances including t_1'' and t_2'' can be constructed either. Therefore, the set of transitions which are concurrent to t_1' and t_2'' must be considered.

Note that in a non-autoconcurrent PN, all instances of a transition t are either sequential of in conflict. In particular, this means that all instances of t with the same length of their local configuration are in conflict.

To develop a condition which will allow the unfolding to be cut "on-the-fly" consider first the following pruning procedure. Suppose that all instances in the truncated PN-unfolding have been ordered, initially in the order of their instantiation. Such an ordering will range the instances of the unfolding in order of the sizes of their local configurations. Let also the instances of one transition which have equal final states of their local configurations be grouped together. In the process of pruning, all instances which have been tested for redundancy and have been decided to be kept in the unfolding will be deposited into the set T^p, which is initially empty.

The remaining ordered set of instances, which are still to be considered for redundancy, is denoted as T^r; initially $T^r = T'$. This set can be viewed as a queue. The pruning procedure starts at the beginning of T^r and works through it. At each iteration it removes an instance t_1' at the beginning of T^r. It then checks if there exists another instance t_2' in T^p such that $F_s(\lceil t_1' \rceil) = F_s(\lceil t_2' \rceil)$. Obviously, if t_2' exists, then the sizes of $\lceil t_1' \rceil$ and $\lceil t_2' \rceil$ are equal, otherwise, t_1' would be a cutoff in the truncated PN-unfolding. If t_1' is decided to be prunable (using some condition, e.g. one suggested later), then it and all its successors are discarded from further consideration. Alternatively, t_1' is kept, and it is deposited into T^p. The pseudo-code of the pruning procedure is shown in Figure 4.10.

Consider now the redundancy condition for instances in the pruning algorithm. The relations between instances in the PN-unfolding are captured in the structure of the unfolding. Using the set T^p, a representative set is defined for each instance as follows.

Definition 4.2.4 A subgraph $\mathcal{E}(t') = \langle P'', T'', F'' \rangle$ of a PN-unfolding $N' = \langle P', T', F', L' \rangle$ is called a *representative set of instance* t' iff

- $T'' \subseteq T^p$
- $\forall t'' \in T'' : (t' \| t'') \wedge (\bullet t'' \cup t'' \bullet \subseteq P'')$

```
proc Prune truncated unfolding($N' = \langle P', T', F', L' \rangle$)
  Initialise $T^r$ with $T'$ and order it
  while $T^r$ not empty do
    Pull $t'_1$ from beginning of $T^r$
    if $\exists t'_2 \in T^p : F_s(\lceil t'_1 \rceil) = F_s(\lceil t'_2 \rceil)$ then do
      if $t'_1$ is prunable then do
        $T^r = T^r \setminus \{t'_i | t'_1 \preceq t'_i\}$
      end do
      else do
        $T^p = T^p \cup \{t'_1\}$
      end do
    end do
  end do
  return $T^p$
end proc
```

Figure 4.10: Algorithm for pruning a truncated PN-unfolding.

- no instance $t'' \in T''$ is a cutoff transition.

□

In other words, the representative set of an instance t' is a subgraph of the unfolding such that every transition instance of this subgraph is concurrent to t', and it is restricted to include all those transition instances of the PN-unfolding which have already been decided to be kept by the pruning procedure.

It is possible to define a *morphism with respect to labelling function* $L' : P' \cup T' \to P \cup T$ between two subgraphs of the PN-unfolding.

Definition 4.2.5 A morphism between two subgraphs $\mathcal{E}_1$ and $\mathcal{E}_2$ of the PN-unfolding N' with respect to the PN-unfolding labelling function $L' : P' \cup T' \to P \cup T$ is a mapping $h : P'_1 \cup T'_1 \to P'_2 \cup T'_2$ such that:

- $\forall p'_1 \in P'_1, h(p'_1) = p'_2 \in P'_2 : L'(p'_1) = L(p'_2)$;
- $\forall t'_1 \in T'_1, h(t'_1) = t'_2 \in T'_2 : L'(t'_1) = L(t'_2)$;
- for every $t'_1 \in T'_1$, the restriction of h to $\bullet t'_1$ is a bijection between $\bullet t'_1$ (in $\mathcal{E}_1$) and $\bullet h(t'_1)$ (in $\mathcal{E}_2$), and similarly for $t'_1\bullet$ and $h(t'_1)\bullet$.

□

In other words, the morphism defined above is a mapping between two subgraphs of the unfolding which preserves the nature of nodes and the environments of transition instances. In addition, h requires that two nodes $x'_1, x'_2 : h(x'_1) = x'_2$ of the subgraphs map onto the same node of the original PN. Note that the definition of morphism h is asymmetric, not every node in the subgraph $\mathcal{E}_2$ is required to have a corresponding node in $\mathcal{E}_1$, i.e. $\mathcal{E}_2$ may be larger than $\mathcal{E}_1$.

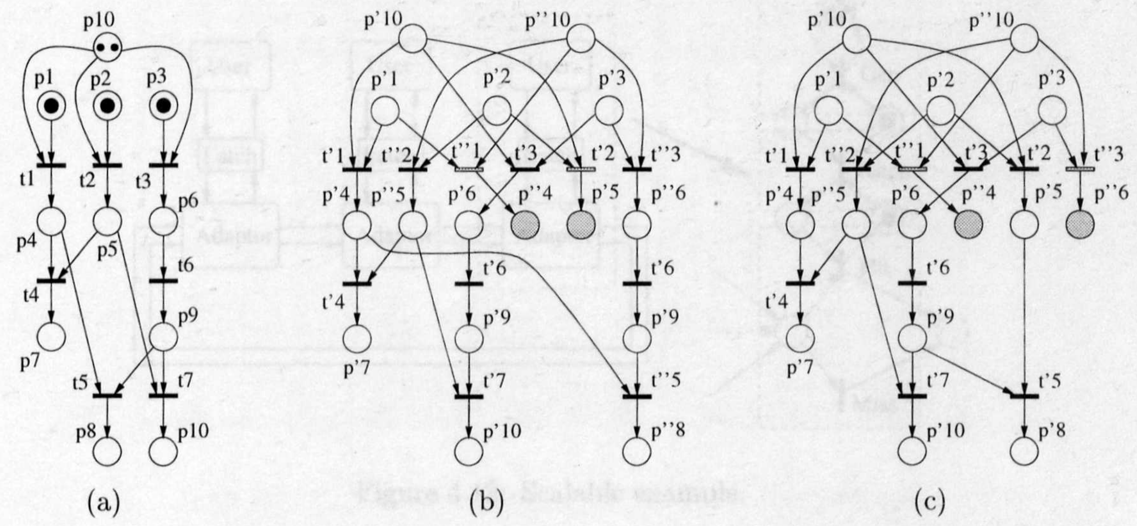


Figure 4.11: Illustration of redundancy in a PN-unfolding segment.

Using the notions of the representative set of an instance and subgraph morphism it is possible to define a prunable instance as follows.

Definition 4.2.6 An instance t' of truncated PN-unfolding constructed for a non-autoconcurrent PN is called *prunable* iff there exists another instance t'' of the same transition t such that $|\lceil t' \rceil| = |\lceil t'' \rceil|$ and $h(\mathcal{E}(t')) = \mathcal{E}(t'')$, where $\mathcal{E}(t')$ and $\mathcal{E}(t'')$ are the representative sets of t' and t'' respectively. □

Thus, in the algorithm in Figure 4.10, if $|\lceil t' \rceil| = |\lceil t'' \rceil|$ and $h(\mathcal{E}_2) = \mathcal{E}_1$ for the representative sets of t'' and t', then t'' is pruned from the truncated PN-unfolding.

Note that the order of instances in T' is the same as the order of generation of instances in the algorithm in Figure 4.7 up to permutations of instances with the same length of local configurations. Therefore, the pruning procedure can be applied along with the procedure deciding if a newly generated instance is a cutoff, i.e. it can be done "on-the-fly". Hence the new cutoff condition is defined as follows.

Definition 4.2.7 Non-autoconcurrent PN cutoff A newly generated instance t'_c is called a *cutoff transition* in the PN-unfolding constructed for a non-autoconcurrent PN iff there exists another instance t' such that

$$F_s(\lceil t' \rceil) = F_s(\lceil t'_c \rceil) \text{ and } \left[\begin{array}{l} |\lceil t' \rceil| < |\lceil t'_c \rceil|, \text{ or} \\ \exists h : h(\mathcal{E}(t')) = \mathcal{E}(t'_c), (\text{if } L'(t') = L'(t'_c)) \end{array} \right.$$

□

The finite fragment of a PN-unfolding constructed using the cutoff condition from Definition 4.2.7 is called a PN-*unfolding segment.*

The algorithm does not always produce a minimal segment as it is illustrated in Figure 4.11. Two possible PN-unfolding segments for a PN from Figure 4.11(a) are shown in Figures 4.11(b) and 4.11(c). As it can be seen, the segment in Figure 4.11(b) has one more instance of t_6 compared to the segment in Figure 4.11(c). Which of the segments is constructed for this particular PN depends on the order in which the transitions of the original

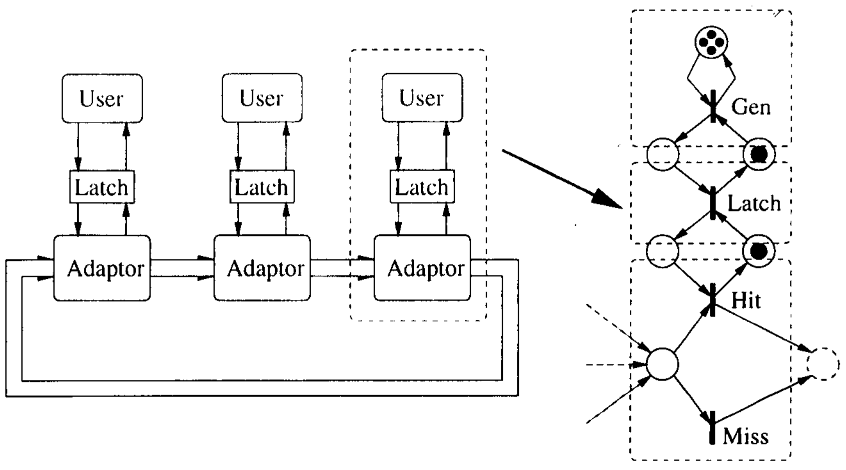


Figure 4.12: Scalable example.

PN are considered. The algorithm attempts to predict the future behaviour of the PN from incomplete information. This is a trade off between efficiency and accuracy of the algorithm. Obviously, it is possible to develop an algorithm which would be an "inter-breed" of both methods. On each step the algorithm would construct a layer of PN-unfolding which contains all concurrent transitions for instances with the current length of their local configurations. Then the redundant copies of transitions would be pruned and the algorithm would go on to the next step.

4.3 Performance Comparison of Algorithms

To compare the performance of different approaches the algorithms described above were implemented in the analysis tool PUNT. Although Esparza's algorithms can be applied to safe PNs only, this has not been an obstacle since it is possible to detect unsafeness of most benchmarks is known or can be detected during unfolding. Thus if a particular benchmark is reported to be unsafe using Esparza's algorithm, then the unfolding process is stopped and the tool is restarted using the cutoff condition based on pruning.

The performance comparison of Esparza's algorithm with the original McMillan's algorithm was presented elsewhere [22]; therefore the PN-unfolding segment algorithm based on the pruning cutoff condition suggested in the previous section is compared with the the original McMillan's algorithm here. The comparison uses two realistic examples; the first one is an example of a token ring protocol, and the second is a production cell example.

4.3.1 Ring Protocol

Token ring protocols, e.g. [100], are common means of communication between two or more users. The communication protocol, chosen here for illustrative purposes, is a somewhat simplified version of a real-life communication protocol. Each user is connected to an adapter which is responsible for access to the ring. The user prepares a data packet and issues a request to its adapter, informing it that the data is ready. Thus the user instructs the adapter to obtain access to the ring and to send the data down the ring to the next one. After the data has been sent, the adapter acknowledges this fact to the user who in turn prepares new

Benchmark		McMillan		Experim.	
No. ads	No. tok.	Time	Size [tr/pl]	Time	Size [tr/pl]
2	1	0.08	10/25 (3/5)	0.07	10/25 (3/5)
2	3	0.17	38/85 (7/13)	0.09	18/45 (11/21)
2	10	2.04	262/547 (21/41)	0.40	46/115 (39/77)
2	20	17.79	922/1887 (41/81)	1.50	86/215 (79/157)
5	1	0.12	25/61 (6/11)	0.13	25/61 (6/11)
5	3	0.39	95/211 (16/31)	0.28	45/111 (26/51)
5	10	6.30	655/1366 (51/101)	1.84	115/286 (96/191)
5	20	53.75	2305/4716 (101/201)	9.42	215/536 (196/391)
10	1	0.27	50/121 (11/21)	0.27	50/121 (11/21)
10	3	1.09	190/421 (31/61)	0.74	90/221 (51/101)
10	10	18.01	1310/2731 (101/201)	7.13	230/571 (191/381)
10	20	148.98	4610/9431 (201/401)	38.99	430/1071 (391/781)
20	1	0.72	100/241 (21/41)	0.71	100/241 (21/41)
20	3	3.72	380/841 (61/121)	2.85	180/441 (101/201)
20	10	67.21	2620/5461 (201/401)	29.97	460/1141 (381/761)
20	20	525.95	9220/18861 (401/801)	165.36	860/2141 (781/1561)

Table 4.1: Experimental results for the token ring protocol.

data and issues a new request.

Adapters are connected between themselves in a ring fashion so that the data propagates in one direction. The fact that an adapter got its turn to send the data is indicated by the arrival of a special data package on its inputs which is usually called a "token". Upon the receipt of the token, the adapter issues a request to its neighbour to indicate that it is sending a data packet from its user or to pass the token on, if there was no data to send. The overall organisation of such communication mechanism is shown in Figure 4.12. The direction of the data flow in this example is clockwise.

Adapters are responsible for arbitrating input requests coming from its user and its left-hand side neighbour. That is, if both requests arrive close to each other, the adapter decides which one is the winner and processes the winning request. If the user' request was the winner (operation called "Hit"), then the adapter removes the token and sends a data packet from its user down the ring. If the user's request was late (operation called "Miss"), then the adapter simply passes the token over. Any data packet from the left-hand side neighbour which is destined for the adapter's user is removed and the outstanding user's packet is sent down the ring.

To make the user's operation more independent from its adapter they are decoupled using a latch. That is, any data packet coming from the user to be sent down the ring is latched after which the user may continue with its other activity. Any data coming from the adapter is also latched so that the user may take it whenever it is ready.

The user is assumed to be able to prepare several data packages which are stored in a pool of data. Packets are sent from this pool regardless of their order.

The PN model of the control for one of the adapters, its corresponding user and a latch is shown in Figure 4.12. The behaviour of an adapter is similar for any type of packet on its input (data or token), and therefore one transition is used for each possible activity. The pool of user's data is represented by an unsafe place where the number of tokens in the place indicates the maximum number of prepared data packages, in this case four. Transition "Gen" represents the request generation from the user. The PN model for the whole ring is obtained by connecting the required number of copies of this net together. The resulting PN is a non-autoconcurrent PN and therefore it can be verified using the new PN-unfolding segment

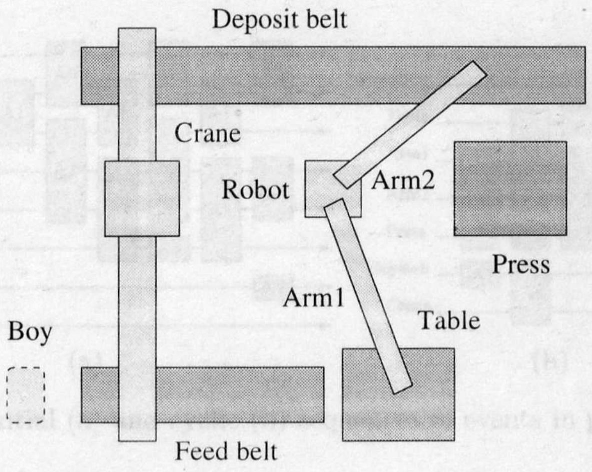


Figure 4.13: Illustration of a production cell.

algorithm.

The experimental results for this example were obtained on a Sun SPARC20 machine and are shown in Table 4.1. Column "Benchmark" presents the number of adapters in the ring and the number of tokens in the unsafe place. The rest of the Table presents the results of constructing the truncated PN-unfolding using McMillan's cutoff condition (columns under "McMillan") and the PN-unfolding segment (columns under "Experim."). For each algorithm two parameters were measured: the time (in seconds) required to construct a finite fragment of the unfolding and its size. The size of the fragment is measured in the number of transition and place instances, separated by a "/". The Table also shows the number of cutoff transitions and places which are their successors (corresponding numbers in brackets in columns "Size").

From the results is can be observed that the PN-unfolding segment algorithm using the cutoff condition based on pruning showed significant gain in speed and reduction in the fragment size for this set of benchmarks. For the PN model of a token ring with 20 adapters and the pool size of 20 the PN-unfolding segment had only 860 transition instances. This gain is due to eliminating conflicting instances of transitions with the same sizes of local configurations which do not represent any new markings. This reduction is significant as most of the algorithms that verify properties of the original PN using the fragment of its unfolding have complexity dependent on the number of instances (transitions and/or places) in the fragment. Note also that most of the transition instances in the PN-unfolding segment were cutoff transitions.

4.3.2 Production Cell

Another example for illustrating the performance of the new method is the model of a production cell presented in [42] as a case study of an industry-oriented problem.

The production cell processes metal blanks which are conveyed to a press by a feed belt. From the feed belt the blank goes to a feeding table. A robot picks the blank up with its Arm1 and deposits it into the press. After that the press processes the blank. The robot uses its Arm2 to remove the forged metal plate from the press and puts it onto the deposit belt. The traveling crane picks the plate from the deposit belt and moves towards the feed belt. On its way it drops the processed plate and picks up a new blank from an infinite supply of

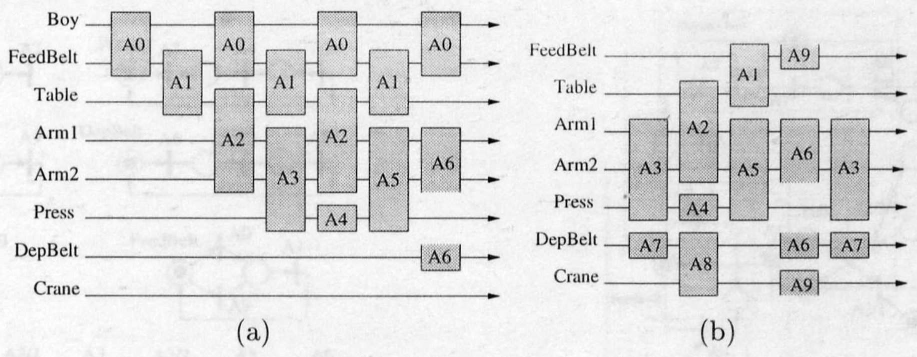


Figure 4.14: Initial (a) and cyclic (b) sequences of events in production cell.

blanks. It acts as a link between the two belts that makes it possible to let the model function continuously. The production cell is configured so that several metal plates can be processed and transported continuously; this should allow optimal utilisation of the cell capacity. The schematic view of a production cell is shown in Figure 4.13.

The initialisation of the cell is done by a boy who deposits initial blanks onto the feed belt. He has a limited number of blanks to deposit into the cell.

The task description is given in terms of Coordinated Atomic Actions [96], i.e. actions coordinating the interaction of two elements of the cell. Part of the specification also includes graphical representation of sequences of executed actions, similar to timing diagrams in circuit design. Two sequences are distinguished: the initialisation sequence, in which the boy deposits the first few blanks into the cell, and the cyclic sequence, which represents activity of the cell after the initial blanks have been deposited. The graphical representation for both sequences is shown in Figure 4.14(a) (for 4 initial blanks) and Figure 4.14(b).

The meaning of the actions is self explanatory. For example, action A0 means that the boy deposits a blank onto the feed belt and requires the coordination of interaction between the boy and the feed belt; action A3 represents depositing a blank from the table in to the press and requires the coordination of interaction between both arms of the robot and the press.

However, the specification can be considered from the view point of elements of the production cell. It is easy to construct models for each of the elements which synchronise whenever an action needs to be performed. In terms of PNs, this means that a PN model representing the behaviour of each element is derived from the sequences, and then these PN fragments are put together by merging the transitions with the same name. The set of PN fragments for the elements of the production cell is shown in Figure 4.15(a). Note that the PN for Arm1/Arm2 has two transitions labelled with A2/1 and A2/2. The former represents the very first occurrence of A2 and the latter represents A2 occurring when the system is in cycle. Formally this is done by the synchronisation operation between LPNs [66]. The composed PN is shown in Figure 4.15(b) where the fragments corresponding to each element of the production cell are shown by a dashed box. The unsafe place represents the number of blanks available to the boy for the initialisation of the cell, 4 as required in the initial specification.

The resulting PN was analysed using the existing unfolding techniques. The results of analysis are shown in Table 4.2. The PN model was analysed with a variable number of

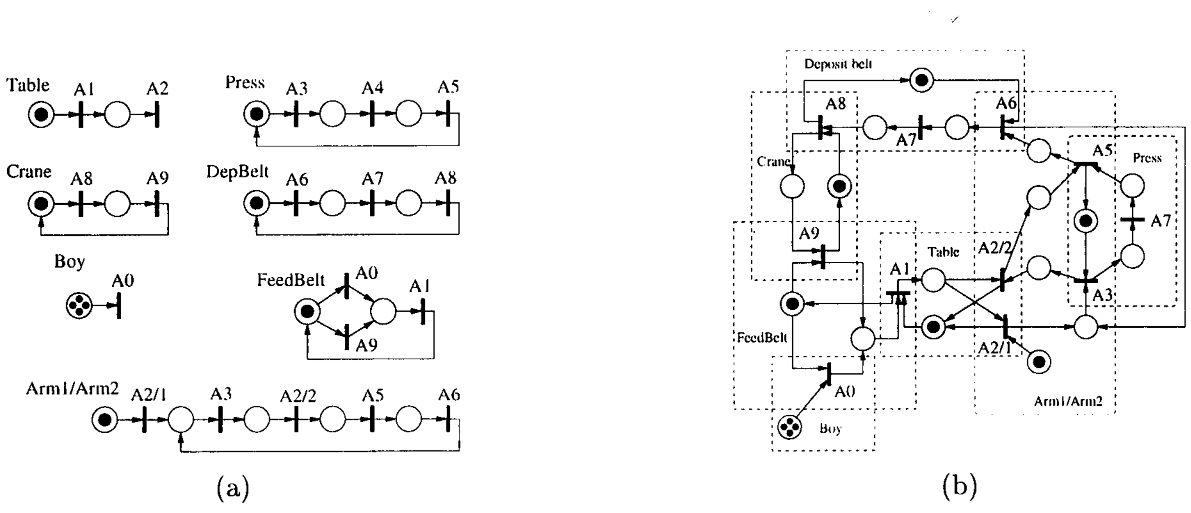


Figure 4.15: PN fragments (a) and the complete PN model (b) for production cell.

Tokens	McMillan		Experim.		Deadlock
	Size [tr/pl]	Time	Size [tr/pl]	Time	
1	5/17 (0/0)	0.10	5/17 (0/0)	0.08	✓
2	32/67 (2/4)	0.19	17/40 (2/4)	0.12	
3	135/253 (12/24)	1.02	28/61 (5/10)	0.18	
4	488/911 (60/120)	8.88	36/78 (9/18)	0.29	
5	1745/3337 (320/640)	106.08	43/93 (14/28)	0.36	
6	****	****	49/106 (19/38)	0.49	✓
10	****	****	73/158 (43/86)	1.11	✓

Table 4.2: Experimental results for the production cell.

tokens in the unsafe place. Column "McMillan" presents the results of truncated PN-unfolding construction using McMillan's cutoff condition. Column "Experim." presents the results of analysis using the PN-unfolding segment suggested in the previous section.

The overall aim of the analysis was to show the sensitivity of the cell to the number of the initially available blanks. The results showed that the model of the production cell is deadlock free[2] when there are 4 blanks available to the boy (line 4 of Table 4.2). Further investigation revealed more interesting results. The production cell was shown to be deadlocking if the number of initially available blanks is 1. However, it also showed that 2 blanks are enough to set the production cell into the cyclic, deadlock free operation, i.e. the initial requirement of 4 blanks is excessive. It was also determined that the cell will deadlock if the number of initial blanks is more than 5. This means that if the boy sends too many blanks into the production cell, it will fill up and come to a halt.

As it can be seen from the results, the PN-unfolding segment showed substantial gain in speed and reduction in the size of the constructed PN-unfolding fragment in comparison with the existing methods. Furthermore, the construction of the truncated PN unfolding failed due to the size of the unfolding for the case of 6 tokens. The analysis of the cell model using any of the existing PN-unfolding methods would not be able to detect the above-mentioned deadlock in the system. The PN-unfolding segment approach did not have any problems in constructing the segment for the case of 10 initial blanks and detecting the deadlock there.

[2]See the next section for discussion about the deadlock freedom check using the PN-unfolding.

4.4 Analysis of LPN Models

This section describes an application of the PN-unfolding segment to the analysis of LPNs. On one hand, LPNs are used to specify the behaviour of the future system or circuit. After the specification was found to be correct, its implementation is obtained by one of the existing techniques, e.g. syntax driven translation. On the other hand, for circuits designed in an "ad hoc" manner, it needs to be shown that they correctly implement the desired behaviour. In this case the circuit is converted to an LPN model which is then verified along with the model of its environment. Circuits built using two-phase protocol have symmetric operation on both up and down signal transitions. Four-phase circuits, however, may have different functions for up and down transitions of the output signal of a gate. A way to model these circuits using LPNs was shown in [47, 35, 101]. However, some properties are hard to check using this model. For example, to detect an illegal adjacency of two up transitions without an intermediate down transition requires checking all feasible sequences of the LPN. Verification of such properties is simpler using the STG based model where this notion is inherent in the validity of the STG. Hence, in this Section, only models of two-phase circuits are considered. The verification of four-phase circuits is presented later.

4.4.1 Verification of LPN Models of Specifications

As discussed in the previous Chapter, the PN-unfolding segment represents every reachable marking M as a final state of some configuration C and, moreover, all transitions enabled at M are represented as instances whose inputs are in $C\bullet$. Hence, any PN conflict can be pin-pointed by finding the instances of places with more than one output arc. Furthermore, if a PN-unfolding segment was constructed successfully, then the LPN is bounded and finite. The safeness of an LPN can be checked as the existence of two concurrent instances of one place. Thus the following important properties of LPNs can be verified:

- Boundedness (safeness), which indicates that the system can be implemented using a finite number of components. In addition, an unsafe place may indicate that a transition becomes autoconcurrent and hence may cause confusion in the system's operation.

- Action persistency, which indicates that transitions of the system cannot disable each other. Whereas this is usually allowed in the environment, such behaviour in the system usually means that some action may be made disabled before its completion and hence lead to a hazard.

- Liveness, where a non-live action indicates that the system never performs an action.

- Deadlock freedom, which indicates that a system never reaches a terminal state. In any system which is supposed to operate in cycles, a presence of a deadlock is regarded as an error.

If the specification is unbounded, then it is not possible to implement this behaviour at all. The rest of the properties are responsible for the correct behaviour of the system.

All these properties can be detected from the PN-unfolding segment. Boundedness is checked during the construction of the PN-unfolding segment. An efficient branch and bound

```
begin proc Check live($N$,$N'$)
    for each $t \in T$ do
        if $\nexists t' \in T' : L'(t') = t$ then do
            Report non-live $t$
        end do
    end do
end proc
```

(a)

```
begin proc Check safe($N$,$N'$)
    for each $p \in P$ do
        Find instances of $p$
        if $\exists p', p'' : p' \| p''$ then do
            Report unsafe $p$
        end do
    end do
end proc
```

(b)

```
begin proc Check persistent($N'$)
    for each $p' \in P'$ such that $p'\bullet > 1$ do
        for each $t'_1, t'_2 \in p'\bullet$ do
            if $\exists C : (\bullet t'_1 \cup \bullet t'_2) \subseteq C\bullet$ then do
                if $\nexists \Lambda(t_1)$ enabled at $L'(C\bullet) \setminus (\bullet t_2) \cup (t_2\bullet)$ then do
                    Report non-persistent $\Lambda(t_1)$
                end do
                if $\nexists \Lambda(t_2)$ enabled at $L'(C\bullet) \setminus (\bullet t_1) \cup (t_1\bullet)$ then do
                    Report non-persistent $\Lambda(t_2)$
                end do
            end do
        end do
    end do
end proc
```

(c)

Figure 4.16: Algorithms for verification of properties in PN-unfolding segment: (a) Liveness check, (b) Safeness check and (c) Persistency check.

algorithm for deadlock detection was suggested by McMillan in [47]. Consider algorithms for boundedness, safeness, persistency and liveness checks separately.

Boundedness check. The PN-unfolding segment of an unbounded PN is infinite. During the execution a PN becomes unbounded if it reaches a marking M' and there exists another, already visited, marking M such that $M \subset M'$ and there exits a sequence $\sigma : M \xrightarrow{\sigma} M'$. This is also true for markings which are represented in the PN-unfolding segment by final states of local configurations. Therefore, if for a newly generated instance t'' there exists another instance t' such that $F_s(\lceil t' \rceil) \subset F_s(\lceil t'' \rceil)$ and $t' \prec t''$, then this PN is unbounded. Once the unboundedness is discovered, it is reported and the segment's construction terminates. Since the PN-unfolding segment construction algorithm already checks newly generated final states against the already existing ones, the boundedness verification can be efficiently incorporated into this check.

Safeness check. To verify the safeness of a place in a PN (or LPN), all instances of this place are found. If the place is unsafe, then there exist at least two instances which belong to the post-set of one configuration. These instances are concurrent. Hence, if two independent instances of one place are found, then this place is unsafe. The pseudo-code of the safeness check algorithm is shown in Figure 4.16(b). The complexity of the algorithm verifying the safeness of a PN is $O(|P'|^2 \times \tau_i)$, where τ_i is the complexity of the independence check for two instances. Since the complexity of the independence check is usually negligible, the complexity of the safeness check is square in the number of place instances in the PN-unfolding segment.

Persistency check. The persistency check is required to find those markings where two transitions of the underlying PN are in conflict. From the properties of the PN-unfolding segment (the unfolding is an acyclic graph) it follows that any extension of a configuration C cannot reach C again. If the same marking as $F_s(C)$ is reached from C, then it is the final state of another configuration $C' : C \subset C'$. Hence, any two transitions which are in conflict at $F_s(C)$ can be identified by finding C and examining its final state. By definition, two transitions t_1 and t_2 may be in conflict if $\bullet t_1 \cap \bullet t_2 \neq \emptyset$.

The pseudo-code for the action persistency check algorithm is shown in Figure 4.16(c). The algorithm iterates through all place instances p' with more than one successor. For each pair of transition instances t_1' and t_2' such that $p' \in \bullet t_1' \cap \bullet t_2'$ it attempts to find a minimal configuration C producing $\bullet t_1' + \bullet t_2'$, i.e. $\bullet t_1' + \bullet t_2' \subseteq C\bullet$. If such a configuration exists, then a state at which both t_1 and t_2 are enabled in conflict is reachable. Then the persistency of $\Lambda(t_1)$ is checked by firing t_2 from $M = F_s(C)$ and checking if $\Lambda(t_1)$ is still enabled at $F_s(C) \setminus (\bullet t_2) + (t_2\bullet)$. Similar for $\Lambda(t_2)$.

Note, that the problem of transition persistency for PNs is simpler than the problem of action persistency in LPNs. In a safe PN, a transition t_1 may only remain persistent if the firing of a conflicting transition t_2 returns tokens to all places in $\bullet t_1 + \bullet t_2$. Hence, t_1 is persistent with respect to t_2 at $F_s(C)$ if $L'(\bullet t_1') \subseteq (F_s(C) \setminus (\bullet t_2) + (t_2\bullet))$.

The complexity of the action persistency check is $O(|P'|)$, linear in the number of place instances in the PN-unfolding segment.

Liveness check. Weak liveness verification is the simplest check. A weakly live transition becomes enabled at least at one reachable marking on the PN. If a transition ever becomes enabled at any reachable marking, then this reachable marking will be represented in the PN-unfolding segment. The transition is then instantiated. Hence a transition is not live if there is no instance of this transition in the PN-unfolding segment. The pseudo-code of the liveness check algorithm is shown in Figure 4.16(a). The complexity of this algorithm is $O(|T'|)$, linear in the number of transition instances in the PN-unfolding segment.

Unfortunately, there is no easy way to check strong liveness of a PN transition. The problem of PN liveness is known to NP-complete for general PNs. There exist polynomial-time algorithms for examination of liveness for certain classes of PNs. These algorithms detect certain structural properties of PNs. The PN-unfolding approach is more suitable for determining other properties of the behaviour described by a PN. In the PN-unfolding segment strong liveness can be checked in the following manner. Firstly, the PN must be deadlock free, and therefore a branch-and-bound algorithm for deadlock detection [47] must be applied. Secondly, deadlock freedom does not guarantee strong liveness of the system, and, furthermore, for some examples a particular marking may only be reached once, yet the PN is strongly live. Hence, to detect a strongly live transition (or an action) in a PN-unfolding segment also requires examining (non-local) configurations of the segment to detect the strongly connected component and examine the liveness of transitions which leading to it.

Degree of concurrency. Another important characteristic of the asynchronous design is its degree of concurrency which is measured in the number of mutually concurrent events. Intuitively, the greater degree of concurrency, the more data is processed at the same time by different parts of the system/circuit. Thus, increasing the degree of concurrency increases the throughput of the circuit.

An example of using the PN-unfolding segment to determine the degree of concurrency was given in [75]. This work proposes a design of an asynchronous version for a simple synchronous microprocessor. The initial implementation was obtained using the refinement technique starting from the high-level description of the microprocessor. Several implementation versions were suggested which were aimed at enhancing the overall performance of the processor. The PN-unfolding segment analysis was used to extract the concurrency relation between transitions of the LPN representation of the microprocessor. It provided guidance for the designer, indicating which particular transitions can be decoupled (made concurrent) from the rest of the design.

4.4.2 Verification of LPN Models of Circuits

The problem of the verification of two-level circuits usually considers a circuit given in the form of a gate net-list or schematic. The environment of the circuit is described by a simple PN, e.g. representing delays (or inverted delays) in the environment from circuit outputs (requests) to circuit inputs (acknowledgements). The verification needs to find an answer to the question "Does a circuit behave without hazards in a particular environment?".

The original set of components used for building *two-phase* (see Subsection 2.1.2) control circuits was proposed by Sutherland [88]. They include the following:

- **C-element**, which implements AND-causality or synchronisation between different processes. The admissible behaviour of this element is such that both inputs are allowed to change their values, say from logical 1 to logical 0, if the output change, from logical 0 to logical 1, has been produced for the previous input change. This component is described by the boolean equation: $Y = x_1x_2 + y(x_1 + x_2)$, where Y is the output and x_1 and x_2 are the inputs of the C-element. The variable y is used to distinguish the feedback wire from the main output.

- **Merge**, or **eXclusive-OR (XOR)**, which realises OR-causality between input changes, but requires that **only** one input can change at a time. Input changes must therefore arrive on a mutually exclusive basis.

- **Toggle**, which switches between two outputs for every input change, as a complementary flip-flop. It is used to *unconditionally* alternate between two possible directions of control flow.

- **Select**, which changes one of the two output signals but, unlike Toggle, does it *conditionally*. The output is selected depending on the state of another input, the *level-based*

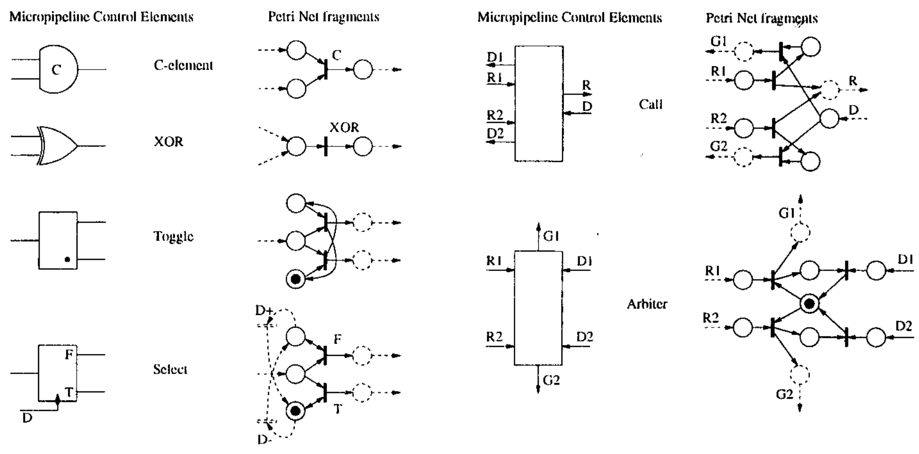


Figure 4.17: Basic two-phase control elements and their LPN models.

one. This component allows the construction of branches in control flow depending upon the state of the data path.

- **Call**, which operates like a control flow multiplexer. It transmits any of its alternative input requests to the single output request, and upon receiving the acknowledgement from the single acknowledgement input, transmits it to the acknowledgement output corresponding to the original request. This module therefore operates as an interface between different parts of the control flow and the single operational unit. It is crucial that input requests arrive in a mutually exclusive manner.

- **Request-Grant-Done (RGD) Arbiter**, which arbitrates between two possibly concurrent input requests (R) and generates only one grant (G) at a time, using a built-in metastability resolution circuit. To indicate that one of the requesters has finished a *critical section* of the computation, it uses another input, called "Done" (D).

These components are modelled by the PN fragments shown in Figure 4.17. The main idea behind this type of modelling is that the places are associated with input/output wires and the transitions with signal events. Since there is no distinction between rising and falling edges of transitions in the two-phase discipline, it is possible to associate one net transition with both.

The model of the Select element has a special feature. The complete model shows the effect of the environment which changes the state of input D (meaning "data"). Since D is a level-based, also called *four-phase* signal, its edges, denoted as $+D$ and $-D$, are not "symmetric", and must be modelled by separate net transitions. The figure does not show the origins of the logic which switches D.

To illustrate conversion of a circuit into an LPN consider the control circuitry for micropipeline FIFO structures. The basic two-phase FIFO structure, proposed by Sutherland [88], is shown in Figure 4.18(a). It is based on a data path storage element called *Capture Pass Element*.

The synchronisation between the data path and control in this FIFO is done using the bundled data technique, which requires using explicit delay elements between the stages. These compensate for delays and signal skewing in the bundles of data wires. Alternatively,

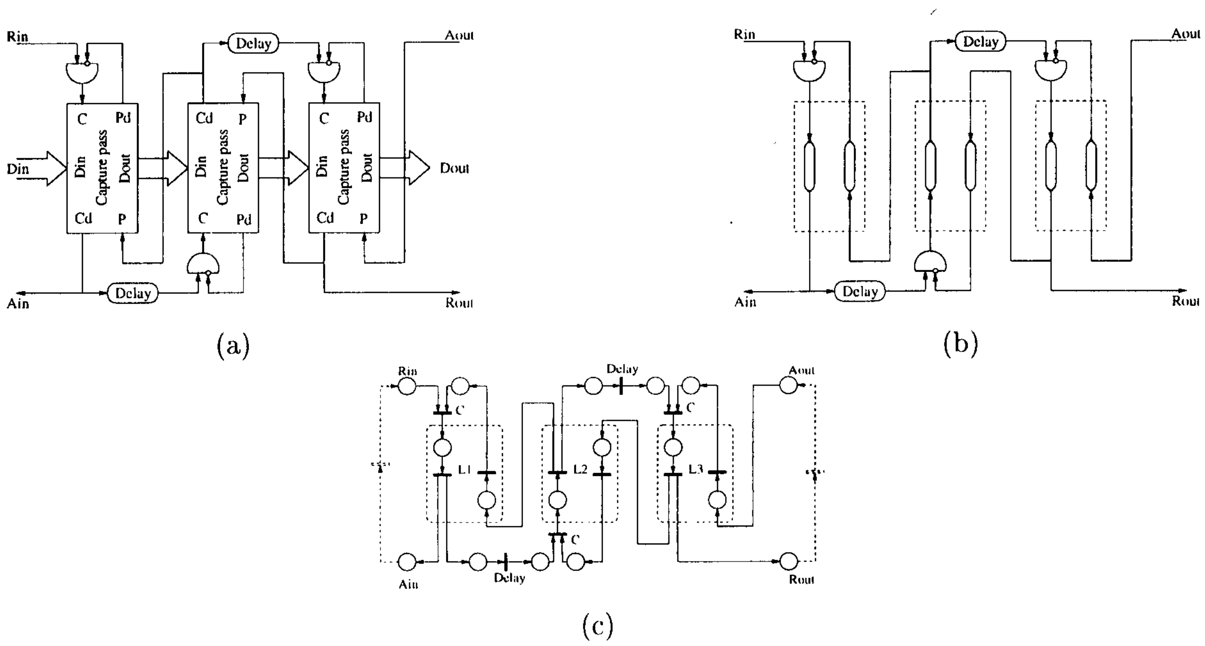


Figure 4.18: Sutherland's micropipeline FIFO (a), its control circuitry (b) and its LPN model (c).

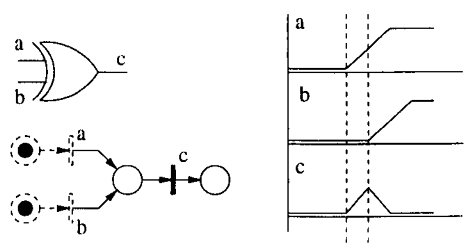


Figure 4.19: Illustration of a hazard of the gate's output.

it is possible to use dual-rail signalling [74], but this option is often discarded in practice as overly expensive in terms of area and power consumption.

Consider the modelling of control flow in such pipelines. The data path is abstracted away by means of modelling the delays in Capture Pass Elements as explicit delays inserted into the appropriate wires. The extracted control circuit for the FIFO micropipeline in Figure 4.18(a) is shown in Figure 4.18(b).

The operation of the FIFO stages is represented by the LPN shown in Figure 4.18(c). This net has a certain degree of redundancy that is caused by the explicit modelling of delays. It is easy to reduce this net to a simple loop with two transitions, each modelling the C-element that synchronises the request signal from the previous stage with an acknowledgement signal from the next stage.

The analysis of this LPN by means of the PN-unfolding segment shows that this net is live, safe, persistent and deadlock free. Hence the circuit is hazard free. Note that an unsafe place may indicate a hazard in a two-phase circuit. This is illustrated in Figure 4.19. This figure shows an XOR and its LPN model. Suppose that an input change caused the output of this gate to start its transition from one level to another. However, while the output is being changed, an event arrives at another input. This will cause the gate output to return to its

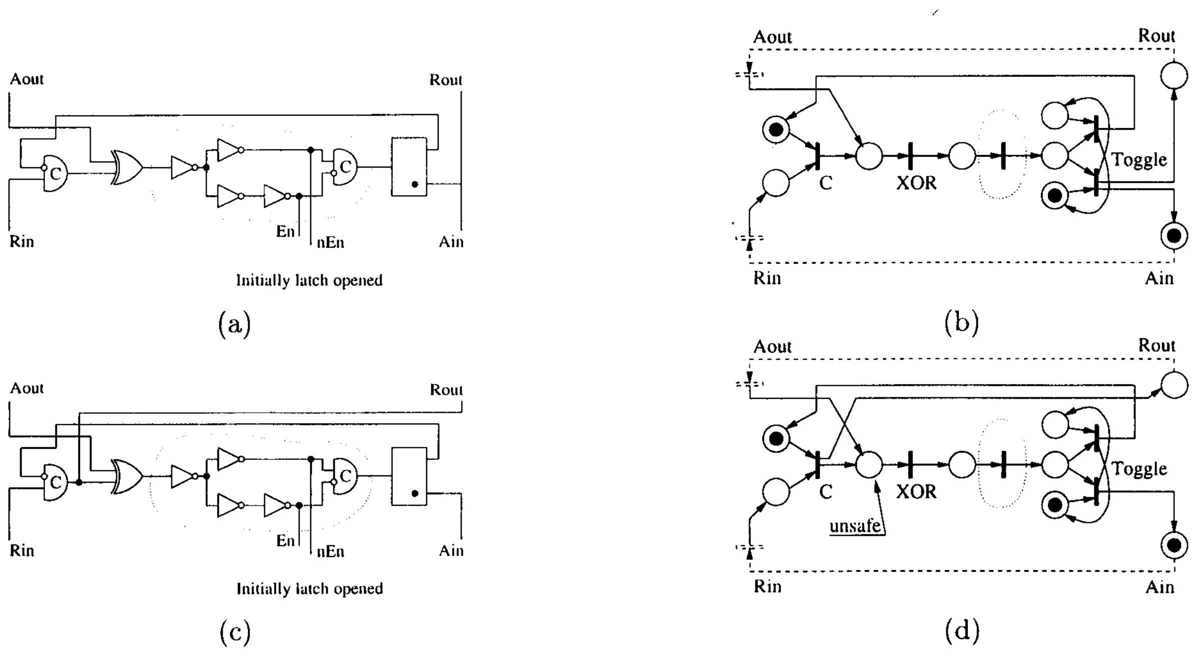


Figure 4.20: Standard (a-b) and Fast Forward (c-d) pipeline with four-phase latch control

original level, i.e. produce a short spike on the output. This is illustrated on the signal level diagram.

The performance of the pipeline is determined by the following two parameters. The first is the *latency*, which is the time it takes to propagate a datum through the stages. In Figure 4.18(a) this would be the time from *Rin* to *Rout*. The second performance factor is the *cycle time*, or the time it takes one stage to process one value and accept the next one. The maximum cycle time of all the stages determines the overall *throughput*. To achieve better performance, designers often perform low-level optimisations of circuits, sometimes relying on the delay ratios. In this case the circuit analysis is required to alert them to possible hazards.

Despite obvious performance gains achieved through the combined use of two-phase control with a Capture-Pass storage element, the designers of AMULET1 considered this design to be too costly in area and transistor count [64, 17]. They preferred to use conventional pass-transistor *Transparent Latch* circuits, which are four-phase operated. The latch is controlled by two complementary *enabling* signals, *En* and *nEn*. As a result, more complex control circuitry has to be used to convert the two-phase control of the interface between the stages to the four-phase control of the latches inside the stages and back. The conversion is done by means of a combination of a Merge (XOR gate) and a Toggle.

To illustrate the circuit analysis further, consider two (out of three) possible designs from [64, 17], based on the four-phase data path. The first, the *standard* one, is shown in Figure 4.20(a) for one stage. The PN-unfolding segment analysis shows that the control circuit is delay-insensitive. The second design, called *fast-forward pipeline*, has better performance (smaller forward latency) but at the cost of becoming sensitive to delays in the environment. It is shown in Figure 4.20(c). Its LPN model, if analysed without taking timing parameters into consideration, is unsafe. The place that models the Merge gate can be marked with more than one token. To avoid this, the designer must guarantee that the delays associated with

Stages	States	Versify		PUNT	
		τ_v	τ_t	Size [tr/pl]	τ_t
2	29	0.01	0.14	14/20 (1/1)	0.09
3	123	0.02	0.24	26/36 (1/1)	0.13
4	521	0.03	0.44	42/57 (1/1)	0.18
5	2207	0.05	0.69	62/83 (1/1)	0.33
10	3.01×10^6	0.21	6.60	222/288 (1/1)	2.90
15	4.11×10^9	0.61	35.27	482/618 (1/1)	14.69
20	5.60×10^{12}	1.08	112.73	842/1073 (1/1)	51.48

Table 4.3: Experimental results for the Sutherland's pipeline.

firing transitions corresponding to Merge, latch control circuitry, Toggle and the environment are carefully balanced and thus hazards are avoided. Such a constraint was satisfied in the AMULET1 design. It can be seen that the reverse latency in this design remains the same as in the previous design.

4.5 Experimental Results

Since STGs are a particular case of LPNs, the verification technique for LPNs is very similar to that of the verification of STG specifications. The results of the specification verification for a wide range of benchmarks can be found in Chapter 5.

To illustrate the performance of the circuit verification using the PN-unfolding segment the algorithms were applied to the example of the control circuitry of the micropipeline. Its model was shown in Figure 4.18(c).

The verification results obtained on a Sun SPARC20 are presented in Table 4.3. All examples were verified for safeness, persistency, liveness and deadlock freedom. Columns, grouped under "Versify" present the results of verification using the "state-of-the-art" verification tool `versify` [69] which uses Binary Decision Diagrams for the PN symbolic traversal. Columns τ_t give the total time (in seconds) spent on the construction of the behavioural representation and on the verification of properties for both tools. Column τ_v shows the fraction of time spent by `versify` on the actual verification of properties. Columns grouped under "PUNT" present the results of the verification using the PN-unfolding segment. Column "Size" shows the size of the PN-unfolding segment (and the number of cutoff transitions and their successors). As it can be seen, the method based on the PN-unfolding segment performs better than the state-based approach. Note also that most of the time spent by `versify` was used for construction of the RG representation. This is exactly where the PN-unfolding approach performs better.

4.6 Concluding Remarks

This chapter introduced a new concept of the PN-unfolding segment. First, it presented the result of [22] which can be applied to analysis of safe PNs. Then a new cutoff condition was suggested which avoids redundancy in the truncated PN-unfolding of a non-autoconcurrent PN. Such redundancy is associated with multiple instances of one transition. The new cutoff condition is based on the notion of a representation set and uses relations between transition instances of the unfolding.

The PN-unfolding segment was applied for the analysis of LPN models of asynchronous circuits and systems. The method was also applied to two real-life examples. The production cell example, in particular, demonstrated the need for the novel technique. The method was compared to other existing analysis tools and demonstrated favourable results.

Chapter 5

Analysis of STG models

A number of works, e.g [73, 13, 41], study the analysis of STGs using the assignment of consistent state vectors to the states in the RG of the underlying PN. This technique can be shown (although this has not been done formally) to be adequate only for cyclic STGs, while it is easy to show that for acyclic STGs or STGs with an acyclic segment such a vector assignment may not satisfy the intuition about the STG consistency. For example, an STG shown in Figure 5.1 is intuitively correct (because it is valid according to Definition 3.4.3), but there is no way to assign consistent state vectors to the markings of the underlying PN. There would either be two vectors associated with each of the markings $\{p_4\}$ and $\{p_5\}$ or the state vector assigned to $\{p_4\}$ will differ from the state vector assigned to $\{p_2\}$ and $\{p_3\}$ in two elements.

The purpose of this chapter is two-fold:

- To introduce a new notion of Full State Graph, which adequately captures the behaviour of an arbitrary STG; and
- To introduce an STG-unfolding segment which allows analysis of the STG behaviour by means of partial orders, based on the PN-unfolding segment introduced in the previous chapter.

5.1 Full State Graph

The purpose of this section is to introduce the notion of the **Full State Graph** (FSG). This new model correctly represents the behaviour of an arbitrary STG (including acyclic and combined STGs).

An STG **marking** is defined for an STG G as a reachable marking of its underlying PN N. All definitions from Section 3.1 for PN markings apply to STG markings. Any reachable marking M of an STG is associated with a binary vector v, called the *state vector*. Each state vector has exactly $|A|$ elements so that every element $v[i]$ corresponds to exactly one signal a_i from A. Unlike PNs, the dynamic behaviour of an STG must include the interpretation of its marking with the corresponding states of its signals in the marking. Indeed, it is always done for the initial state — one has to specify the initial marking and the initial state of an STG (although the latter is often done implicitly through the initial marking). It is therefore

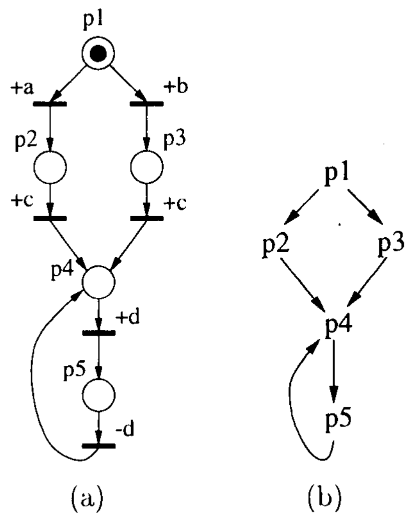


Figure 5.1: An STG (a) and its corresponding RG (b).

natural to define a full state which captures a marking together with the associated state vector.

Definition 5.1.1 A *full state* (FS) of an STG G is a pair $s = (M, v)$, where

- M is a reachable marking of N;
- v is a binary state vector.

□

Let $v[j]$ denote the element corresponding to signal $a_j \in A$. The binary state vectors for reachable markings can be found in the way that they satisfy the following requirement (also called the *consistency between an* FS *and a signal transition*):

Definition 5.1.2 A binary state vector v is said to be *assigned consistently to a marking M* (forming an FS $s = (M, v)$ of an STG) if for every FS $s' = (M', v')$ reachable from M by firing a transition labelled with a signal transition $*a_j$:

- if $M \xrightarrow{+a_j} M'$ and $v[j] = 0$, then $v'[j] = 1$
- if $M \xrightarrow{-a_j} M'$ and $v[j] = 1$, then $v'[j] = 0$
- $v[i] = v'[i]$ for all $i \neq j$.

The full state FS $s = (M, v)$ is called *consistent.* □

A state vector can not be assigned if the above conditions are not held.

An initial FS is denoted as $s_0 = (M_0, v_0)$. Several full states may correspond to one marking in the RG. It is natural to define equality between two FSs as

$$s_1 = s_2 \Leftrightarrow (M_1 = M_2) \wedge (v_1 = v_2).$$

The next definition lays out the concept of the *Full State Graph.*

```
proc Get FSG(G)
    Set $\Phi = \{s_0 = (M_0, v_0)\}$
    Push $s_0$ onto STACK
    DFS($\Phi$, STACK)
    return $\Phi$
end proc

proc DFS($\Phi$, STACK)
    while STACK not empty do
        Pop $s = (M, v)$ from STACK
        for each $t$ enabled at $M$ do
            if $*a_i = \Lambda(t)$ consistent with $v[i]$ then do
                Find $M'$ by firing $t$ from $M$
                Find $v'$ for $M'$
                if $s' = (M', v')$ not in $\Phi$ then do
                    $\Phi = \Phi \cup s'$
                    Push $s'$ onto STACK
                    DFS($\Phi$, STACK)
                end do
            end do
        end do
    end do
end proc
```

Figure 5.2: Algorithm for constructing an FSG from an STG.

Definition 5.1.3 A *Full State Graph* (FSG) is a quadruple $\Phi = \langle S, E, A, \Psi \rangle$ defined from an STG $\langle N, A, v_0, \Lambda \rangle$ such that

- S is a set of all consistent full states of the STG which are reachable from the initial full state $s_0 = (M_0, v_0)$ through valid firing sequences; $s_0 \in S$,
- E is a set of all arcs $e_{i,j} : s_i \rightarrow s_j$ connecting pairs of consistent full states $s_i = (M_i, v_i)$ and $s_j = (M_j, v_j)$ if there exists a transition $t \in T : M_i \xrightarrow{t} M_j$,
- A is a set of signals,
- $\Psi : E \rightarrow *A$ is a labelling function, which labels all arcs with signal transitions from $*A$.

□

The FSG represents those and only those FSs that are reachable from the initial FS and satisfy the condition of consistency between a full state and a signal transition. It follows from the above definitions that two state vectors of two FSs connected with an arc can only differ in one element. Note, however, that the definition of an FSG does not require two FSs to have equal state vectors if their marking components are equal, as it is presumed in the conventional definition of the SG.

The algorithm for the construction of an FSG for a given STG is shown in Figure 5.2. This algorithm is a straightforward adaptation of the basic algorithm for constructing of the RG for a PN [65]. The difference is in the construction of the new states. For each new state a binary vector satisfying the consistency conditions (laid down in Definition 5.1.2) is found. An example of the FSG built for the STG from Figure 5.1(a), where no consistent state assignment to its RG is possible, is shown in Figure 5.3.

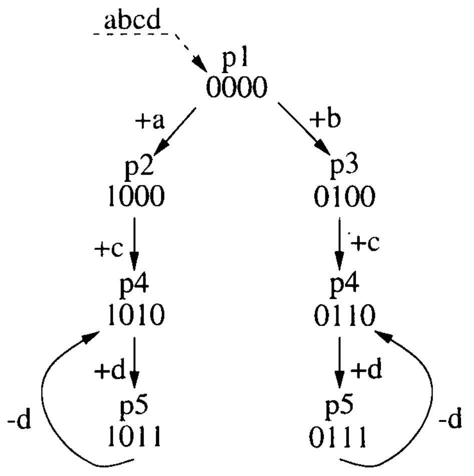


Figure 5.3: An FSG constructed for the STG from Figure 5.1(a).

As it can be seen, it is possible to start building the FSG and reach an FS $s = (M, v)$ from which a particular transition cannot be constructed. This can happen because of two reasons. Firstly, it may happen if there are no transitions enabled at M. Analogously to PNs, such a state is called *deadlock*. Secondly, if in such a state a signal transition $+a_j(-a_j)$ is enabled at marking M but the corresponding element in v is $1(0)$. Such a state is called *signal deadlock with respect to signal transition* $+a(-a)$ and is defined below.

Definition 5.1.4 A *signal deadlock with respect to signal transition* $+a_j(-a_j)$ is a state $s = (M, v)$ in the FSG such that:

$$\exists t : \bullet t \subseteq M \text{ and } \Lambda(t) = +a_j(-a_j),\ v[j] = 1(0),$$

where $v[j]$ is an element of the state vector corresponding to signal a_j. □

In other words, a signal deadlock with respect to $*a_j$ is a reachable FS s such that the value of $v[j]$ is inconsistent with the sign of the transition of a_j enabled at M.

Due to the condition of consistency and thus the potential presence of signal deadlocks, some of the reachable markings of the underlying PN may not be associated with state vectors. Noting this, the coverability of FSG is defined as follows:

Definition 5.1.5 The FSG, constructed for an STG, *covers* the RG obtained from its underlying PN if there are no signal deadlocks in the FSG.
Such an FSG is also called *consistent.* □

Example. Consider an STG shown in Figure 5.4(a). Its underlying PN has an RG shown in Figure 5.4(b). As it can be seen, the FSG (shown in Figure 5.4(c)) has two signal deadlocks with respect to $+b$ and hence it does not cover the RG. The initial state of the STG, corresponding to the initial placement of tokens, is 00 for the order of the signals *ab*. ■

Proposition 5.1.1 If there exist a valid feasible sequence $\sigma : M_0 \xrightarrow{\sigma} M_i$ (Definition 3.4.2), then there exists a full state $s_i = (M_i, v_i)$, where v_i is a signal state obtained by changing the initial state vector v_0 in the order of transitions in σ.

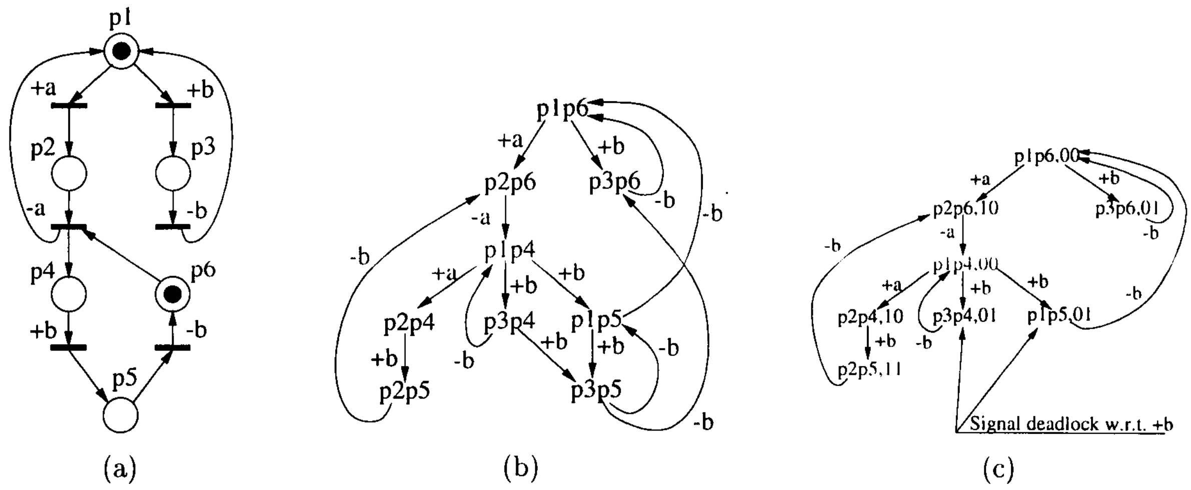


Figure 5.4: Example of an STG (a) and its RG (b) which is not covered by its FSG (c).

Proof: If σ is valid, then all signal changes in σ are consistent with values of binary vectors associated with all visited states. Hence all visited states (including s_i) exist in FSG.

□

Theorem 5.1.1 Let an STG have a bounded and finite underlying PN. The FSG built from an STG covers the RG built from the underlying PN iff this STG is valid.

Proof: [*if*] If an STG is valid then all conditions of Definition 3.4.3 hold; also, all feasible sequences generated by this STG are valid. It follows that no reachable state of this STG satisfies the conditions of Definition 5.1.4 and, hence, there are no signal deadlocks in the FSG built from this STG. Then, by Definition 5.1.5, such an FSG covers the RG constructed for the underlying PN.

[*only if*] Suppose that the STG is invalid. It needs to be shown that the FSG does not cover the RG obtained from the underlying PN. Note that there exist a signal, e.g. a_j, for which at least one of the conditions of Definition 3.4.2 is not true. Without loss of generality assume that $v_0[j] = 0$. Consider two possible cases:

1. In the underlying PN there exist a sequence $\sigma : M_0 \xrightarrow{\sigma} M_k$ such that $\sigma = \sigma_1 t_i \sigma_2$ and $\Lambda(t_i) = -a_j$, there are no instances of a_j preceding t_i and σ_1 is valid. According to Definition 5.1.4, $s = (M_k, v_k)$, where s is the full state reached after σ_1, is a signal deadlock with respect to the signal transition $-a_j$ (since $v_k[j] = 0$) and thus the FSG does not cover the RG.
2. In the underlying PN there exist a sequence $M_0 \xrightarrow{\sigma} M_k \xrightarrow{t_j}$ such that $\sigma = \sigma_1 t_i \sigma_2$ and $\Lambda(t_i) = \Lambda(t_j) = +a_j$, there are no instances of a_j in σ_2 and σ is valid. According to Definition 5.1.4, the FS $s_k = (M_k, v_k)$, which is reached after σ, is a signal deadlock with respect to the signal transition $+a_j$ (since $v_k[j] = 0$) and thus, as in the previous case, the FSG does not cover the RG.

Hence, the FSG does not cover the RG of the underlying PN for an invalid STG and therefore this part of the proof also holds.

□

It is clear that the STG, shown in Figure 5.3, is valid, according to Definition 5.1.5. This method gives an opportunity to verify the validity of an arbitrary STG based on Definition 3.4.3. However, this method has limited practicality due to its exponential complexity. The following sections describe the application of PN-unfolding to this problem and introduce the STG-unfolding segment. This is an event-based representation capturing the concurrency in its natural form, which allows the analysis of STGs and attempts to avoid the exponential growth of the explored number of states.

5.2 STG-unfolding Segment

In order to analyse an STG specification of a circuit, the PN-unfolding segment needs to be modified so that it covers the FSG of an STG. Similar to the presentation of the PN-unfolding, the STG-unfolding is introduced first, and then its truncation is suggested.

It is clear that marking components of the full states can be traced as final states of the PN-unfolding segment of the underlying PN. Due to the potential presence of signal deadlocks, configurations of the PN-unfolding segment may represent marking in the RG which will not be covered by the FSG. In order to be able to derive the state vector component the notion of a signal state of configuration is required.

Definition 5.2.1 Let N' be the the PN-unfolding constructed for the underlying PN of an STG G. The *signal state of configuration* C is a binary vector ξ, with $n = |A|$ components each of which corresponds to one and only one signal. It is defined iff for any signal $a \in A$ represented by $T^a \subseteq C, T^a = \{t'_i \mid |\Lambda(L'(t'_i))| = a\}$ the following is true:

- All signal changes of a are in total order such that $t'_1, t'_2 \dots t'_k$ (i.e. $\{t'_1, t'_2 \dots t'_k\} = T^a$) and $\forall i = 2 \dots k$: the sign of $\Lambda(L'(t'_{i-1}))$ is opposite to that of $\Lambda(L'(t'_i))$;
- the first signal transition $\Lambda(L'(t'_1))$ is consistent with the initial state of the STG.

The value of each element $\xi[j]$ is calculated as: $\xi[j] = \xi_0[j] \oplus |T^{a_j}|$, where $|T^{a_j}|$ is the number of occurrences of the transitions representing the signal $a_j \in A$ in the configuration C, $\oplus$ represents the modulo 2 sum and $\xi_0[j]$ is the signal state of $\lceil\perp\rceil$ equal to the initial state of G. □

A local configuration, satisfying the conditions of the above definition, also can have a signal state because it is a configuration.

Definition 5.2.2 An STG-*unfolding* is a tuple $G' = \langle N', \Xi \rangle$ where:

- N' - is an unfolding obtained from the PN underlying an STG;
- Ξ - is a set of defined signal states of local configurations $\xi_{\lceil t' \rceil}$ labelling local configurations $\lceil t' \rceil$.

□

Note that the PN-unfolding of the underlying PN may contain configurations whose signal state is undefined, e.g. configurations representing marking components of signal deadlocks.

Proposition 5.2.1 Every configuration $C' \subseteq C$ has a defined signal state iff there exists a defined signal state for configuration C. □

The above proposition follows directly from the definition of the signal state of configuration. The next proposition establishes the conditions for configuration extensions which preserve the existence of signal states.

Proposition 5.2.2 Let C be a configuration of an STG-unfolding with a defined signal state ξ. Let also t'_i be an instance with a defined signal state of its local configuration $\xi_{\lceil t'_i \rceil}$. If there exists a configuration $C' = \lceil t'_i \rceil \cup C$ and $\forall t'_j \in \lceil t'_i \rceil, t'_k \in C, t'_k \| t'_j : |\Lambda(L'(t'_k))| \neq |\Lambda(L'(t'_j))|$, then C' has a defined signal state ξ'.

Proof: Since ξ is defined for C, all instances in $\lceil t'_i \rceil \cap C$ have signal states defined for their local configurations. Therefore consider instances in $\lceil t'_i \rceil \setminus C$. For these instances the proposition is proved by induction. For the base consider, without loss of generality, any instance $t'_j \in \lceil t'_i \rceil : \bullet t'_j \subseteq C\bullet$. Since $\xi_{\lceil t'_j \rceil}$ is defined, then the signal change $\Lambda(L'(t'_j))$ is consistent with the initial state. On the other hand, t'_j is the only instance added to C where all (existing) instances of $|\Lambda(L'(t'_j))|$ are in total order. The order of these instances in $C \cup \{t'_j\}$ may only become partial if there exists an instance $t'_k \in C, t'_k \| t'_j :$ $|\Lambda(L'(t'_k))| = |\Lambda(L'(t'_j))|$ which contradicts the conditions of the proposition. Hence the base case is done. The inductive step is proved using similar arguments for all instances in $\lceil t'_i \rceil \setminus C$.

□

Corollary 5.2.1 Let $t'_i \in T'$ be an instance of an STG-unfolding and C be a configuration such that $C = \lceil t'_i \rceil \setminus \{t'_i\}$. Let $Max(C)$ be the set of maximal instances in C. The signal state is defined for local configuration $\lceil t'_i \rceil$ if the signal states are defined for local configurations $\lceil t'_j \rceil : t'_j \in Max(C)$, there are no concurrent transitions $t'_k, t'_l \in C$ such that $|\Lambda(L'(t'_k))| = |\Lambda(L'(t'_l))|$ and $\Lambda(L'(t'_i))$ is consistent with ξ of the configuration C. □

Consider the formula

$$\xi[i] = \xi_0[i] \oplus |T^{a_i}|, \quad \forall i = 1, \ldots, |A|$$

for calculating the signal state of configuration C. It would be convenient to rewrite this formula in a vector form, i.e. $\xi = \xi_0 \oplus \zeta$. Each component $\zeta[i]$ corresponds to one and only one signal $a_i \in A$ and represents the number of occurrences of a given signal in C. Such a vector is called the *cumulative state of configuration* C. The notion of the cumulative state used here is similar to the one used in [49, 35]. The notion of the cumulative state of a configuration proves to be useful in the calculation of the cumulative state of configuration C from the cumulative states of local configurations comprising C.

Proposition 5.2.3 Let C be a configuration satisfying the conditions of existence of the signal state ξ; let also $Max(C) = \{t'_1, t'_2 \dots t'_k\}$ be the set of maximal instances of C. The cumulative state ζ of C is calculated as follows:

$$\forall i: \ 1 \le i \le n, \ \zeta[i] = \max_{j=1}^{k}(\zeta_{\lceil t'_j \rceil}[i]),$$

where $\zeta_{\lceil t'_j \rceil}$ is the cumulative state of the local configuration $\lceil t'_j \rceil$.

Proof: Take any $a \in A$ and T^a. Since all instances in T^a are totally ordered and C is finite, there is $t'_i \in Max(C)$ such that $T^a \subseteq \lceil t'_i \rceil$.

$\square$

The cumulative state of $\lceil \bot \rceil$ is a 0-vector. Using cumulative states it is possible to adapt the PN-unfolding algorithm to construct the STG-unfolding which takes into account signal states of configurations. Indeed, whenever a new transition t' is added to the unfolding, the signal state of its local configuration is calculated from the states of local configurations of instances in $Max(C)$ where $C = \lceil t' \rceil \setminus \{t'\}$, i.e. the local configuration of t' without t' itself. This is the minimal run of the underlying PN which enables this instance. Obviously, $\lceil t' \rceil$ must satisfy the conditions of the signal state existence (Definition 5.2.1). If $\lceil t' \rceil$ satisfies these conditions, then $\zeta_{\lceil t' \rceil}$ is found from the cumulative states of the local configurations of instances in $Max(C)$ and the signal state $\xi_{\lceil t' \rceil}$ is obtained.

Proposition 5.2.4 An FS $s = (M, v)$ exists in the FSG of an STG G if there exists a configuration $C : F_s(C) = M$ which has a defined signal state $\xi = v$.

Proof: If C exists and its signal state is defined, then for every signal all changes represented by instances in C are in an alternating total order; also, the first instance is consistent with the initial state of the signal. Hence, there exists at least one valid feasible sequence which includes all instances of C and therefore, by Proposition 5.1.1 there exists an FS $s = (F_s(C), \xi)$.

$\square$

Proposition 5.2.5 The FSG Φ has a signal deadlock iff there exists a configuration C in the STG-unfolding whose signal state ξ is undefined.

Proof: [*if*] Consider a configuration C whose signal state ξ is undefined. Let also $C' = C \setminus \{t'\}$ be a configuration with a defined signal state (it is always possible to find C, C' and t' satisfying these conditions). From Proposition 5.2.4 it follows that an FS $s' = (F_s(C'), \xi')$ exists. The signal state of a configuration C could only be undefined if: 1) t' has another concurrent instance in C' which represents a transition of the same signal or 2) $\Lambda(L'(t')) = *a_i$ is inconsistent with $\xi'[i]$ due to the presence of another instance $t'' : t'' \prec t'$ such that $\Lambda(L'(t'')) = \Lambda(L'(t'))$. In either case, by Definition 5.1.4, s' is a deadlock with respect to a_i.

[*only if*] Choose an FS $s = (M, v)$ for which there is minimal (in length) feasible firing sequence σ such that $M_0 \xrightarrow{\sigma} M$ and M is a signal deadlock with respect to some a_i.

Consider this FS. If this FS is s_0, then some instances $t' : L'(\bullet t') \subseteq M_0$ represents a signal transition of a_i which is inconsistent with $\xi_{\lceil\perp\rceil}[i]$, and hence $\xi_{\lceil t'\rceil}$ is undefined. Otherwise, in the PN-unfolding of the underlying PN there exists a configuration C' : $F_s(C') = M$. If at least one feasible sequence consisting of instances from C' is invalid (according to Definition 3.4.2), then the signal state of C' is undefined, in which case the proposition holds. Therefore, let all sequences be valid and ξ' be defined. There exists an instance $t' : \bullet t' \subseteq C'\bullet$ such that $\Lambda(L'(t')) = *a_i$. Obviously, there exists a configuration $C = C' \cup \{t'\}$. From Definition 5.1.4, $\xi'[i]$ is inconsistent with $\Lambda(L'(t'))$ and, therefore, C cannot have a defined signal state.

□

Theorem 5.2.1 Let Φ be an FSG free from signal deadlocks constructed for an STG G. Let also G' be the STG-unfolding constructed for G with signal states defined for all configurations. An FS $s = (M, v)$ exists in Φ iff there exists a configuration C such that $C = M$ and $\xi = v$ in G'.

Proof: [*if*] Follows from Proposition 5.2.4.

[*only if*] Suppose that FSG is signal deadlock free, but there exists a state $s = (M, v)$ for which there is no corresponding configuration C. Consider an FS $s' = (M', v')$ such that $M' \xrightarrow{*a_i} M$. There exists a configuration C' such that $F_s(C') = M'$ and $\xi' = v'$. Since s' is not a signal deadlock, then $\xi'[i]$ is consistent with $*a_i$ and therefore there exists a configuration $C = C' \cup \{t'\} : \Lambda(L'(t')) = *a_i$. Furthermore, $\Lambda(L'(t'))$ is consistent with $\xi'[i]$ and hence C has a defined signal state ξ. Therefore there exists an FS $s = (M, \xi)$.

□

If for a constructed STG-unfolding all configurations have defined signal states, then the FSG of the original STG has no signal deadlocks. At the same time, it follows from Theorem 5.1.1 that an STG is valid if its FSG covers the RG of the underlying PN (the FSG does not have signal deadlocks). Hence the analysis of STG validity can be done using the STG-unfolding. Similar to the PN-unfolding, the STG-unfolding can be infinite. Thus a cutoff condition is required. Obviously, this cutoff condition should be based on the cutoff condition of the PN-unfolding segment and take into account signal states of the STG-unfolding.

Definition 5.2.3 STG-unfolding cutoff
A newly generated instance t'_c is a cutoff transition in the STG-unfolding iff there exists another instance t' such that t'_c and t' satisfy the conditions of PN-unfolding segment cutoff transition (Definition 4.2.3 or Definition 4.2.7) and $\xi_{\lceil t'\rceil} = \xi_{\lceil t'_c\rceil}$. □

In addition to the cutoff conditions for the PN-unfolding segment, the cutoff condition for the STG-unfolding requires the signal states of local configurations to be equal.

Definition 5.2.4 The STG-*unfolding segment* $G'' = \langle N'', \Xi\rangle$ is the greatest backwards closed subnet of the STG-unfolding G' where for all $t' \in T''$ there exists an STG-unfolding cutoff (according to Definition 5.2.3) such that $t' \preceq t'_c$. □

```
proc Build STG-unfolding segment(G)
   Initialise $N'$ with instances of places in $M_0$
   Initialise QUEUE with t enabled at $M_0$
   while QUEUE not empty do
      Pull $t$ from QUEUE
      Add $t'$ and $t'\bullet$ to $N'$
      if $t'$ is a cutoff then do
         Mark $t'$ and $t'\bullet$ as cutoff points
      end do
      for each $t$ in $T$ do
         Find unused set of mutually concurrent instances of places in $\bullet t$
            which are not successors of a cut-off transition
         if such set exists then do
**             if signal state for $\lceil t' \rceil$ doesn't exist then do
**                Report signal deadlock in $\lceil t' \rceil$ and TERMINATE
**             end do
            Add $t$ to QUEUE according to the order $\Subset$
         end do
      end do
   end do
** for each maximal configuration $C_{max}$ do
**    if signal state for $C_{max}$ doesn't exist then do
**       Report signal deadlock in $\lceil t' \rceil$ and TERMINATE
**    end do
** end do
   return $N'$
end proc
```

Figure 5.5: Algorithm for obtaining STG-unfolding segment.

Property 5.2.1 The STG-unfolding segment G'' constructed for an STG G is finite if its underlying PN N is bounded. □

Lemma 5.2.1 Let G' be an STG-unfolding of an STG G. For any configuration C of G' with a defined signal state ξ there exists a configuration C' such that C' contains no cutoff transitions and $F_s(C) = F_s(C')$ and $\xi = \xi'$. □

The proof of the above statement is similar to those in [47] and [22] for bounded and safe PNs respectively.

Corollary 5.2.2 Let Φ be an FSG free from signal deadlocks constructed for an STG G. Let also G'' be the STG-unfolding segment constructed for G with signal states defined for all configurations. An FS $s = (M, v)$ exists in Φ iff there exists a configuration C such that $C = M$ and $\xi = v$ in G''. □

From the above corollary it follows that the STG-unfolding segment can be used for verification of STG validity and analysis. The STG-unfolding segment is constructed for an STG. If during construction of this segment a configuration is found whose signal state is undefined,

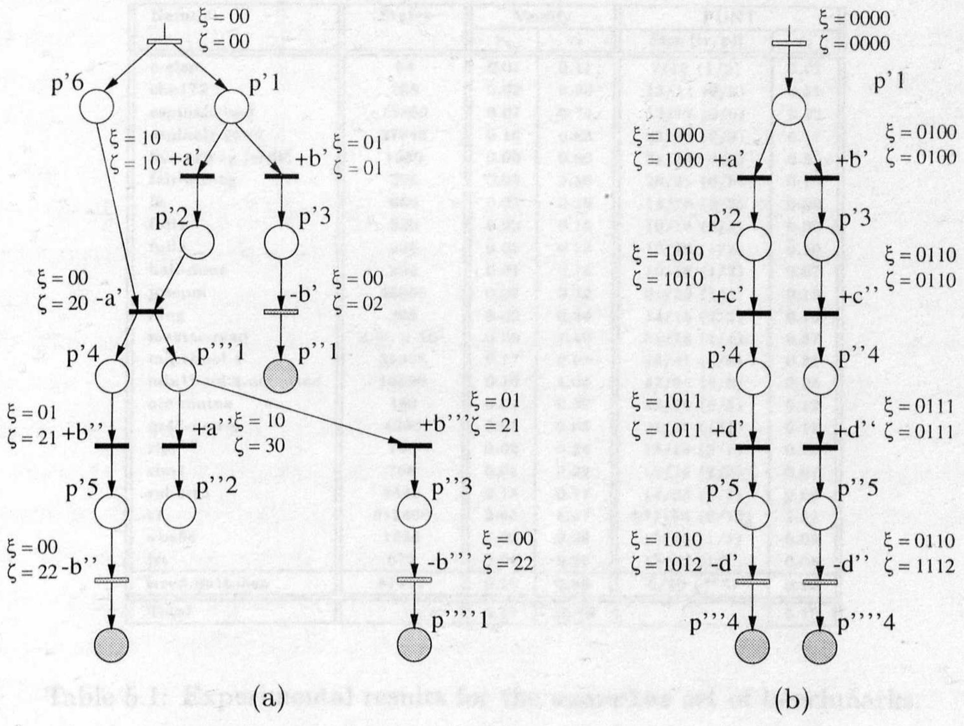


Figure 5.6: Examples of STG-unfolding segments.

then the STG is invalid. Otherwise, a finite STG-segment is constructed which represents the FSG.

It is impractical to check the existence of signal states of all configurations whenever a new instance is added. Let C_{max} be a maximal configuration in the STG-unfolding segment, i.e. there are no instances in the segment which can be added to C_{max}. Such a configuration is called *final configuration.* Proposition 5.2.1 implies that if the signal state existence conditions are satisfied for all C_{max}, then all configurations in G'' have their signal states defined. At the same time, signal states are calculated for all local configurations in G''. Thus, if such a local configuration is found, the segment construction is aborted and the signal deadlock is reported. The pseudo-code of the STG-segment construction is given in Figure 5.5. As it can be seen this algorithm is an extension of the PN-unfolding segment construction algorithm; the differences are in the lines marked with $**$.

Example. The STG-unfolding segment obtained from the STG in Figure 5.4(a) is shown in Figure 5.6(a). In order to avoid cluttering in figures the names of signals with apostrophe are used instead of the corresponding transition instance names; the number of apostrophes indicates the number of instances of a particular signal transition. The STG in invalid because final configuration $C_{max} = \{+a', -a', +b'', +b''', -b'', -b'''\}$ does not have a defined signal state. This configuration has concurrent transition instances of signal b.

Another STG-unfolding segment for the STG from Figure 5.1(a) is shown in Figure 5.6(b). As it can be observed all configurations have their binary states defined and therefore this STG is valid. Observe that in the PN-unfolding segment transition $+c''$ would be a cutoff transition. In the STG-unfolding segment this is not the case

Name	States	Versify		PUNT	
		τ_v	τ_t	Size [tr/pl]	τ_t
c-elem	64	0.01	0.11	7/12 (1/2)	0.07
chu172	768	0.02	0.26	13/14 (2/2)	0.11
espinalt-bad	15360	0.07	0.74	13/17 (0/0)	0.12
espinalt-good	27648	0.10	0.83	25/30 (2/3)	0.17
fair-arb-sg.jordic	1280	0.09	0.80	32/33 (9/9)	0.51
fair-arb-sg	208	0.03	0.36	20/21 (6/6)	0.14
fc	960	0.03	0.29	14/16 (2/2)	0.09
full1	320	0.02	0.15	10/14 (1/2)	0.06
full2	224	0.02	0.19	10/16 (1/2)	0.10
half-done	224	0.01	0.18	10/16 (1/2)	0.07
josepm	45056	0.07	0.72	21/29 (1/1)	0.12
lung	208	0.02	0.24	14/15 (2/2)	0.15
master-read	3.45×10^7	0.39	7.40	51/78 (1/1)	0.37
mix1bool.4	35328	0.17	0.98	28/41 (4/6)	0.21
mix1bool.4.nomutex	16896	0.16	1.04	32/54 (4/8)	0.24
orc-mutex	480	0.05	0.37	19/20 (5/5)	0.12
qr42-nousc	4096	0.07	0.65	21/31 (1/1)	0.11
rlm	768	0.02	0.24	13/14 (2/2)	0.13
rlm1	768	0.01	0.22	13/14 (2/2)	0.07
roberto	8448	0.13	0.77	14/23 (1/1)	0.06
t1	618496	2.65	8.97	67/104 (6/12)	2.87
vbe5c	1536	0.02	0.28	12/16 (1/1)	0.09
jst	672	0.02	0.26	12/22 (1/1)	0.08
irred.no1token	41472	0.19	0.93	6/10 (***)	0.07
Total		4.37	26.98		6.13

Table 5.1: Experimental results for the `examples` set of benchmarks.

because instances $+c'$ and $+c''$ have different signal states of their local configurations. ■

5.3 Low-level System Analysis

Analysis of asynchronous systems at the low level is divided into two categories: *analysis of circuit specifications* and *analysis of asynchronous circuits*. In the first case, the specification of the future circuit is verified for the correctness of its behaviour and implementability. The circuit specification is given in the form of an STG. In the second case, the existing circuit is verified for the correctness of its behaviour in a given environment. The circuit is usually designed by hand or using semi-automated techniques. In most cases the environment can be expressed in the form of an STG.

5.3.1 Analysis of STG specifications

Since an STG is a special case of LPNs, the algorithms suggested in the previous chapter for boundedness, safeness, liveness and persistency checks can all be applied in STG verification. Furthermore, all configurations have defined signal states in an STG-unfolding segment for a non-autoconcurrent[1] and valid STG. Thus, if a configuration is found which has an undefined signal state, it is reported along with the trace leading into it. Otherwise, the segment is constructed and it is used for the verification of necessary properties.

Experimental results for specification verification To illustrate the performance of the new method based on the STG-unfolding segment, the STG-unfolding segment construction algorithm was implemented as a part of the tool PUNT. This implementation was tested

[1] A non-autoconcurrent STG is an STG with a non-autoconcurrent underlying PN.

on a number of benchmarks available in the asynchronous community [40]. These benchmarks include a wide range of STGs, most of which have safe underlying PNs. The results of experiments are shown in Table 5.1 for the subset of benchmarks called `examples` and in Table 5.2 for the subset called `no-usc`. The experiments were carried out on a Sun SPARC20 workstation.

As previously, the new STG analysis method was compared to the method based on PN symbolic traversal and implemented in the tool `versify`. The tables present timing results (in seconds) for comparison. Columns "τ_t" for both tools show the total time taken for the verification of a benchmark. This time includes the time spent on the construction of the behaviour representation and verification of the properties. Verified properties include boundedness, safeness, signal (or action) persistency, liveness and deadlock freedom. Column "τ_v" for `versify` shows the fraction of time spent on verification of the properties on an already built BDD representation of the RG. Column "Size" for PUNT provides information about the STG-unfolding size in transition instances and place instances. It also shows in brackets the number of cutoff transitions and their immediate successors. For illustrative purposes, the size of the RG is also given in column "States".

As it can be observed from both tables, the method based on the STG-unfolding segment consistently outperforms the PN symbolic traversal approach. It demonstrates gain in speed for all benchmarks in these sets. Note the last line in Table 5.1. This STG, called `irred.no1token`, is an invalid STG. The STG-unfolding segment detects the signal deadlock in a fraction of a second and reports the trace leading into it. At the same time, `versify` constructs the whole RG and then detects problems in it. The PN-unfolding segment of the underlying PN had 25 transition instances and the verification of PN properties was completed in 0.13 seconds.

5.3.2 Analysis of asynchronous circuits

To analyse an asynchronous circuit it needs to be represented as an STG. The basic idea of its translation is similar to the modelling of two-phase circuitry: each gate is represented by an STG fragment. These fragments are composed together and this composition is then composed with the model of the circuit environment. However, unlike their two-phase counterparts, four-phase gates may have asymmetrical excitation functions for up and down transitions of the output signal, i.e. an up (and/or down) transition may be driven by a subset of input values. Furthermore, there may be several subsets driving the output of a gate. Therefore, transitions of the output signal for each gate have to be represented explicitly. Since all fragments are STGs, the result of their composition is an STG.

Modelling four phase circuits The STG representation of a gate, often referred to as a *Circuit* PN, can be obtained from its boolean function. Consider, for example an AND-gate with three inputs. Its boolean function can be written as:

$$y = S + y\,\overline{R}$$

where y is the output signal of the gate, S and R are *set* and *reset* functions respectively defined as:

$$S = i_1\, i_2\, i_3$$

Name	States	Versify		PUNT	
		τ_v	τ_t	Size [tr/pl]	τ_t
alex1.1.nousc	1216	0.02	0.30	13/27 (1/2)	0.12
alex1.2.nousc	288	0.01	0.17	13/21 (1/1)	0.12
alex1.3.nousc	288	0.01	0.24	11/25 (1/2)	0.11
alex1.nousc	1344	0.03	0.28	16/30 (1/3)	0.11
alex2.nousc	24	0.00	0.07	6/10 (1/2)	0.10
alloc-outbound.nousc	2176	0.02	0.35	18/19 (2/2)	0.13
future.nousc	9216	0.09	1.03	28/33 (1/2)	0.18
intel-edge.nousc	224	0.09	0.92	36/37 (8/8)	1.28
lin-edac93.nousc	320	0.01	0.27	12/15 (1/1)	0.08
master-read.1.nousc	7.31×10^7	0.25	2.46	74/113 (1/2)	0.53
master-read.nousc	8.99×10^6	0.19	1.53	40/64 (1/1)	0.29
mmu.nousc	20992	0.08	0.81	21/35 (1/1)	0.12
mp-forward-pkt.nousc	2560	0.06	0.53	15/28 (1/2)	0.11
nak-pa.nousc	28672	0.06	0.63	18/23 (1/1)	0.11
nousc.min-ser	256	0.02	0.24	12/20 (2/2)	0.06
nousc	256	0.02	0.22	12/20 (2/2)	0.10
nousc.tmp1	256	0.01	0.19	10/19 (1/2)	0.06
pe-rcv-ifc.chu	57344	0.48	2.71	49/67 (4/4)	0.68
pe-rcv-ifc.fc.nousc	11776	0.28	1.87	41/55 (4/4)	0.47
pe-rcv-ifc.nousc	12032	0.33	1.95	41/58 (4/6)	0.46
pe-send-ifc.nousc	30720	0.52	2.24	45/61 (6/12)	1.05
pulse.nousc	96	0.01	0.18	12/13 (1/1)	0.09
ram-read-sbuf.nousc	36864	0.07	0.77	21/29 (1/1)	0.14
rcv-setup.nousc	448	0.01	0.26	15/17 (3/3)	0.14
sbuf-ram-write.nousc	59392	0.09	0.93	24/36 (1/1)	0.18
sbuf-read-ctl.nousc	896	0.02	0.26	13/17 (1/1)	0.09
sbuf-read-ctl.nousc.old	960	0.02	0.31	15/17 (2/2)	0.10
sbuf-send-ctl.nousc	1280	0.03	0.47	18/23 (2/2)	0.10
sbuf-send-pkt2.nousc	1344	0.03	0.45	20/23 (4/4)	0.18
sbuf-send-pkt2.yun.nousc	1664	0.06	0.61	23/31 (4/4)	0.24
sendr-done.nousc	56	0.01	0.11	6/9 (1/2)	0.07
Total		2.94	23.36		7.60

Table 5.2: Experimental results for the `No-USC` set of benchmarks.

and

$$R = \overline{i_1\, i_2\, i_3} = \overline{i_1} + \overline{i_2} + \overline{i_3}$$

expressed in terms of the inputs of this gate.

The STG fragment corresponding to this AND gate is shown in Figure 5.7(a). Each signal has two corresponding places, one for each of its "high" and "low" states. Thus in this example there are six places representing the input signals of the gate and two places representing the output signal. The number of transitions switching the output is equal to the number of terms in the set and reset functions. The gate does not control its inputs. Therefore, transitions changing output value return tokens into their input places. Bi-directional arcs are used as a shorthand to represent arcs going to and from input places.

Sequential gates, such as Muller C-elements [53] (or generalised C-elements [2]), are also modelled by STG fragments. An example of the STG fragment for a generalised C-element with a logic function:

$$y = i_1\, i_3 + y(i_2 + i_3)$$

is given in Figure 5.7(b). This C-element has an asymmetric excitation function and has two transitions representing output signal changes. Each transition is connected to a different set of input places. Another example of a sequential gate is the Mutual Exclusion (ME) element. An ME-element resolves the conflict between two (or more) input requests. The illustration of the STG fragment for an ME-element is shown in Figure 5.7(c). This STG fragment has an additional place, representing the "memory" of the element. As it can be seen, this version of the ME-element resolves the conflict only between the "high" values of requests. It requires

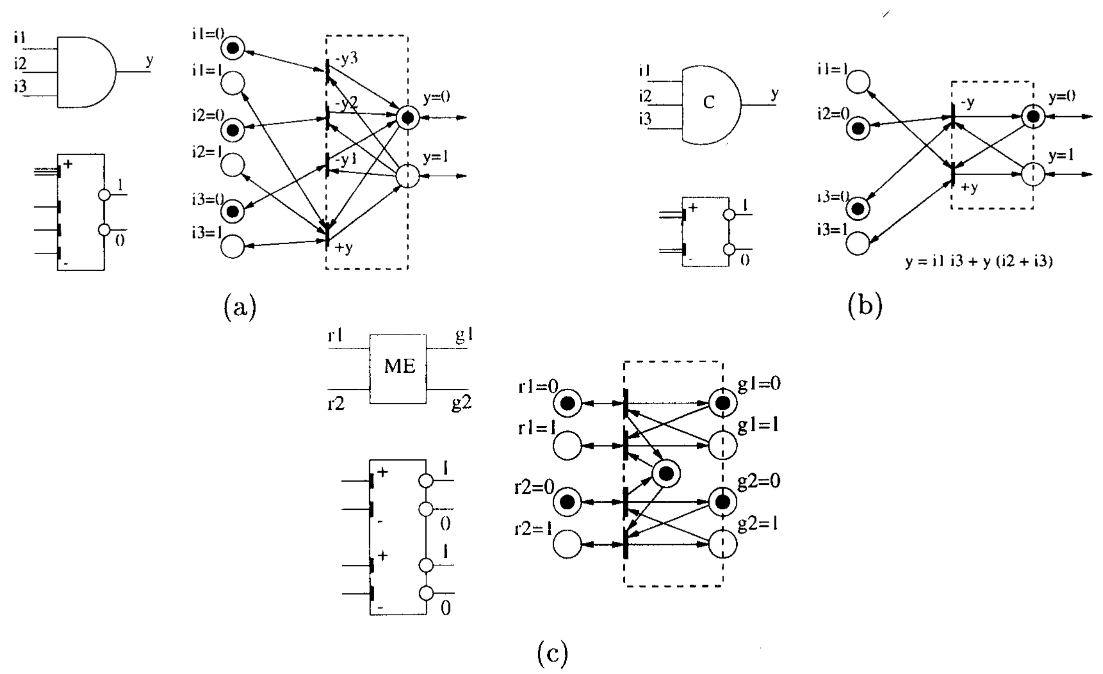


Figure 5.7: Examples of a 3 input AND gate (a), Muller C-element (b) and ME-element (c).

the request signal which has been given a "grant" to be lowered before the next conflict is resolved. ME-elements are a very useful tool for designing arbiters as they allow localisation of non-deterministic choice in one (specially designed) element [16].

These examples illustrate how the STG modelling is done for a gate with a given boolean logic function or behaviour. It is convenient to introduce a schematic-like depiction for gates which is also given for all three examples. This representation is similar to the representation of processors from [18]. Each gate shows its input-dependent transitions and places for output values. Up and down transitions are marked with "+" and "−" respectively; values of the outputs are labelled "0" and "1" for "high" and "low". The representation is chosen so that the only type of arcs that go between two gate boxes are bi-directional arcs. In order to avoid cluttering in the figures arrows on these arcs are not shown.

STG fragments are composed by connecting transitions which correspond to the output signal changes of one gate to the input places of those gate(s) to which this output is connected. If an output of the gate is forked to several gates, then output places are duplicated, one for each gate which uses this output as an input. Thus the true concurrency of operation of the gates in the circuit is preserved.

Experimental results for circuit verification To illustrate the analysis of four-phase circuits by means of the STG-unfolding segment a set of already designed circuits was used. Most of these circuits come from [35] where they were also used as examples for verification using CDs. Note that direct comparison between CD based and STG-unfolding segment based methods was impossible as Forcage, the only available CD-based verification tool, uses a different computer platform. However, it was possible to compare the verification results of

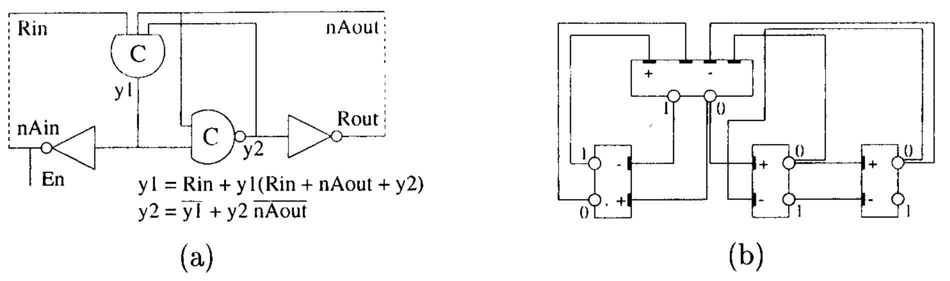


Figure 5.8: Four-phase micropipeline control circuit.

the STG models for these circuits using `versify`[2].

The results of the circuit verification are presented in Table 5.3. As previously, columns "τ_t" show the total time (in seconds) spent by each tool on the verification of the STG model of each circuit; column "τ_v" shows the fraction of time spent by **versify** on the verification of the properties on the BDD representation of the SG; column "Size" shows the size of the STG-unfolding segment. The size of the SG is given in the column "States" and the number of gates in each model is shown in column "Gates".

All circuit models were found safe, live and deadlock free. The top section of Table 5.3 presents the analysis results of circuits which were persistent, i.e. no signal transition can disable a transition of another signal. The lower part of the Table shows the verification results for circuits which had some non-persistent signals. For example, verification of the initial four-phase pipeline control circuit [17] from Figure 5.8(a) was performed using the STG model shown in Figure 5.8(b). For simplicity, delays in the environment are assumed to be represented by the existing inverters. The verification revealed that the outputs of both C-elements are non-persistent and reported the following trace: $+Rin, +y_1, -Rin, -y_2, +nAout, -y_1, +Rin$ which leads to a state in which both $+y_2$ and $+y_1$ are enabled but firing $y1$ will remove the excitation from y_2. In the corrected circuit, suggested in [98], the boolean function for the first C-element was $y_1 = Rin\ y_2 + y_1(Rin + nAout + y_2)$. The verification showed (line 1 in Table 5.3) that the STG model is persistent and the circuit is free from hazards.

Most of the circuits analysed here were deterministic circuits, i.e. circuits without internal arbitration. However, two benchmarks used in the experiments, **shapiro** and **LowLat2**, are implementations of a tree arbitration cell from [27] and an implementation of the Low Latency arbiter from [76]. Both circuits are non-deterministic and are designed so that all non-determinism of the circuit is concentrated in the ME-elements. The outputs of an ME-element are, of course, non-persistent and the alarm raised by their non-determinism should be disregarded. Both circuits were found persistent with respect to the rest of the signals.

It was also observed that the size of the STG-unfolding segment may grow drastically if the circuit is non-persistent. The non-persistency can, however, be detected within the first few transition instances. If the segment construction continues, e.g. when the circuit has arbitration in it, the STG-unfolding segment may suffer from an unfolding explosion. This is attributed to the presence of bi-directional arcs.

[2]`versify` is capable of the circuit verification using the boolean logic function for each gate. However the comparison with this approach was not considered here.

Name	Gates	States	Versify		PUNT	
			τ_v	τ_t	Size	τ_t
yak-kish	4	224	0.05	0.27	11/38 (1/2)	0.15
a4-tflo1	8	5120	0.36	1.84	20/115 (1/4)	0.42
joinMeng	6	4096	0.25	1.03	18/90 (1/3)	0.27
kish238	9	14336	0.45	1.89	21/107 (2/9)	0.39
kish327	4	160	0.11	0.76	9/55 (2/11)	0.20
shapiro	9	53248	0.75	2.46	26/130 (2/10)	0.38
LowLat2	10	407552	10.33	22.65	136/978 (27/212)	20.35
paul-day	4	224	0.04	0.22	12/37 (2/5)	0.15
c-elemt	6	2112	0.17	1.00	20/109 (6/38)	0.48
a4-tflo3	9	95744	2.10	4.87	130/597 (53/226)	11.19
kish132	9	40960	0.70	2.60	69/313 (14/55)	2.81

Table 5.3: Experimental results of four-phase circuit verification.

When a two-phase circuit is modelled using LPNs, the places represent the wires in the circuit. Since transitions in LPN represent events, the tokens propagation in the model denotes changes in the voltage levels on the wires. Once an event has occurred, the new conditions for this event can only be established when new tokens arrive into its input places. However, when a four-phase circuit is modelled, places represent current states of the signals. Thus transitions can only move tokens in the places representing the output signal of a gate. The enabling conditions are established by connecting the input places and transitions with bi-directional. The transition thus only "observes" the value of the gate's inputs and fires when the excitation conditions are reached.

The dependency between places and transitions connected by bi-directional arcs can be interpreted as the context in which a transition is allowed to fire. This was captured in the notion of *contextual nets* (see Chapter 7). If an ordinary net is used for modelling contextual dependencies, then the power of the STG-unfolding segment will make a distinction between an input place marked before and after a transition fired. This, in turn, may cause an exponential explosion in the size of the unfolding which would simply represent all possible firing of contextually dependent transitions. To avoid this, a *Contextual Net unfolding* is introduced later in Chapter7.

5.4 Conclusions

This chapter introduced the semantical model of an FSG was introduced. This formalism, unlike previous ones, can capture the behaviour of an arbitrary STG. States of the FSG consist of two components: a marking and a binary code. Unlike the conventional RG, the FSG distinguishes states with different markings but equal binary codes. Hence it is possible to build a correct behavioural representation for an acyclic STG and, thus, exclude false alarms raised by the conventional SG analysis methods.

The second main result of this chapter is the STG-unfolding segment. It is based on the PN-unfolding segment but also takes into consideration signal interpretations of the transitions. The STG-unfolding segment has been applied to the verification of STG specifications as well as verification of existing circuits. It was demonstrated that the new STG analysis method based on the STG-unfolding segment is more favourable for a large number of benchmarks.

It was also observed that the circuit analysis may suffer from the explosion of the STG-unfolding due to the high degree of concurrency in the STG and the presence of bi-directional arcs between places and transitions. Using contextual nets and the contextual net unfolding avoids this explosion. This method is described later in this thesis.

Chapter 6

Synthesis from STG-unfolding

The aim of this chapter is to introduce a new technique for the synthesis of SI circuits from their STG specifications. This new method uses partial order presented as an STG-unfolding segment to derive the boolean logic implementation. It is based on the idea of a *slice*, which localises the behaviour of a particular signal instance in a structural fragment of the segment. Two approaches are suggested: the exact cover calculation and the cover approximation. Within the approximation approach two strategies for approximate cover derivation are considered. The method is applied to the synthesis in three main implementation architectures. Experimental results show the power of the approximation approach in comparison with the existing methods.

First, this chapter draws up a classification of common implementation architectures for SI circuits and summarises the correctness criteria for each architecture. Then it introduces new notions defined on the STG-unfolding for the synthesis procedure. After that, a new method is described for each architecture. Finally, two techniques for improving the implementations and speeding up the synthesis are suggested and experimental results results are given that compare the new method with the existing "state-of-the-art" ones.

6.1 Motivation

There exists a variety of approaches to the synthesis of SI circuits from their STG specifications. These approaches can be divided into groups according to the types of gate libraries used for the implementation of output signals. For example, [3, 2] uses a Muller C-element for each implementable signal and a network of gates to drive it. Work presented in [35] uses an RS-latch in similar conditions. Early methods, e.g. [13, 48], assume that each signal is implemented as a single complex gate. Later techniques, e.g. [83, 38], attempt to decompose the complex gates preserving the speed-independence of the circuit.

Another taxonomy in the synthesis of SI circuits is established by the various methods used to obtain the boolean logic functions. Two primary approaches exist to date: State Graph (SG) based and structural methods (eliciting information for the synthesis from the structure of the STG). The first approach constructs an SG as a model that represents the behavioural properties of the "would-be-circuit". It then proceeds with extracting the subsets of states required for the implementation. This method is used in such tools as SIS [82] and Assassin [102]. A recently developed tool `petrify` (and approach) [15] uses BDDs to represent

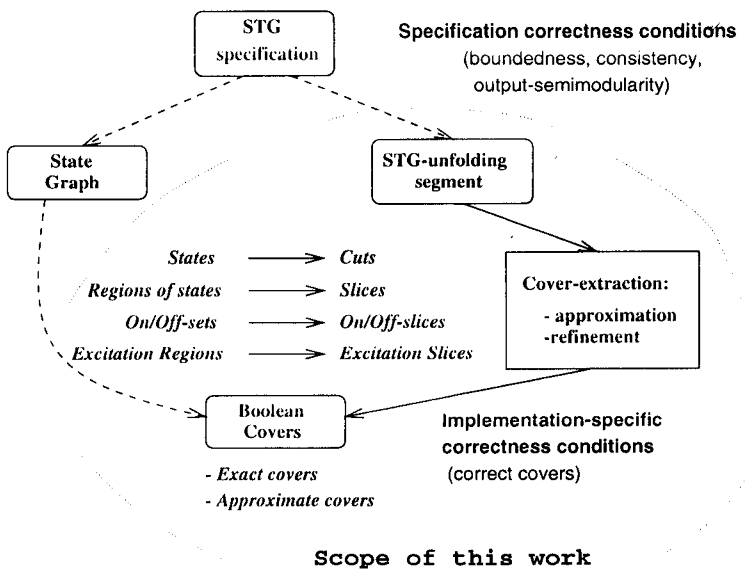

STG specification
Specification correctness conditions
(boundedness, consistency, output-semimodularity)
State Graph
STG-unfolding segment
States → Cuts
Regions of states → Slices
On/Off-sets → On/Off-slices
Excitation Regions → Excitation Slices
Cover-extraction:
- approximation
-refinement
Boolean Covers
- Exact covers
- Approximate covers
Implementation-specific correctness conditions
(correct covers)
Scope of this work

Figure 6.1: Overview of synthesis issues discussed in the work.

the state space. Due to implicit representation of the SG this tool is efficient in synthesising moderate-sized examples. Since all these methods work with the full SG, they suffer from the state explosion problem, i.e. the number of reachable states grows exponentially with the size of the specification.

The structural method of [62] uses State Machine (SM) decompositions of STGs to obtain concurrency relations between signal transitions. Using these relations, this method finds an approximate implementation avoiding the exploration of the state space. Thus it demonstrates impressive results although it is restricted to free-choice specifications. The method described in [103] also uses structural information of the STGs (lock relations between signals) to synthesise circuits.

Partial order techniques have also been applied in the synthesis process. CDs were introduced in [35] for the synthesis of SI circuits from choice-free (no choice at all) specifications. The absence of choice constructs significantly restricted their use. Nevertheless, this work was a significant step in the development of this approach – it was first to establish the relationship between the sets of connected states (e.g. excitation regions in the SG) and elements of the event-based description (event instances in the CD-unfolding).

A more recent work of [50] uses PN-unfoldings to derive logic functions. This work, however, is based on restoring the state space from the partial order and is, therefore, also prone to the state explosion.

It was shown in previous chapters that the partial order technique, based on the PN-unfolding, can obtain an implicit reachability graph representation, in the form of a finite segment of the unfolding. It can often construct an STG-unfolding segment for those examples where the construction of the reachability graph fails. This STG-unfolding segment represents all states of an asynchronous system. In addition, as the segment is constructed it is verified for correctness. Thus, after the verification stage is completed, the implementation can be derived from the constructed STG-unfolding segment.

This chapter proposes a novel approach for the synthesis of SI circuits from the STG-

unfolding segment of their specifications. Initially it suggests an *exact* method, which produces implementations comparable with those of the SG approach. Although the exact approach benefits from STG-unfolding segment properties, this method may still suffer from state explosion. To overcome this problem, an *approximation* method is suggested, which is based on temporal relations found in the segment. However, unlike [62], this approach works with the partial order representation of a fragment of system's execution. Therefore, it uses local dependency information available for each instance of every signal transition. Only instances of signal transitions which are concurrent to a particular instance are considered. This gives a more accurate initial approximation and a more precise refinement of cover functions for signal implementation. The proposed approach is applied to the synthesis in three major implementation architectures. The scope of synthesis issues presented in this work is summarised in Figure 6.1.

6.2 Implementation of SI Circuits

6.2.1 Basic Synthesis Concepts

Conventionally, to obtain an implementation for an STG G a corresponding SG (or FSG[1]) S is derived by constructing the reachability graph of the underlying PN and then assigning a binary vector v_j to each state s_j. The binary vectors are assigned *consistently* (see Chapter 5). States of the SG generated by a consistent STG have therefore two components: a marking and a binary vector. At the circuit level, however, the states are represented only by their binary vectors which are in this case values of the signals in the circuit. Thus it may be possible that two states with equal binary vectors will be indistinguishable at the circuit level. This situation is often referred to as *coding conflict.* The *Complete State Coding* (CSC) condition introduced in [13] requires any two states with equal binary vectors to have the same set of excited output signals. If for some signal a_i this requirement is not satisfied, then it is impossible to extract the boolean function for its implementation. It was shown in [13] that STGs satisfying CSC are implementable as SI circuits.

Once a consistent state assignment has been performed, truth tables are obtained for each output signal and an implementation is produced. The process of obtaining a truth table depends on the implementation architecture chosen (for this particular signal).

Correctness criteria for the synthesis of SI circuits can be divided into *general specification correctness criteria* and *architecture specific correctness criteria.* General correctness criteria are verified by examining the behavioural properties of an STG. These criteria constitute the requirements for an STG to be implementable "in principle". The general criteria are:

- STG **validity**, which requires an STG to be consistent with its implementation as a circuit;
- **Boundedness**, which guarantees that the behaviour specified by an STG can be implemented using a finite number of components;

[1]Traditionally an SG is constructed, but the FSG is required to represent correctly the behaviour of acyclic STGs.

- **Semi-modularity** (also called "output signal persistency") which ensures that the output signals cannot be disabled by some input signal change and thus cause a hazard;
- CSC **satisfiability**, which ensures that the STG specification has sufficient state coding power to allow the synthesis procedure to produce logic functions for the output signals.

It was shown earlier that the first three criteria can be checked on the STG-unfolding segment G' constructed for an STG G. Verification of the latter criterion requires obtaining all reachable states explicitly. Instead of that, the approach given here attempts to construct an implementation. If it fails, then the STG does not satisfy CSC. It is, nevertheless, possible to correct an STG specification by inserting additional (internal) signals to resolve the coding conflict. This, however, is considered to be outside of the scope of this work. Several automated techniques for resolving CSC conflicts at the SG level have been published, e.g. [92, 14].

An implementation is obtained by building a *cover* function. A boolean function with a variable corresponding to each signal is said to be covering a state s_j if it evaluates to TRUE when the variables have the values equal to the elements of binary vector v_j assigned to s_j. A function C covering a set of states is called a *cover function* (or simply *cover*) for this set of states $\{s_i\}$. Each term of the cover is traditionally called a *cube*, denoted as B, as it may cover several states. Each variable in a cube is called a *literal*. Note, however, that a cube may have literals for all signals, in which case it is called a *minterm*.

The concepts of a minterm and its corresponding binary vector are inherently very close. A minterm can be easily obtained from a binary vector by substituting all 1's with the literals of corresponding signals, and all 0's with their complementary literals. If a binary vector corresponds to a set of states, i.e. some of the variables are "$-$", then the corresponding signal's literal is omitted from the minterm, which produces a cube. For example, a binary vector $v = \{101\}$ corresponds to a cube $B = a\bar{b}c$ and vice versa, on a set of signals $\{a, b, c\}$. Similarly, a binary vector $v = \{-10\}$, which represents two vectors $\{010\}$ and $\{110\}$, corresponds to a cube $B = \bar{a}b\bar{c} + ab\bar{c} = b\bar{c}$ and vise versa. Therefore, among the "synthesis community", set-theoretical operations on sets of states are often interchanged freely with boolean logic operations applied to minterms and cubes, e.g. $\{10-\} \cup \{-10\} \Leftrightarrow a\bar{b} + b\bar{c}$. Henceforth, this work will assume that these operations are interchangeable for the purposes of cover derivation. In addition, transformations between binary vectors and cubes do not require any additional operators.

The cover is not required to cover *only* states in $\{s_i\}$. If a cover is obtained by performing standard boolean transformations on the set of terms which are extracted from the binary vectors of the states, then, of course, it covers only the states in $\{s_i\}$. Such a cover is called the **exact cover**. However, if a cover was obtained differently (e.g. using an oracle or any other method), it may include some other states. In this case such a cover is called the **approximated cover**. Approximated covers need to be checked for correctness. The notion of correctness, applied at the implementation stage, is concerned with the requirement of hazard-freedom with respect to the gates of the synthesized circuit. It is different from the general correctness conditions (discussed above) related to the specification. As a matter of fact, this notion is essentially dependent on the implementation architecture because the sets of states s_i, for which the boolean covers are extracted, are different for different architectures.

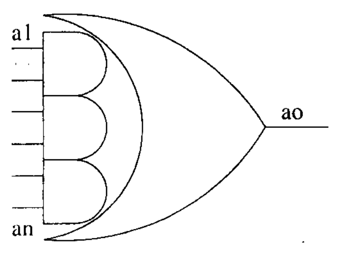


Figure 6.2: Atomic complex gate architecture.

Thus, the *architecture-specific correctness criteria* establish relations between sets of states s_i and cover functions.

There are three basic implementation architectures which have been rigorously researched in a number of works:

- Atomic Complex Gate per Signal (ACGPS) implementation;
- Atomic Complex Gate per Excitation Function (ACGPEF) implementation;
- Atomic Complex Gate per Excitation Region (ACGPER) implementation.

The first architecture can be considered as the basic one. The other two aim at reducing the sizes of customised complex gates. Each implementation architecture is discussed along with the corresponding correctness criteria in more detail in the following subsections. In this chapter, however, the correctness criteria are chosen to be relatively simple. They do, however, guarantee existence of an implementation for any STG satisfying CSC. For instance, in the last two architectures, as a target for boolean covering, only those states are considered where the implementable signals are excited. Several recent papers, e.g. [62, 38], examine the possibility of expanding the set of covered states with states at which those signals are stable. This, however, can be viewed as an optimisation aimed at reducing the size of customised complex gates.

6.2.2 ACGPS Implementation

This is the initial architecture for SI circuits studied in [13, 48, 73]. The circuit is implemented as *a network of atomic gates*, each one implementing one output signal. The boolean function for each gate can be represented as a Sum-Of-Products (SOP). The general view of such a gate is shown in Figure 6.2. Each atomic gate contains a combinational part, and a possibly sequential part implemented as an internal feedback. The delay between its "ANDing" and "ORing" parts, and the internal feedback is assumed to be negligible. The diagrammatic gate representation is used to denote the implemented logic function, but the actual implementation is resolved at the transistor level.

Recall (from Section 2.2.3) that a signal is said to be stable in a particular state if it is equal to the value computed by the corresponding logic function under the values given by the vector; otherwise it is called excited. Two (mutually complementary) subsets of the reachable states are distinguished in the SG for every signal a_i:

- the *on-set*, denoted as $On(a_i)$, which includes all states at which the output signal a_i has the implied value of TRUE, i.e. $s \in On(a_i)$ if a_i in s is stable at 1 or is excited to

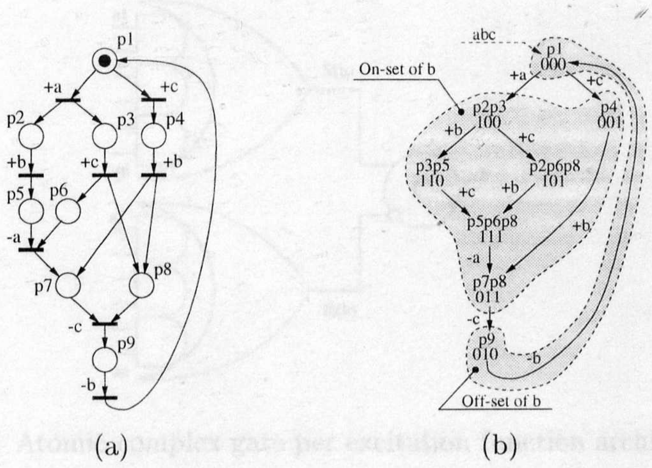


Figure 6.3: Example of an STG (a) and its corresponding SG (b).

switch to 1; and

- the *off-set*, denoted as $Off(a_i)$, which includes all states at which a_i has the implied value of FALSE.

The implementation is derived by building a cover for the on-set[2]. Each state $s = (M, v)$ can be represented by a minterm B which has $|A|$ variables each corresponding to one and only one signal a_i. The minterm becomes TRUE only when the values of the variables are equal to those in the binary vector assigned to the state. The cover $\mathcal{C}_{On}(a_i)$ for the implementation is obtained as:

$$\mathcal{C}_{On}(a_i) = \sum_j B_j,$$

where B_j are the minterms obtained from the binary vectors v_j for all states $s_j \in On(a_i)$.

In many cases the union of the on- and off- sets is smaller than the set of all possible combinations of the corresponding boolean variables. The difference in this case is called the *Don't care* set (DC-set). None of the states in the DC-set is reachable in the system and thus the terms corresponding to the states in the DC-set can be used for minimisation. This is done using standard minimisation tools, such as Espresso [82].

Example. The synthesis procedure for this architecture is illustrated in Figure 6.3(b) for an STG shown in the Figure 6.3(a). Suppose that an implementation of the signal b is required. The on-set for b is found from the SG as: $On(b) = \{(p_2, p_3), (p_3, p_5), (p_2, p_6, p_8), (p_5, p_6, p_8), (p_7, p_8), (p_4)\}$. The cover function $\mathcal{C}(b)$ is obtained as: $\mathcal{C}(b) = a\bar{b}\bar{c} + ab\bar{c} + a\bar{b}c + abc + \bar{a}bc + \bar{a}\bar{b}c = a + c$.

The DC-set in this example is empty, so no further minimisation can be attempted. ■

Obtaining exact covers usually means that all states in the on- or off- set must be known. Recent research (including this thesis) is aimed at obtaining the covers without exhaustive exploration. Thus the problem is posed as follows: given two covers (obtained by some

[2]Here and later, for simplicity, it is assumed that the on-set is constructed. Usually the simplest of on-set or off-set is chosen for implementation.

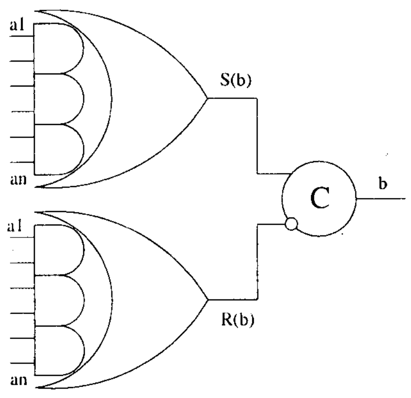


Figure 6.4: Atomic complex gate per excitation function architecture.

algorithm, e.g. approximation) to determine if they represent a correct implementation. For this implementation architecture the covers (exact or approximate) should satisfy the following condition:

Definition 6.2.1 Two covers $C^*_{On}(a_i)$ and $C^*_{Off}(a_i)$ are said to satisfy the ACGPS *correctness condition* iff:

$$\begin{aligned} On(a_i) &\subseteq C^*_{On}(a_i) \\ Off(a_i) &\subseteq C^*_{Off}(a_i) \\ C^*_{On}(a_i) \cap C^*_{Off}(a_i) &\subseteq \text{DC-set} \end{aligned}$$

□

It should be emphasised that the correctness condition above *does not require* the covers to be exact. Exact covers are a particular case and satisfy this condition iff the STG has CSC since in such an STG $C_{On}(a_i)$ and $C_{Off}(a_i)$ cover $On(a_i)$ and $Off(a_i)$ respectively and their intersection is empty.

6.2.3 ACGPEF Implementation

The ACGPEF architecture was studied extensively in a number of papers, e.g. [3, 37, 2]. It assumes that a separate memory element is used to produce an output signal. The Set and Reset *excitation* functions for this memory element are implemented as atomic complex gates. Depending on which memory element is used, the implementations are divided into:

- *Standard C-element* implementation, which uses Muller C-element as the memory element (e.g. shown in Figure 6.4(a)), and
- *Standard RS-latch* implementation, where an RS-latch is used.

For synthesis purposes the following sets of states, called regions, are identified on the SG for each signal transition.

Definition 6.2.2 The *Generalised Excitation Region* (GER) of a signal transition $*a_i$, denoted as $GER(*a_i)$, is the set of all states of the SG S in which $*a_i$ is excited, i.e.:

$$GER(*a_i) = \{s_j | \exists e = (s_j, s_k) : \Psi(e) = *a_i\}.$$

□

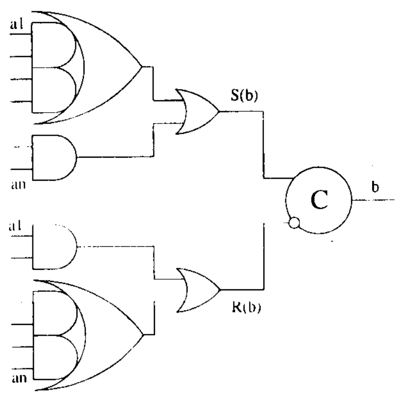


Figure 6.5: Atomic complex gate per excitation region architecture.

A GER for some signal transition $*a_i$ represents all states in which this signal transition is excited. Since there are two possible transitions for one signal $+a_i$ and $-a_i$, there are two corresponding GERs $GER(+a_i)$ and $GER(-a_i)$ found for each signal.

An implementation is obtained by finding covers C_S and C_R for the set and reset functions of a memory element from the minterms corresponding to the states in $GER(+a_i)$ and $GER(-a_i)$, respectively.

The set function is obtained from the terms corresponding to the states in $GER(+a_i)$. Similarly, the minimal reset function is obtained from the terms for states in $GER(-a_i)$. It is possible to show the existence of an implementation in the ACGPEF architecture for any STG satisfying the CSC condition.

Example. The covers C_S and C_R for signal b in Figure 6.3(b) are obtained as follows: $C_S(b) = a\bar{b}\bar{c} + a\bar{b}c + \bar{a}\bar{b}c = a\bar{b} + \bar{b}c$ and $C_R(b) = \bar{a}b\bar{c}$ which is an implementation of the signal in the ACGPEF architecture. ■

The cover correctness condition in this architecture is as follows:

Definition 6.2.3 A set (reset) cover $C_S(a_i)$ ($C_R(a_i)$) is said to be correct if the only reachable states covered by $C_S(a_i)$ ($C_R(a_i)$) belong to $GER(+a_i)$ ($GER(-a_i)$) and it covers all states in this GER. □

The conditions imposed on the set and reset covers correspond to the "intuitive" understanding of the correctness of a circuit, i.e. absence of hazards. Indeed, according to the conditions of the definition, the output signals of atomic complex gates implementing the set and reset functions may only be switched where it is excited in the specification; and it is not allowed to switch at any other reachable state outside their corresponding GERs. Note that these conditions do not restrict the correct cover to the one which covers GER exactly. The latter is a special case of the correct cover. An approximated cover may also include states from the DC-set. An implementation of an STG with CSC by means of exact GER covers always exists as they satisfy these correctness conditions.

6.2.4 ACGPER Implementation

Signals in this architecture are created using *networks of atomic complex gates to implement set and reset functions of the memory element*. As a result, smaller complex gates are used which are then connected to an OR-gate whose output is in turn fed into the memory element. The basic structure of this architecture is shown in Figure 6.5. Similar to the previous architecture, the memory element can be a Muller C-element or an RS-latch.

As it can be noted, the GER for a particular signal instance $*a_i$ can have several connected sub-regions. This is captured in the definition of the excitation region.

Definition 6.2.4 The *Excitation Region* (ER) of a signal transition $*a_i$, denoted as $ER(*a_i)$, is the maximal connected set of states of the SG S in which $*a_i$ is excited. □

There may exist several connected ERs for one signal transition $*a_i$. From definitions it follows that

$$\mathrm{GER}(*a_i) = \bigcup_j ER_j(*a_i)$$

.

The correctness condition for this architecture is as follows:

Definition 6.2.5 A set of covers $C_S^1(a_i), \ldots, C_S^n(a_i)$ $(C_R^1(a_i), \ldots, C_R^m(a_i))$ for the set (reset) function of signal a_i is said to be correct if each $ER_j(+a_i)$ $(ER_j(-a_i))$ is covered by its corresponding $C_S^j(a_i)$ $(C_R^j(a_i))$ and the only reachable states covered by $C_S^j(a_i)$ $(C_R^j(a_i))$ belong to $ER_j(+a_i)$ $(ER_j(-a_i))$. □

Example. Signal b in the STG of Figure 6.3 can also be implemented in the ACGPER architecture. In this case the set network contains two gates corresponding to two covers obtained for both ERs of $b+$ whereas the reset cover contains only one gate: $C_S^1(b) = a\bar{b}\bar{c} + a\bar{b}c = a\bar{b}$, $C_S^2(b) = \bar{a}\bar{b}c$ and $C_R(b) = \bar{a}b\bar{c}$. ■

The correctness condition requires that the cover for each ER is interpreted as a separate gate. However, this condition also requires that only one atomic complex gate is switching at a time, thus avoiding dynamic delay hazards[2]. Similar to the previous architecture, the covers need not cover their ERs exactly. Exact covers are a special case which always satisfy this condition. In an extreme case, when the set cover consists of only one cover the OR-gate, merging the outputs of gates implementing ER covers, becomes redundant.

Any circuit implementable in ACGPER architecture is also implementable in ACGPEF architecture. In this case all covers for ERs are implemented as one atomic complex gate. The ACGPER architecture can be viewed as an attempt to improve over ACGPEF architecture. If the set and reset covers covering GERs in ACGPEF architecture were obtained from several covers for ERs, then the ACGPER architecture implementation can be found by checking which of the ER covers have non-empty intersection and implementing them as one gate per a connected ER.

6.3 Basic Definitions

The purpose of this section is to define new concepts for the synthesis based on the STG-unfolding segment, which will be used later. Two main notions are introduced: a *cut* and a *slice*; they allow identification of sets of states at the unfolding level.

6.3.1 STG-unfolding Cuts

It was shown earlier that all states of an FSG (and hence an SG) corresponding to an STG are represented in the STG-unfolding segment. A state of the SG is captured by a *cut* as follows:

Definition 6.3.1 A maximal set of places c of the unfolding N' satisfying the following: $\forall p'_i \in \mathsf{c}, \forall p'_j \neq p'_i \in \mathsf{c} : p'_i \| p'_j$ is called a *cut* of the unfolding N'. □

From definitions it follows that for each cut c_i there exists a unique configuration C_i such that $C_i\bullet = \mathsf{c}_i$. Thus in an STG-unfolding segment constructed for a valid STG it is possible to associate a binary vector ξ_{C_i} with every cut c_i. Hence, each cut corresponds to a reachable state of the SG built for an STG.

It is convenient to define relations between cuts.

Definition 6.3.2 A cut c_1 is said to be in relation $\preceq$ with another cut c_2, if $\forall p'_i \in \mathsf{c}_1 \ \exists p'_j \in \mathsf{c}_2 : p'_i \preceq p'_j$. □

Relation $\mathsf{c}_1 \preceq \mathsf{c}_2$ can be viewed as cut precedence, i.e. cut c_1 is preceding cut c_2. It has been shown [21] that if cut $\mathsf{c}_1 = C_1\bullet$ precedes another cut $\mathsf{c}_2 = C_2\bullet$, then $C_1 \subseteq C_2$. Two cuts c_1 and c_2 are said to be in conflict if there exists at least one pair of places $p'_1 \in \mathsf{c}_1$ and $p'_2 \in \mathsf{c}_2$ such that $p'_1 \# p'_2$.

Two special cuts are uniquely identified by any transition instance of the STG-unfolding. These cuts represent states at which the corresponding signal transition first time becomes enabled and stable.

Definition 6.3.3 A cut $\mathsf{c}_e^{min}(t'_k)$ is called a *minimal excitation cut* of t'_k iff $\bullet t'_k \subseteq \mathsf{c}_e^{min}(t'_k)$ and $\forall t'_j, t'_j \| t'_k : (t'_j\bullet) \cap \mathsf{c}_e^{min}(t'_k) = \emptyset$. □

Definition 6.3.4 A cut $\mathsf{c}_s^{min}(t'_k)$ is called a *minimal stable cut* of t'_k iff $t'_k\bullet \subseteq \mathsf{c}_s^{min}(t'_k)$ and $\forall t'_j, t'_j \| t'_k : (t'_j\bullet) \cap \mathsf{c}_s^{min}(t'_k) = \emptyset$. □

As it can be seen from the definition, the minimal stable cut is the post-set of the local configuration of t'_k. The initial transition $\perp$ has only maximal stable cut $\mathsf{c}_s^{min}(\perp) = \perp\bullet$ defined for it. For any other instance there exist exactly one minimal excitation cut and one minimal stable cuts. Note, however, that if the cuts of unfolding correspond to states of an STG, the minimal stable cut $\mathsf{c}_s^{min}(t')$ for an instance t' may in fact correspond to a state which excites another transition of the signal a_i labelling t'. The only interesting case for this work is when a_i is an output signal. In this case, such an STG does not have CSC since there are two states in the SG of this STG assigned with the same binary state vector. Hence, this property can be used in verification of CSC using the STG-unfolding segment. There also exists a correspondence between the cuts found in the STG-unfolding segment and the concepts of the Change Diagram theory [35]. For example, the minimal excitation cut corresponds to the minimal entry point of an ER of the signal labelling t'_k.

Two other special types of cuts can be found for each transition instance. These represent states from which the system cannot make any further progress unless some condition is violated.

Definition 6.3.5 A cut $\mathsf{c}_e^{max}(t'_k)$ is called a *maximal excitation cut* of t'_k iff $\bullet t'_k \subseteq \mathsf{c}_e^{max}(t'_k)$ and there is no other cut c_j such that $\bullet t'_k \subseteq \mathsf{c}_j$ and $\mathsf{c}_e^{max}(t'_k) \prec \mathsf{c}_j$. □

A maximal excitation cut defines a cut which represents a marking from which no further advancement of the PN can be made unless t'_k is fired or it is disabled by firing of some other transition. There may be several maximal excitation cuts for a particular instance t', the number of maximal stable cuts for each instance is equal to the number of configurations C such that $\bullet t'_k \subseteq C$ and t'_k is the only instance that can be added to C.

To define the next type of cuts the notion of the *next transition* for an instance t'_i is defined in the STG-unfolding.

Definition 6.3.6 The set of instances $next(t'_i)$ is called the *set of next transitions for* t'_i iff $\forall t'_k \in next(t'_i)$ the following is true: $|\Lambda(L'(t'_k))| = |\Lambda(L'(t'_i))|$, $t'_i \prec t'_k$, there is no t'_l such that $(|\Lambda(L'(t'_l))| = |\Lambda(L'(t'_i))|) \wedge (t'_i \prec t'_l \prec t'_k)$ and $next(t'_i)$ is maximal. □

In other words, $next(t'_i)$ is the subsequent change of the signal labelling t'_i. From the definition it follows that in a STG-unfolding segment of a correct STG these two instances will be labelled with signal transitions of opposite sign. $first(a_i)$ is the set of all instances labelled with $*a_i$ which appear first in any sequence represented in the STG-unfolding segment, i.e. $\forall t'_j \in first(a_i) : \forall t'_l, |\Lambda(L'(t'_l))| = a_i : t'_l \not\prec t'_j$. Obviously, all instances in $first(a_i)$ found in an STG-unfolding segment built for a valid STG are labelled with the same signal transition $*a_i$.

Definition 6.3.7 A cut $c_s^{max}(t'_i)$ is called a *maximal stable cut* of t'_i if the following hold:

- For every cut c_j with $c_s^{min}(t'_i) \preceq c_j \preceq c_s^{max}(t'_i)$ there is no $t'_k \in next(t'_i)$ such that $\bullet t'_k \subseteq c_j$; and
- Let C be the configuration generating $c_s^{max}(t'_i)$. Then for every configuration $C \cup \{t'_m\}$ there is an instance $t'_k \in next(t'_i)$ such that $\bullet t'_k \subseteq (C \cup \{t'_m\}\bullet)$.

□

A maximal stable cut for an instance t'_i represents a maximal state which is reachable from t'_i but does not enable an instance of $next(t'_i)$. If there is no next instance in a particular configuration, then the maximal cut is bounded by the cutoff transitions or maximal transitions of this configuration.

A maximal stable cut is defined for the initial transition $\perp$ with respect to a particular signal a_i. Such a cut $c_s^{max}(\perp |a_i)$ is defined as a cut satisfying:

- For every cut c_j with $c_s^{min}(\perp) \preceq c_j \preceq c_s^{max}(\perp |a_i)$ there is no $t'_k \in first(a_i)$ such that $\bullet t'_k \subseteq c_j$; and
- Let C be the configuration generating $c_s^{max}(\perp |a_i)$. Then for every configuration $C \cup \{t'_m\}$ there is an instance $t'_k \in first(t'_i)$ such that $\bullet t'_k \subseteq C \cup \{t'_m\}\bullet$.

A maximal stable cut for the initial transition can be viewed a maximal stable cut which would be found if this transition were labelled with a signal transition of a_i.

There may exist several maximal excitation and maximal stable cuts for each instance t'_i; however, their number is always finite in the STG-unfolding segment for a valid STG. The sets of all maximal excitation and maximal stable cuts for a particular instance t'_i are denoted as $C_e^{max}(t'_i)$ and $C_s^{max}(t'_i)$, respectively. Furthermore, an instance t'_i may have no maximal stable

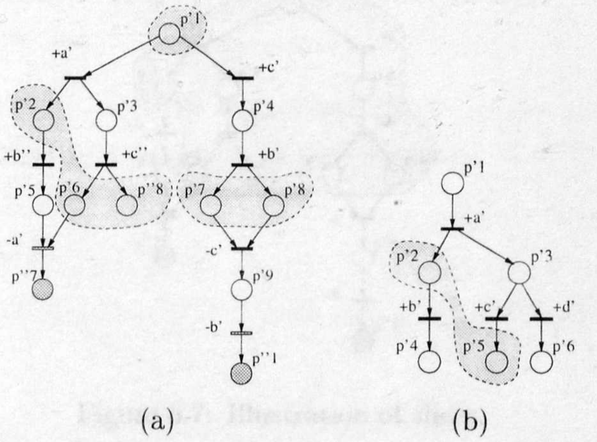


Figure 6.6: Illustration of cuts.

cuts defined for it. In particular, there is no maximal stable cut for t'_i if $c_s^{min}(t'_i)$ enables a transition in $next(t'_j)$. Similar to the case considered above for the minimal stable cut, an STG for which the STG-unfolding was constructed does not have CSC, and hence this property can also be employed in verification of CSC satisfiability for a particular STG.

Example. All four types of cuts are illustrated in Figure 6.6.

Figure 6.6(a) shows the STG-unfolding segment for the STG from Figure 6.3. The minimal excitation cut of instance $+a'$ is the same as the minimal excitation cut of $+c'$ and is equal to (p'_1). The maximal excitation cut of $+b''$ is (p'_2, p'_6, p''_8). The maximal stable cut of $+b'$ is (p'_7, p'_8). Another example of the STG-unfolding segment, shown in Figure 6.6(b), illustrates several maximal excitation cuts for one instance. Instance $+b'$ has two maximal excitation cuts: (p'_2, p'_5) and (p'_2, p'_6) (only one is shown). ■

Lastly, a partial cut is introduced as follows.

Definition 6.3.8 A set of places c^* is called a *partial cut* of the unfolding N' iff $\forall p'_i \in c^*, \forall p'_j \neq p'_i \in c^* : p'_i \| p'_j$. □

By definition, any subset of place instances from any cut c is a partial cut c^*.

6.3.2 STG-unfolding Slices

Each cut of the STG-unfolding captures one particular state of the SG. However, in order to synthesise an SI circuit, it is necessary to define a notion that captures a set of states. Such notion is a *slice* of the STG-unfolding.

Definition 6.3.9 The set of cuts of the STG-unfolding segment is called a *slice* and is defined as a tuple $\mathcal{S} = \langle c^{min}, C^{max} \rangle$ where:

- c^{min} is a cut called the *minimal cut (min-cut) of slice* $\mathcal{S}$, and
- C^{max} is a set of cuts, called the *set of maximal cuts (max-cuts) of slice* $\mathcal{S}$ such that $\forall c_i \in C^{max} : (c^{min} \preceq c_i) \wedge (\nexists c_j \in C^{max}, c_i \neq c_j : c_j \prec c_i)$.

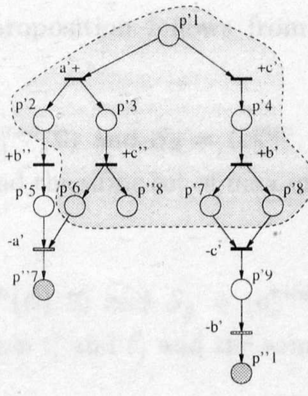


Figure 6.7: Illustration of slices.

Every cut $\mathsf{c}_k : (\mathsf{c}^{min} \preceq \mathsf{c}_k) \wedge (\exists \mathsf{c}_j \in \mathsf{C}^{max} : \mathsf{c}_k \preceq \mathsf{c}_j)$ is said to be *encapsulated by* $\mathcal{S}$ and is denoted as $\mathsf{c}_k \in \mathcal{S}$. □

Thus a slice is a set of cuts which is captured between the min-cut and the set of max-cuts of the slice. Since each cut represents some state of the SG, the slice represents a set of reachable states of the SG. From the above definition one can conclude the following properties.

Property 6.3.1 For every cut c encapsulated by a slice $\mathcal{S} = \langle \mathsf{c}^{min}, \mathsf{C}^{max} \rangle$ there exists a max-cut $\mathsf{c}' \in \mathsf{C}^{max}$ such that $\mathsf{c}^{min} \preceq \mathsf{c} \preceq \mathsf{c}'$. □

Property 6.3.2 Let $\mathcal{S} = \langle \mathsf{c}^{min}, \mathsf{C}^{max} \rangle$ be a slice. If a cut c is such that there exist two cuts $\mathsf{c}' \in \mathcal{S}$ and $\mathsf{c}'' \in \mathcal{S}$ such that $\mathsf{c}' \preceq \mathsf{c} \preceq \mathsf{c}''$, then $\mathsf{c} \in \mathcal{S}$. □

These two properties are important when using slices for identification of states. Since every cut corresponds to some state in the SG, a slice represents a connected set of states. Moreover, following Property 6.3.1 any state which is represented by c encapsulated by $\mathcal{S}$ is reachable from the state represented by c^{min}. Furthermore, this connected set of states does not have "holes", i.e. it does not contain a state (or a subset of states) which is surrounded by states represented as cuts in a slice.

Example. The notion of a slice is illustrated in the example in Figure 6.7. A slice $\mathcal{S}$ can be defined using a min-cut $\mathsf{c} = (p_1')$ and a set of max-cuts $\mathsf{C} = \{(p_2', p_6', p_8''), (p_7', p_8')\}$. This slice captures cuts (p_2', p_3') and (p_4'). However, no slice can be defined in this example if the min-cut is $\mathsf{c} = (p_4')$ and the set of max-cuts includes a cut which is non-sequential to c, e.g. (p_2', p_3'). ■

For any two slices the following relation is defined:

Definition 6.3.10 Two slices $\mathcal{S}_1$ and $\mathcal{S}_2$ are said to be in $\sqsubseteq$ relation ($\mathcal{S}_1 \sqsubseteq \mathcal{S}_2$) iff $\forall c \in \mathcal{S}_1 \Rightarrow c \in \mathcal{S}_2$. □

In other words, if $\mathcal{S}_1 \sqsubseteq \mathcal{S}_2$, then all cuts encapsulated in $\mathcal{S}_1$ are encapsulated in $\mathcal{S}_2$.

Property 6.3.3 For any two slices $\mathcal{S}_1$ and $\mathcal{S}_2$ the following is true: $\mathcal{S}_1 \sqsubseteq \mathcal{S}_2$ iff

- $\mathsf{c}_2^{min} \preceq \mathsf{c}_1^{min}$,
- $\forall \mathsf{c}_j \in \mathsf{C}_1^{max}, \exists \mathsf{c}_i \in \mathsf{C}_2^{max} : \mathsf{c}_j \preceq \mathsf{c}_i$. □

The proof of the following proposition follows from the definitions of a slice and the sequence relation between cuts.

Proposition 6.3.1 Let $\mathcal{S}_1 = \langle \mathsf{c}_1^{min}, \mathsf{C}\rangle$ and $\mathcal{S}_2 = \langle \mathsf{c}_2^{min}, \mathsf{C}\rangle$ be two slices defined using min-cuts c_1^{min} and c_2^{min} respectively and the same set of max-cuts C. If $\mathsf{c}_1^{min} \preceq \mathsf{c}_2^{min}$, then $\mathcal{S}_1 \sqsubseteq \mathcal{S}_2$. □

Corollary 6.3.1 Let $\mathcal{S}_i = \langle \mathsf{c}_e^{min}(t_i'), \mathsf{C}\rangle$ and $\mathcal{S}_j = \langle \mathsf{c}_e^{min}(t_j'), \mathsf{C}\rangle$ be two slices defined using *minimal excitation cuts* of instances t_i' and t_j' and the same set of max-cuts C. If $t_i' \preceq t_j'$, then $\mathcal{S}_j \sqsubseteq \mathcal{S}_i$. □

Corollary 6.3.2 Let $\mathcal{S}_i = \langle \mathsf{c}_s^{min}(t_i'), \mathsf{C}\rangle$ and $\mathcal{S}_j = \langle \mathsf{c}_s^{min}(t_j'), \mathsf{C}\rangle$ be two slices defined using *minimal stable cuts* of instances t_i' and t_j' and the same set of max-cuts C. If $t_i' \preceq t_j'$, then $\mathcal{S}_j \sqsubseteq \mathcal{S}_i$. □

Corollary 6.3.3 For two slices $\mathcal{S}_i' = \langle \mathsf{c}_e^{min}(t_i'), \mathsf{C}\rangle$ and $\mathcal{S}_i'' = \langle \mathsf{c}_s^{min}(t_i'), \mathsf{C}\rangle$ defined using minimal excitation and stable cuts of instance t_i' and the same set of max-cuts C the following is true: $\mathcal{S}_i'' \sqsubset \mathcal{S}_i'$. □

Finally, a slice can be associated with a fragment of the STG-unfolding and therefore it is convenient to refer to the elements of the unfolding (instances of places and transitions) which are within this fragment as those *belonging to the slice.*

6.3.3 Concluding Remarks

This section provided basic definitions for the synthesis method described in the following sections. It also established useful properties of cuts and slices. Although these are tailored for the synthesis from STGs, similar definitions will hold for ordinary PNs. In this case a slice of the PN-unfolding segment represents a set of reachable markings.

6.4 Exact Cover Implementation

This section suggests a new method for extracting exact cover implementations using the STG-unfolding segment for all three main architectures.

6.4.1 ACGPS Implementation

To obtain an implementation of an STG specified behaviour in complex gates, the covers for the on-set and off-set of each output signal are required. In this section it will be shown how the *exact covers* are obtained from the STG-unfolding segment.

Here only the on-set cover calculation is considered. To obtain the off-set cover the instances of $-a_i$ should be used. The exact cover for the on-set $On(a_i)$ is obtained by interpreting the binary vectors of the states in $On(a_i)$ as minterms of a Boolean function.

A slice of an STG-segment represents states reachable in the SG of an STG. Therefore, the problem of finding the on-set $On(a_i)$ from the STG-segment can be restated as finding a partitioning of the segment with a set of slices representing $On(a_i)$. Each slice in this case represents a subset of $On(a_i)$. To define each slice its min-cut and the set of max-cuts need to be identified in the segment.

Consider finding the min-cuts for the $On(a_i)$ partitioning. The local configuration of any instance t'_j in the STG-segment represents a set of minimal runs of the original STG leading to the marking reached by the firing of t'_j as soon as it becomes enabled. Any cut of the STG-segment which represents a state at which t_j is enabled is sequential to its minimal excitation cut $c_e(t'_j)$. Recall that every instance t' uniquely defines its minimal excitation cut and the set of maximal stable cuts in the STG-unfolding segment. Therefore, t' uniquely defines a slice $\mathcal{S}(t') = \langle c_e^{min}(t'), C_s^{max}(t')\rangle$ from its minimal excitation cut $c_e^{min}(t')$ up to, but not including, the cuts enabling instances in $next(t')$ (if there are any). Similarly, the initial transition uniquely identifies a slice $\mathcal{S}(\perp|a_i) = \langle c_s^{min}(\perp), C_s^{max}(\perp|a_i)\rangle$ which encapsulates all cuts starting from the very first one up to those enabling transitions in $first(a_i)$.

Definition 6.4.1 A set of slices $\mathbb{S}_{On}$ ($\mathbb{S}_{Off}$) in an STG-unfolding segment G' is called the *On-set (Off-set) partitioning of G' w.r.t. a signal a_i* iff for each instance $+a'_i$ ($-a'_i$) its corresponding slice $\mathcal{S}^j_{On}(+a'_i)$ ($\mathcal{S}^j_{Off}(-a'_i)$) is defined with $c_e^{min}(+a'_i)$ ($c_e^{min}(-a'_i)$) as its min-cut and $C_s^{max}(+a'_i)$ ($C_s^{max}(-a'_i)$) as a set of its max-cuts.

If signal a_i is at "1" ("0") in the initial state of the STG, then the set of slices $\mathbb{S}_{On}$ ($\mathbb{S}_{Off}$) also includes a slice defined with $c_s^{min}(\perp)$ as its min-cut and $C_s^{max}(\perp)$ as the set of its max-cuts. □

Each slice in $\mathbb{S}_{On}$ ($\mathbb{S}_{Off}$) is called an *on-slice* (*off-slice*). The set of instances $*a_i$ is called the set of *entry transitions* for the On-set (Off-set) partitioning of the STG-unfolding segment.

The exact cover for the states which have corresponding cuts in an on-slice $\mathcal{S}_{On}(+a'_i)$ is found from:

$$\mathcal{C}_{On}(+a'_i) = \sum_j \xi_{C_j},\ C_j\bullet = c_j \in \mathcal{S}_{On}(+a'_i)$$

from which the on-set cover is obtained using all instances of $+a_i$ as:

$$\mathcal{C}_{On} = \sum_k \mathcal{C}_{On}(+a^k_i)$$

where $\mathcal{C}_{On}(+a^k_i)$ is found for each slice in $\mathbb{S}_{On}$ as above. In other words, the on-set cover is found as the union of all covers found for the slices of on-set partitioning. Each slice cover is obtained by taking the union of all minterms corresponding to the signal states of configurations C_j whose post-set is a cut c_j encapsulated by $\mathcal{S}_{On}(+a^k_i)$.

Lemma 6.4.1 A state s of the SG S constructed for a valid and CSC-compliant STG G belongs to an on-set (off-set) of a_i iff there exists a corresponding cut c in the STG-unfolding segment G' which is encapsulated into at least one on-slice (off-slice) for $+a_i$.

Proof: (for on-set, the proof for off-set is similar) [*if*] Follows from the definition of the on-slice. Since none of the cuts in an on-slice can enable a $-a_i$ transition and the only cuts that have the value of $\xi[a_i] = 0$ for the corresponding configuration C are those enabling $+a_i$, then every cut covered in some on-slice belongs to the on-set of a_i.

[*only if*] Since the STG is valid, there exists a corresponding cut c for any reachable state $s = (M, v)$ and no instance in c is a successor of a cutoff transition (Lemma 5.2.1).

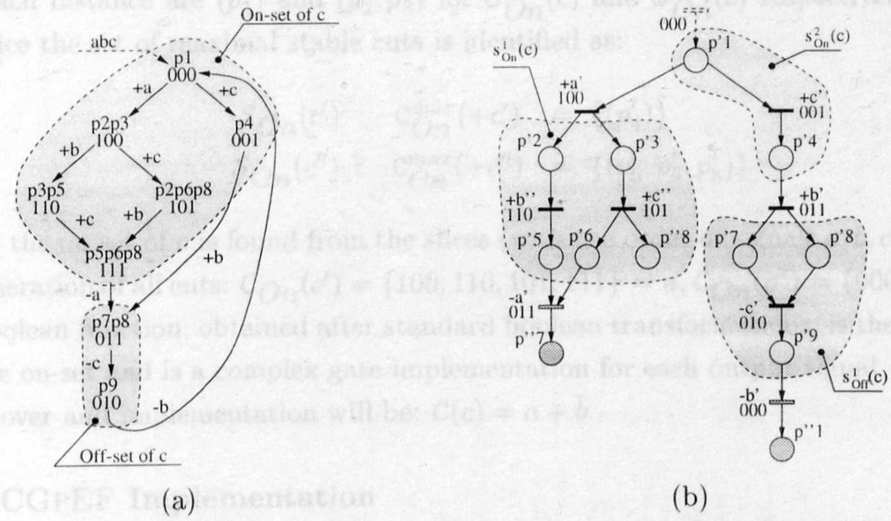


Figure 6.8: An SG (a) and STG-segment (b) for the STG in Figure 6.3.

Suppose that such a cut exists but it is not encapsulated by any of the slices in the On-set partitioning with respect to a_i. Two cases are possible: (1) cut c represents a state exciting $+a_i$, i.e. $v[i] = 0$; and (2) cut c represents a state in which a_i is stable, i.e. $v[i] = 1$. Consider each of these cases separately.

(1) The STG-unfolding segment includes all cutoff point transitions. Therefore, there must exist an instance $+a_i'$ such that $\bullet(+a_i') \subseteq \mathsf{c}$ and, furthermore, $\mathsf{c}_e^{min}(+a_i') \preceq \mathsf{c}$. Instance $+a_i'$ will be used to find the On-set partitioning of G' w.r.t. a_i. Hence, c will be encapsulated into $\mathcal{S}_{On}(+a_i') = \langle \mathsf{c}_e^{min}(+a_i'), \mathsf{C}_s^{max}(+a_i') \rangle$ and this is a contradiction.

(2) If c is not encapsulated by any of the slices in $\mathbb{S}_{On}(a_i)$, then either

i) there exists an instance $-a_i'$ (which changed the value of v to 1) such that $\mathsf{c}_e^{min}(-a_i') \preceq \mathsf{c}$ and no instance $+a_i'$ which was used to define an on-slice encapsulating c such that $-a_i' \prec +a_i'$; or

ii) $\mathsf{c}_s^{min}(\bot) \preceq \mathsf{c}$ and no instance $+a_i'$ which was used to define an on-slice encapsulating c such that $\bot \prec +a_i'$.

However, in both cases since c does not belong to the On-set partitioning, then it must be encapsulated by the Off-set partitioning of G'. Since $v[i] = 1$, then the state v belongs to those states in the off-set which excite $-a_i$. However, a_i must be stable at v. Otherwise, there exist two states which have different markings but equal binary vectors with different sets of excited output signal transitions enabled in them. This contradicts the condition that the STG has CSC.

□

Example. Consider the calculation of the on-set for the signal c of the STG in Figure 6.3 from its STG-segment shown in Figure 6.8(b). For illustrative purposes, the signal states of all local configurations are shown and the SG of this STG is also given. The set of entry transitions is found as $T_e' = \{+c', +c''\}$. The minimal excitation cuts of the slices for each instance are (p'_1) and (p'_2, p'_3) for $\mathcal{S}^1_{On}(c)$ and $\mathcal{S}^2_{On}(c)$ respectively. For each on-slice the set of maximal stable cuts is identified as:

$$\begin{aligned} \mathcal{S}_{On}(c') &: \mathsf{c}^{max}_{On}(+c') &= \{(p'_4)\} \\ \mathcal{S}_{On}(c'') &: \mathsf{c}^{max}_{On}(+c'') &= \{(p'_5, p'_6, p'_8)\} \end{aligned}$$

Thus the on-set of c is found from the slices (with the order of signals a, b, c) by explicit enumeration of all cuts: $\mathcal{C}_{On}(c') = \{100, 110, 101, 111\} = a$, $\mathcal{C}_{On}(c'') = \{000, 001\} = \bar{a}\bar{b}$. A Boolean function, obtained after standard boolean transformations, is the exact cover of the on-set and is a complex gate implementation for each output signal. For signal c the cover and implementation will be: $\mathcal{C}(c) = a + \bar{b}$. ■

6.4.2 ACGpEF Implementation

This type of architecture assumes that the boolean function for each output signal a_i is given in the form: $a_i = S + a_i\overline{R}$, where S and R are the set and reset functions for a_i. Each function is a polyterm cover which is implemented as an atomic complex gate.

Consider finding the set function for a_i; the reset function is obtained in a similar way using, the instances of $-a_i$.

The set function cover $\mathcal{C}_S(a_i)$ must cover all states in the *generalised excitation region* of a particular signal a_i. The problem of obtaining the set function from the STG-segment is restated as finding a set of slices in the STG-segment which represent the states from ERs of the signal. Several transitions of the underlying PN may be labelled with one signal transition $+a_i$. Furthermore, several transitions of the STG may correspond to one signal transition in the SG.

Consider obtaining the exact set cover for a particular signal transition $+a_i$. Each instance $+a'_i$ uniquely defines a minimal excitation cut $\mathsf{c}^{min}_e(+'a_i)$ and a set of maximal excitation cuts $\mathsf{c}^{max}_e(+'a_i)$. Therefore, it uniquely defines a slice $\mathcal{S}_e(+a'_i)$ with $\mathsf{c}^{min}_e(+'a_i)$ as its min-cut and a set of maximal excitation cuts $\mathsf{c}^{max}_e(+'a_i)$ as its set of max-cuts. Any cut enabling $+a'_i$ is encapsulated by $\mathcal{S}_e(+a'_i)$. The set (reset) partitioning is, therefore, defined as follows.

Definition 6.4.2 A set of slices $\mathbb{S}_S(a_i)$ ($\mathbb{S}_R(a_i)$) in an STG-unfolding segment G' is called the *Set (Reset) partitioning of G' with respect to signal a_i* iff for each instance $+a'_i$ ($-a'_i$) its corresponding slice $\mathcal{S}_e(+a'_i)$ ($\mathcal{S}_e(-a'_i)$) is defined with $\mathsf{c}^{min}_e(+a'_i)$ ($\mathsf{c}^{min}_e(-a'_i)$) as its min-cut and $\mathsf{C}^{max}_e(+a'_i)$ ($\mathsf{C}^{max}_e(-a'_i)$) as a set of its max-cuts. □

Each slice $\mathcal{S}_e(*a'_i)$ is called an *excitation slice of* $*a'_i$. Similar to the previous architecture, the cover for the set function is obtained from the binary vectors corresponding to the cuts encapsulated by the slices:

$$\mathcal{C}_e(+a'_i) = \sum_j \xi_{C_j},\ C_j\bullet = \mathsf{c}_j : \mathsf{c}_j \in \mathcal{S}_e(+a'_i).$$

from which the set cover is found as:

$$\mathcal{C}_S(a_i) = \sum_k \mathcal{C}_e(+a^k_i)$$

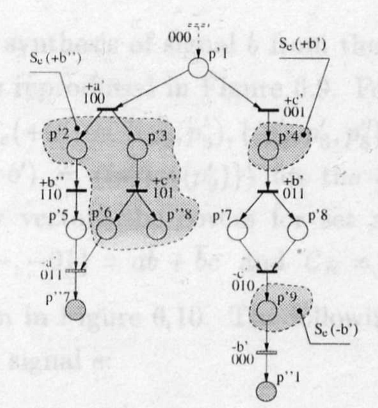


Figure 6.9: Illustration of the Set cover calculation on STG-unfolding.

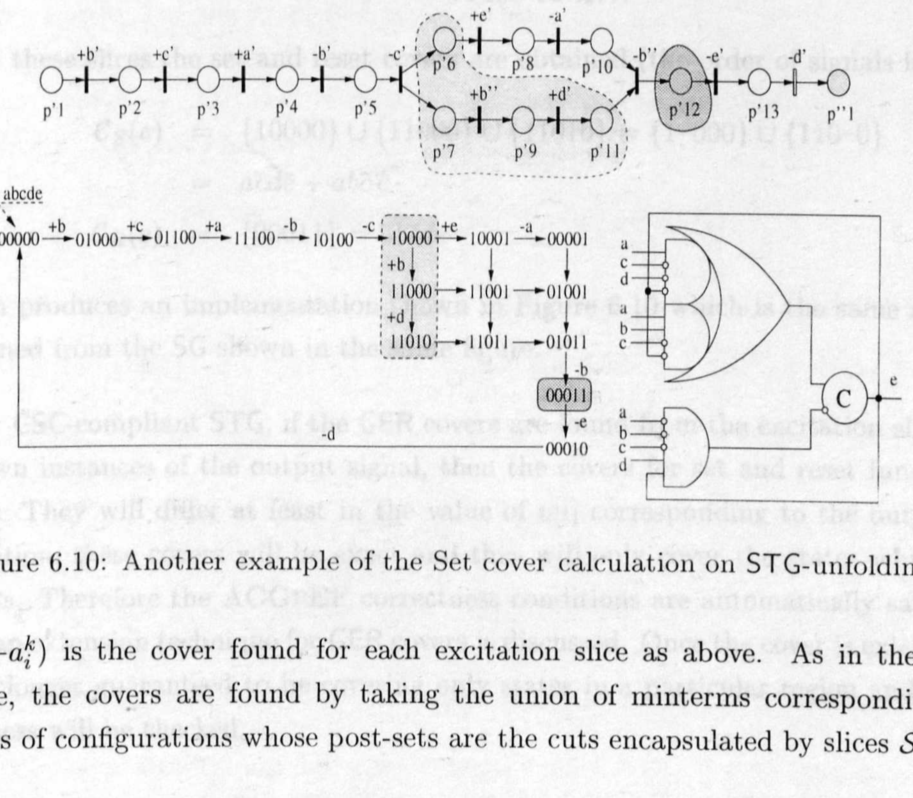


Figure 6.10: Another example of the Set cover calculation on STG-unfolding.

where $\mathcal{C}_e(+a_i^k)$ is the cover found for each excitation slice as above. As in the previous architecture, the covers are found by taking the union of minterms corresponding to the signal states of configurations whose post-sets are the cuts encapsulated by slices $\mathcal{S}_e(+a_i^k)$.

Lemma 6.4.2 A state s of the SG S constructed for a valid and CSC-compliant STG G belongs to $GER(+a_i)$ $(GER(-a_i))$ iff there exists a corresponding cut c in the STG-unfolding segment G' which is encapsulated into at least one excitation slice $+a_i$ $(-a_i)$.

Proof: (only $+a_i$ is considered, for $-a_i$ the proof is similar) [*if*] Any cut encapsulated by some excitation slice $\mathcal{S}_e(+a_i')$ enables $+a_i'$. Thus, by the definition of GER, it belongs to $GER(+a_i)$.

[*only if*] The STG-unfolding segment represents any reachable state s as a cut c and no instance in c is a successor of a cutoff transition (Lemma 5.2.1). For any transition $+a_i$ enabled in s there exists an instance $+a_i'$ and, furthermore, $c_e^{min}(+a_i') \preceq$ c. Instance $+a_i'$ will be used in the Set partitioning of G' and c will be encapsulated by the excitation slice $\mathcal{S}_e(+a_i')$.

□

Example. Consider again the synthesis of signal b from the example in Figure 6.3(b). The STG-unfolding segment is reproduced in Figure 6.9. For each instance of b the excitation slices are found as $\mathcal{S}_e(+b'') = \langle (p_2', p_3'), \{(p_2', p_6', p_8'')\}\rangle$ and $\mathcal{S}_e(+b') = \langle (p_4'), \{(p_4')\}\rangle$ for GER of $+b$ and $\mathcal{S}_e(-b') = \langle (p_9'), \{(p_9')\}\rangle$ for the opposite signal transition. After extracting the binary vectors the covers for set and reset functions are: $\mathcal{C}_S = \{100, 101\} \cup \{001\} = \{10-, -01\} = a\bar{b} + \bar{b}c$ and $\mathcal{C}_R = \{010\} = \bar{a}b\bar{c}$.

Another example is shown in Figure 6.10. The following slices are found on the STG-unfolding segment for the signal e:

$$\begin{aligned} \mathcal{S}_e(+e') &= \langle (p_6', p_7'), \{(p_6', p_{11}')\}\rangle \\ \mathcal{S}_e(-e') &= \langle (p_{12}'), \{(p_{12}')\}\rangle \end{aligned}$$

From these slices the set and reset covers are obtained (the order of signals is $abcde$):

$$\begin{aligned} \mathcal{C}_S(e) &= \{10000\} \cup \{11000\} \cup \{11010\} = \{1{-}000\} \cup \{110{-}0\} \\ &= a\bar{c}\bar{d}\bar{e} + ab\bar{c}\,\bar{e} \\ \mathcal{C}_R(e) &= \{00011\} = \bar{a}\bar{b}\bar{c}de \end{aligned}$$

which produces an implementation shown in Figure 6.10 which is the same as the one obtained from the SG shown in the same figure. ■

For any CSC-compliant STG, if the GER covers are found from the excitation slices of the up and down instances of the output signal, then the covers for set and reset functions will be disjoint. They will differ at least in the value of $v[i]$ corresponding to the output signal a_i. In addition, these covers will be exact and thus will only cover the states which belong to the GERs. Therefore the ACGPEF correctness conditions are automatically satisfied. In Chapter 8 an extension technique for GER covers is discussed. Once the cover is extended, this cover is no longer guaranteed to be covering only states in a particular region and therefore its correctness will be checked.

6.4.3 ACGPER Implementation

In this implementation an attempt is made to implement each ER as one gate whose outputs are then fed into an OR-gate to form a set or reset function of the memory element.

Suppose that the excitation slices were calculated for each instance of $+a_i$ in an STG-unfolding segment built for a CSC-conformant STG. Suppose also that for each slice $\mathcal{S}_e(t_j')$ a cover $\mathcal{C}_e(t_j')$ for all states represented by the cuts encapsulated in $\mathcal{S}_e(t_j')$ was obtained. As was discussed earlier, the set cover satisfies, by construction, the ACGPEF correctness conditions if it is obtained as a union of all covers $\mathcal{C}_e(t_j')$. Thus, the implementation in the ACGPER architecture only differs from the ACGPER architecture in the interpretation of the covers found for excitation slices.

Suppose that two exact covers were found for two connected sets of states. To find out if these two sets are parts of one connected set, the intersection of these two covers needs to be checked. If the intersection is non-empty, then two subsets are parts of one connected set of states; alternatively, they are two unconnected sets of states. From Properties 6.3.1 and 6.3.2

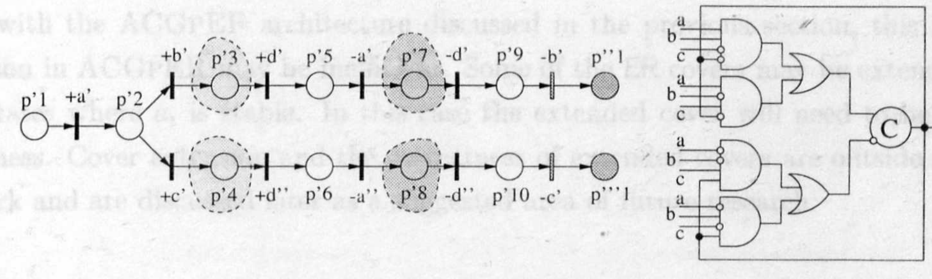


Figure 6.11: Illustration of slices and cover calculation for ACGPER architecture.

it follows that a slice represents a set of connected states. Hence, if the exact covers were obtained for two slices, then their non-empty intersection will indicate that these two slices represent two portions of one connected set of states.

Therefore, once the covers $\mathcal{C}_e(t'_j)$ are found, their intersection is checked iteratively. The iteration continues until the pairwise intersection in the newly obtained set of covers $\mathcal{C}_e(*a'_i), \mathcal{C}_e(*a''_i), \ldots \mathcal{C}_e(*a^n_i)$ is non-empty, i.e.:

$$\text{for each } ER_k(*a_i): \; \mathcal{C}^k_S(*a_i) = \sum_j \mathcal{C}_e(*a^j_i)$$

where for every pair $\mathcal{C}_e(*a'_i)$ and $\mathcal{C}_e(*a''_i)$ in $\mathcal{C}_e(*a'_i), \mathcal{C}_e(*a''_i), \ldots \mathcal{C}_e(*a^n_i)$ the following is true:

$$\text{iff } \mathcal{C}_e(*a'_i) \in \mathcal{C}^k_S(*a_i) \text{ and } \mathcal{C}_e(*a'_i) \cdot \mathcal{C}_e(*a''_i) \neq \emptyset \text{ then } \mathcal{C}_e(*a''_i) \in \mathcal{C}^k_S(*a_i).$$

The resulting set of covers represents the set (reset) function for a memory element in the ACGPER architecture.

The only exception is the case of a *fake conflict* (see Section 3.1). If two signal transitions are in fake conflict, then in the STG-unfolding segment there will be two (or more) excitation slices for each instance in fake conflict. However, when the STG-unfolding segment is constructed, fake conflicts are determined when the semi-modularity of STG is checked. Thus the union of two covers is taken even if two excitation slices are produced due to a fake conflict.

Example. Consider an STG-unfolding segment in Figure 6.11. For an output signal d the following excitation slices will be found from the STG-unfolding segment:

$$\mathcal{S}_e(+d') = \langle (p'_3), \{(p'_3)\}\rangle \qquad \mathcal{S}_e(+d'') = \langle (p'_4), \{(p'_4)\}\rangle$$
$$\mathcal{S}_e(-d') = \langle (p'_7), \{(p'_7)\}\rangle \qquad \mathcal{S}_e(-d'') = \langle (p'_8), \{(p'_8)\}\rangle$$

from which the covers for the excitation slice of each instance will be as follows:

$$\mathcal{C}_e(+d') = \{1100\} \quad \mathcal{C}_e(+d'') = \{1010\}$$
$$\mathcal{C}_e(-d') = \{0101\} \quad \mathcal{C}_e(-d'') = \{0011\}$$

The intersection of $\mathcal{C}_e(+d')$ and $\mathcal{C}_e(+d'')$ is empty as is the intersection of $\mathcal{C}_e(-d')$ and $\mathcal{C}_e(-d'')$. Thus these covers may be implemented as separate gates. The implementation is shown in Figure 6.11.

In the example in Figure 6.8 signal instances $+c'$ and $+c''$ are in fake conflict. Thus, although two slices will be found for instances of $+c$, there will be only one cover $\mathcal{C}_S(c) = \overline{b}\overline{c} + a\overline{c}$. ■

As with the ACGPEF architecture discussed in the previous section, this basic implementation in ACGPER may be inefficient. Some of the ER covers may be extended covering other states where a_i is stable. In this case the extended cover will need to be checked for correctness. Cover extension and the correctness of extended covers are outside the scope of this work and are discussed later as a suggested area of future research.

6.5 Strategies for Deriving Approximated Covers

The synthesis procedure described in the previous Sections has one drawback. If many concurrent transitions belong to a slice, then examining all cuts will suffer from an exponential explosion of states. Recall that the correctness criteria for each architecture allows the implemented cover to be greater than the exact cover. This can be exploited in an approximation method for covering the desired slices (on-, off- and excitation slices). This section describes the approximation approach and examines possible strategies for deriving the approximated covers from the STG-unfolding segment.

To help identify the rest of reachable states in an SG, a *Generalised Quiescent Region*, which has the opposite notion to the GER notion, is defined as:

Definition 6.5.1 The set of all states $GQR(+a_i)$ $(GQR(-a_i))$ of the SG S such that a_i is stable and for all states in $GQR(+a_i)$ $(GQR(-a_i))$ the value of the corresponding elements $v[i]$ is equal to 1 (0) is called a *Generalised Quiescent Region* (GQR) of $*a_i$. □

There are two specific sets of reachable states for each cover C satisfying the correctness requirement of any implementation architecture:

- the *Positive set*, denoted as $\mathcal{P}$-set, which is a set of states that **must** be covered by C; and
- the *Negative set*, denoted as $\mathcal{N}$-set, which is a set of states that **must not** be covered by C.

The choice of the $\mathcal{P}$-set and $\mathcal{N}$-set comes from the cover correctness conditions (Definitions 6.2.1, 6.2.3 and 6.2.5)and is made for the architectures as follows:

- ACGPS implementation: $\mathcal{P}$-set is taken as the on-set and $\mathcal{N}$-set is taken as the off-set of a particular signal a_i;
- ACGPEF implementation: the $\mathcal{P}$-set is taken as the $GER(+a_i)$ $(GER(-a_i))$ and the $\mathcal{N}$-set is taken as the rest of reachable states for the set (reset) function of a particular signal a_i, i.e. $\mathcal{N}$-set $= GQR(+a_i) \cup GER(-a_i) \cup GQR(-a_i)$ ($\mathcal{N}$-set $= GQR(-a_i) \cup GER(+a_i) \cup GQR(+a_i)$).
- ACGPER implementation: the $\mathcal{P}$-set is taken as the $ER_j(+a_i)$ $(ER_j(-a_i))$ and the $\mathcal{N}$-set is taken as the rest of reachable states for the set (reset) function of a particular signal a_i, i.e. $\mathcal{N}$-set $= \cup_{k,k\neq j} ER_k(+a_i) \cup GQR(+a_i) \cup GER(-a_i) \cup GQR(-a_i)$ ($\mathcal{N}$-set $= \cup_{k,k\neq j} ER_k(-a_i) \cup GQR(-a_i) \cup GER(+a_i) \cup GQR(+a_i)$).

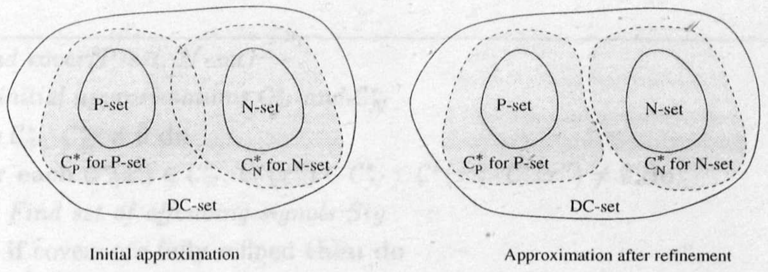


Figure 6.12: Illustration of the Negative set approximation strategy.

The cover is obtained by finding the partitioning of the STG-unfolding segment into slices which represents the $\mathcal{P}$-set. However, the $\mathcal{N}$-set also corresponds to some slices. The interpretation of both sets and the covers and slices in the STG-unfolding segment suggests (at least) two possible strategies for deriving the cover. Both strategies use *initial approximation* and *refinement* of covers, which are discussed in the following sections.

6.5.1 Negative Set Approximation

This strategy assumes that the initial approximation is found from the STG-unfolding segment not only for $\mathcal{C}_{\mathcal{P}}^*$, covering the states in the $\mathcal{P}$-set, but also for $\mathcal{C}_{\mathcal{N}}^*$, which covers the states in the $\mathcal{N}$-set. Thus the partitioning of the segment for $\mathcal{C}_{\mathcal{N}}$ is also required. Assume that the approximations $\mathcal{C}_{\mathcal{P}}^*$ and $\mathcal{C}_{\mathcal{N}}^*$ have been constructed so that all states in the $\mathcal{P}$-set and the $\mathcal{N}$-set, respectively, are covered. By making sure that $\mathcal{C}_{\mathcal{P}}^* \cdot \mathcal{C}_{\mathcal{N}}^* = \emptyset$ it is guaranteed that no state from $\mathcal{N}$-set is covered by $\mathcal{C}_{\mathcal{P}}^*$. However, if the intersection of approximations $\mathcal{C}_{\mathcal{P}}^*$ and $\mathcal{C}_{\mathcal{N}}^*$ is not empty, then the cover approximations must be *refined.* To assist the refinement, a set of *offending signals* is found, i.e. those signals which cause the covers to be loose. The refinement procedure "fills in" literals for some of the offending signals and produces covers for a smaller number binary vectors. Eventually, in the worst case, after full refinement, it must produce exact covers for the $\mathcal{P}$-set and the $\mathcal{N}$-set. If the intersection of the $\mathcal{P}$-set and the $\mathcal{N}$-set is empty, then the exact covers for these sets will also have an empty intersection. However, if the STG does not satisfy the CSC condition, then the intersection of the $\mathcal{P}$-set and the $\mathcal{N}$-set will not be empty. Hence, if the refinement procedure terminates with fully refined covers but their intersection is still non-empty, then this STG does not satisfy the CSC condition. While the refinement is performed, no optimisation is done. Two fully refined covers can only intersect on minterms. Since minterms have all literals present, the set of offending signals will be empty. Thus, if after the refinement step the set of offending signals is empty, the CSC problem is reported. The general idea behind this strategy is illustrated in Figure 6.12.

This strategy finds approximations $\mathcal{C}_{\mathcal{P}}^*$ and $\mathcal{C}_{\mathcal{N}}^*$ that cover the required sets of states and partition the set of combinations which are allowed to be covered by them into two disjoint sets. It therefore produces *pessimistic covers* as it does not allow the combinations from the DC-set to be shared by both covers. The pseudo-code of the procedure implementing this strategy is shown in Figure 6.13.

For example, in the ACGPS architecture the $\mathcal{P}$-set and $\mathcal{N}$-set are taken as the on- and off-set. The on- and off-sets are disjoint by construction for a CSC-compliant STG. If approx-

```
proc Find cover($\mathcal{P}$-set, $\mathcal{N}$-set)
    Find initial approximations $\mathcal{C}^*_{\mathcal{P}}$ and $\mathcal{C}^*_{\mathcal{N}}$
    while $\mathcal{C}^*_{\mathcal{P}} \cdot \mathcal{C}^*_{\mathcal{N}} \neq \emptyset$ do
        for each $\mathcal{C}^*(x') \in \mathcal{C}^*_{\mathcal{P}}$, $\mathcal{C}^*(x'') \in \mathcal{C}^*_{\mathcal{N}} : \mathcal{C}^*(x') \cdot \mathcal{C}^*(x'') \neq \emptyset$ do
            Find set of offending signals Sig
            if covers are fully refined then do
                Report CSC problem and TERMINATE
            end do
            else do
                Refine cover $(\mathcal{C}(x'), Sig)$
                Refine cover $(\mathcal{C}(x''), Sig)$
            end do
        end do
    end do
    return $\mathcal{C}^*_{\mathcal{P}}$
end proc
```

Figure 6.13: Procedure for the Negative set approximation strategy.

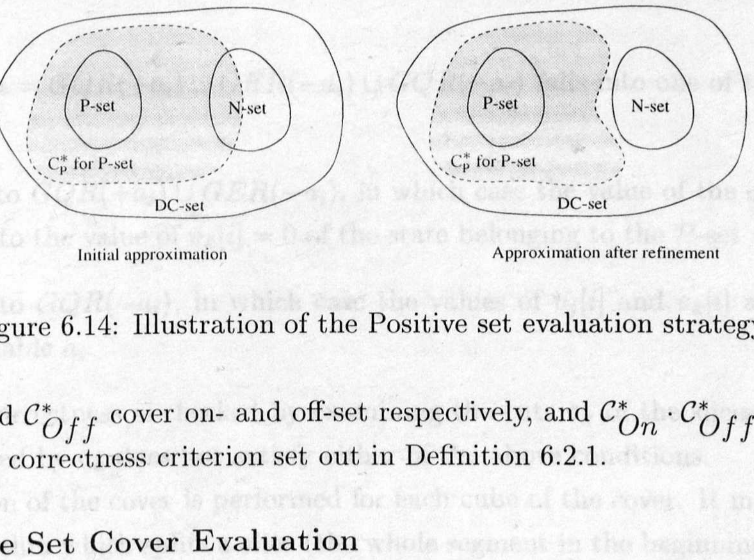


Figure 6.14: Illustration of the Positive set evaluation strategy.

imations $\mathcal{C}^*_{On}$ and $\mathcal{C}^*_{Off}$ cover on- and off-set respectively, and $\mathcal{C}^*_{On} \cdot \mathcal{C}^*_{Off} = \emptyset$, then these covers satisfy the correctness criterion set out in Definition 6.2.1.

6.5.2 Positive Set Cover Evaluation

This strategy assumes that only the approximation $\mathcal{C}^*_{\mathcal{P}}$ covering the $\mathcal{P}$-set is found in the STG-unfolding segment. The cover $\mathcal{C}^*_{\mathcal{P}}$ is then *evaluated* by finding where in the STG-unfolding segment it becomes TRUE. The evaluation finds a set of slices $\mathcal{S}$ which represent all states in which the cover becomes TRUE. If no cut in $\mathcal{S}$ represents a state from the $\mathcal{N}$-set, then the cover satisfies the correctness criterion. Otherwise, approximation $\mathcal{C}^*_{\mathcal{P}}$ is refined until no cut represents a state from the $\mathcal{N}$-set. As with the previous strategy, no optimisation is performed until the refinement procedure finishes. Therefore, the set of offending signals is found for each cover $\mathcal{C}(x')$ comprising the $\mathcal{P}$-set cover approximation. The refinement procedure uses this set of offending signals to tighten up the cover approximation and eliminate the intersection. The general idea behind this strategy is illustrated in Figure 6.14. When $\mathcal{C}^*_{\mathcal{P}}$ is fully refined but such a cut still exists, the refining procedure aborts and reports the CSC violation. The pseudo-code of the procedure for this strategy is shown in Figure 6.15.

For example, suppose that $\mathcal{C}^*_{S}(a_i)$ was found for the ACGpEF implementation. Any state

```
proc Find cover($\mathcal{P}$-set, $\mathcal{N}$-set)
    Find initial approximation $C^*_{\mathcal{P}}$
    Evaluate $C^*_{\mathcal{P}}$
    while some cut from $\mathcal{S}$ is in $\mathcal{N}$-set do
        for each $C(x')$ covering cuts in $\mathcal{N}$-set do
            Find set of offending signals Sig
            if covers are fully refined then do
                Report CSC problem and TERMINATE
            end do
            else do
                Refine cover ($C^*(x'), Sig$)
            end do
        end do
        Evaluate $C^*_{\mathcal{P}}$
    end do
    return $C^*_{\mathcal{P}}$
end proc
```

Figure 6.15: Procedure for the Positive set cover evaluation strategy.

s_j from the $\mathcal{N}$-set $= GQR(+a_i) \cup GER(-a_i) \cup GQR(-a_i)$ falls into one of the two following categories:

- s_j belongs to $GQR(+a_i) \cup GER(-a_i)$, in which case the value of the element $v_j[i] = 1$ is opposite to the value of $v_k[i] = 0$ of the state belonging to the $\mathcal{P}$-set $= GER(+a_i)$; or
- s_j belongs to $GQR(-a_i)$, in which case the values of $v_j[i]$ and $v_k[i]$ are equal, but s_j does not enable a_i.

Thus the cover correctness is checked by examining the cuts c_k of the slices $\mathcal{S}$ and checking that s_k represented by c_k does not satisfy either of the above conditions.

The evaluation of the cover is performed for each cube of the cover. It may use a branch-and-bound algorithm which splits a slice (the whole segment in the beginning) into sub-slices according to the literals present in each cube.

6.5.3 Concluding Remarks

The fundamental difference between the discussed strategies is that the Negative set approximation strategy attempts to find an approximate cover for the set of states which must not be covered by the $\mathcal{P}$-set cover. On the contrary, the Positive set evaluation strategy only checks that no state covered by the $\mathcal{P}$-set cover belongs to the $\mathcal{N}$-set.

The implementation in the ACGPER architecture is a special case of the approximation technique. It roots from the fact that any circuit implementable in the ACGPER architecture can be implemented in the ACGPEF architecture.

Consider finding the set function cover for the ACGPER architecture using the Negative set approximation strategy. Recall that the Set and Reset partitioning of the STG-unfolding segment does not provide distinction between excitation slices belonging to different ERs.

Such distinction can only be made after the exact covers for all slices have been found. Thus it is impossible to construct, without knowing the exact covers, an approximation of the $\cup_{k,k\neq j}ER_k(+a_i)$ part of the $\mathcal{N}$-set, i.e. an approximation which, after full refinement, will only cover states in $\cup_{k,k\neq j}ER_k(+a_i)$. Thus the $\mathcal{P}$-set cover approximation must be taken for the ACGPER architecture to be the same as for the ACGPEF architecture, i.e. the GER cover approximation.

On the other hand, slices and cover approximations for GERs and GQRs are clearly distinguishable. Therefore, if an approximated cover for the $\mathcal{P}$-set does not intersect with the approximation of the $GQR(+a_i) \cup GER(-a_i) \cup GQR(-a_i)$ part of the $\mathcal{N}$-set, then an approximated cover for the GER can be obtained from a set of approximation covers for each slice. Two approximated covers found from two excitation slices can intersect either (1) on reachable states or (2) on the DC-set or (3) both. However, since the DC-set is unknown, none of these cases can be distinguished. After the approximated covers are obtained, they are interpreted in a similar way to their interpretation when the covers are calculated exactly.

Consider now the Positive set cover evaluation strategy for the ACGPER architecture. Since the $\cup_{k,k\neq j}ER_k(+a_i)$ part of the $\mathcal{N}$-set is not available a priory, no specific properties can be drawn for the states in $\cup_{k,k\neq j}ER_k(+a_i)$ which distinguish these states from those in $ER_j(+a_i)$. The properties of the rest of the states in the $\mathcal{N}$-set can be clearly identified (as in the earlier example for the ACGPEF architecture). Hence the cover approximation needs to be taken as the GER cover which is then refined until it satisfies the ACGPEF correctness condition. Once the correct cover approximation for GER is found, the covers for each individual excitation slice are interpreted. This interpretation produces an implementation in ACGPER architecture. Note, however, that the exact covers are produced in the worst case only (full refinement). At the same time, if the approximated covers have an intersection on the DC-set, the correctness conditions for the ACGPEF architecture may be satisfied for approximated covers. In this case a pessimistic implementation is produced where intersecting covers are implemented as one atomic complex gate.

For the reasons explained above the last two architectures are considered together where the approximation method is used.

The Positive set cover evaluation strategy works with the exact negative set. Therefore the results of the approximation should be slightly better than the results from the Negative set approximation cover. Indeed, the Positive set cover evaluation will produce a satisfiable cover as soon as no states from the negative set are covered by the approximation. However, to achieve this it requires the evaluation procedure which is more complex than the refinement of both approximations for the $\mathcal{P}$-set and $\mathcal{N}$-set. Furthermore, in the ACGPS architecture both covers are required for the implementation and hence the refinement process obtains both cover approximations at the same time. For these reasons, the Positive cover evaluation strategy is not considered in this thesis.

6.6 Initial Cover Approximation

The slices in the STG-unfolding segment represent two types of states: those where a signal transition $*a_i$ is excited ($GER(*a_i)$) and those in which the signal a_i is stable ($GQR(*a_i)$). At the STG-unfolding level, the instances of places and transitions are available. Each transition instance t' identifies the set of place instances which are immediate predecessors of t'. All cuts enabling t' will include places from $\bullet t'$. Therefore the approximation for all cuts enabling t' can be found as an approximation for the partial cut $\mathsf{c}^* = \bullet t'$. Similarly, each place instance p' which is sequential to the signal transition instance $*a_i'$ but precedes transitions from $next(*a_i')$ can only be marked in the states from $GQR(*a_i)$. Hence the states at which p' is marked can be approximated by finding a cover approximation for the partial cut $\mathsf{c}^* = (p')$.

Therefore, the problem of finding a cover approximation of the slice $\mathcal{S}$ is restated as a problem of finding cover approximations for partial cuts which are formed by some elements of the STG-unfolding segment belonging to $\mathcal{S}$.

6.6.1 Partial Cut Cover Approximation

Consider obtaining a cover approximation for a partial cut c^* such that exists a cut $\mathsf{c} : \mathsf{c}^* \subseteq \mathsf{c}$ and c is encapsulated by a slice $\mathcal{S}$. $\mathcal{S}$ represents a connected set of states which are represented as cuts in $\mathcal{S}$. Any state in this connected set of states in which all places p' of c^* are marked is reached by firing the instances which belong to the slice $\mathcal{S}$. If a signal instance $*a_i'$ is concurrent to all places in c^*, then the value of its corresponding element in the binary vector may take values of both "0" and "1". Therefore the cover approximation for the partial cut is defined as follows.

Definition 6.6.1 A cover approximation $C^*(\mathsf{c}^*)$ is called a *partial cut approximation cover for* c^* and is calculated as follows:

$$C^*(\mathsf{c}^*)[i] = \begin{cases} -, & \text{if } *a_i' \in \{t_l' | (\exists \mathsf{c}_1, \mathsf{c}_2 \in \mathcal{S} : \mathsf{c}_1 \xrightarrow{t_l'} \mathsf{c}_2) \wedge (\forall p' \in \mathsf{c}^* : t_l' \| p')\} \\ \xi_{\mathsf{c}^*}[i], & \text{otherwise} \end{cases}$$

where ξ_{c^*} is calculated from the signal states of the local configurations of instances $\bullet p_j'$: $p_j' \in \mathsf{c}^* = \{p_1', p_2', \ldots p_n'\}$, using the cumulative states of local configurations $\zeta_{\lceil \bullet p_j' \rceil}$ as:

$$\begin{aligned} \zeta_{\mathsf{c}^*} &= \sum_j \zeta_{\lceil \bullet p_j' \rceil} : p_j' \in \mathsf{c}^* \\ \xi_{\mathsf{c}^*} &= \xi_0 \oplus \zeta_{\mathsf{c}^*} \end{aligned}$$

where ξ_0 is the signal state of the initial transition $\perp$, i.e. the binary vector of the initial state s_0. □

In other words, the literals corresponding to the signals whose instances belong to $\mathcal{S}$ and are concurrent to places in the partial cut c^* are substituted by "–" (don't care).

Property 6.6.1 Any instance $*a_j'$ is concurrent to all instances $p' \in \bullet(*a_i')$ iff $*a_j' \| * a_i'$. □

Property 6.6.2 All instances $*a_j'$ of the STG-unfolding segment which are concurrent to $*a_i'$ belong to the excitation slice of $*a_i'$.

Proof: Follows from the definition of the excitation slice for $*a_i'$ (Section 6.4).

□

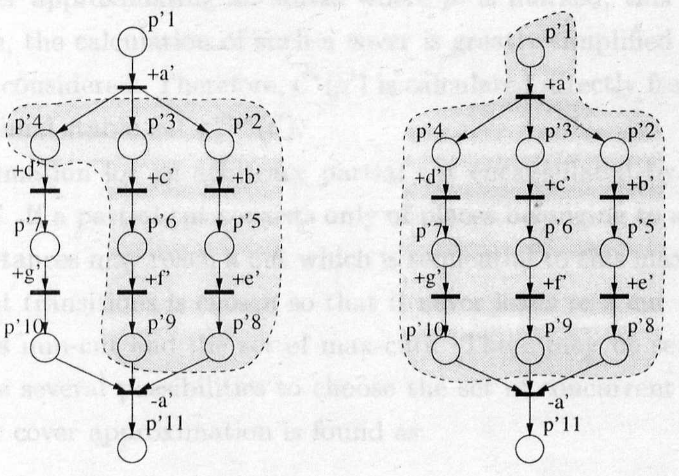


Figure 6.16: Illustration of cover approximation for a place (a) and a slice (b).

Following the above property, the cover approximation for the partial cut $\bullet(*a'_i)$ for a transition defining its excitation slice $\mathcal{S}_e(*a'_i)$ can be found by examining all instances concurrent to $*a'_i$ (no check is needed if $*a'_j$ belongs to $\mathcal{S}_e(*a'_i)$). The approximated cover for all states represented in an excitation slice of $*a'_i$ is denoted as $\mathcal{C}^*_e(*a'_i)$. Furthermore, by its definition, the binary vector ξ_{C^*} is equal to the signal state of the configuration producing the minimal excitation cut of $*a'_i$. The minimal excitation and minimal stable cuts are only different in the value of $\xi_{\lceil *a'_i \rceil}[i]$ as $*a'_i$ is the only instance separating them. Thus, instead of computing the binary state ξ_{C^*} from the cumulative states of predecessors of $\bullet(*a'_i)$, it can be found from $\xi_{\lceil *a'_i \rceil}$ by inverting the value of $\xi_{\lceil *a'_i \rceil}[i]$.

Example. Consider the cover approximation for $\bullet(+d')$ in Figure 6.16(a). The binary vector of its minimal excitation cut $\mathsf{c}_e^{min}(+d') = (p'_2, p'_3, p'_4)$ is found from the signal state of its local configuration as $\xi = \{1000000\}$ (the order of signals is $abcdefg$). There are four signals $\{b, c, e, f\}$ whose instances belong to the excitation slice of $+d'$ $\mathcal{S}_e(+d') = \langle (p'_2, p'_3, p'_4))\{(p'_4, p'_8, p'_9)\}\rangle$ and are concurrent to $+d'$. Thus the cover approximation for $+d'$ will be $\mathcal{C}^*_e(+d') = \{1--0--0\} = a\overline{d}\overline{g}$. ■

Property 6.6.3 The cover approximation $\mathcal{C}^*_e(+a'_i)$ ($\mathcal{C}^*_e(-a'_i)$) obtained from the STG-unfolding segment of an STG satisfying the general specification correctness criteria (Section 6.2) always has the value of the element corresponding to a_i set to "0" ("1").

Proof: Follows from the fact that in an STG-unfolding segment constructed for an STG satisfying general specification correctness criteria no two instances of a_i can be concurrent. Thus the element corresponding to a_i will remain equal to that of $\xi[i]$ assigned to the minimal excitation cut $\mathsf{c}_e^{min}(+a'_i)$ ($\mathsf{c}_e^{min}(-a'_i)$), i.e. "0" ("1"). □

A partial cut by definition is any set of mutually concurrent instances p'. Thus a single instance p' is also a partial cut. In this case, the partial cut approximation cover can be viewed as a cover approximating all states where p' is marked; this cover is denoted as $\mathcal{C}^*(p')$. Furthermore, the calculation of such a cover is greatly simplified as there is only one predecessor t' to be considered. Therefore, $\mathcal{C}^*(p')$ is calculated directly from the binary vector assigned to the minimal stable cut $\mathsf{c}_s^{min}(t')$.

A cover approximation for an arbitrary partial cut encapsulated by a slice $\mathcal{S}$ must not cover cuts outside $\mathcal{S}$. If a partial cut consists only of places belonging to a max-cut, the firing of all concurrent instances may reach a cut which is sequential to this max-cut. To avoid this, the set of concurrent transitions is chosen so that it never leads to a cut which is outside the bounds of $\mathcal{S}$, i.e. its min-cut and the set of max-cuts. There may be several max-cuts and, hence, there may be several possibilities to choose the set of concurrent transitions. In this case, the partial cut cover approximation is found as:

$$\mathcal{C}^*(\mathsf{c}^*) = \sum_k \mathcal{C}^*_{t'_k}(\mathsf{c}^*),$$

where $\mathcal{C}^*_{t'_k}(\mathsf{c}^*)$ is found using a restricted set of concurrent instances, i.e.:

$$\mathcal{C}^*_{t'_k}(\mathsf{c}^*)[i] = \begin{cases} -, & \text{if } a'_i \in \{t'_l | (k \neq l) \wedge (\exists \mathsf{c}_1, \mathsf{c}_2 \in \mathcal{S} : \mathsf{c}_1 \xrightarrow{t'_l} \mathsf{c}_2) \wedge (\forall p' \in \mathsf{c}^* : \; t'_l \| p')\} \\ \xi_{\mathsf{c}^*}[i], & \text{otherwise} \end{cases}$$

for each t'_k such that $(t'_k \bullet \cup \mathsf{c}^*) \subseteq \mathsf{c} : \; \mathsf{c} \in \mathsf{C}^{max}$. There are no successor transitions of t' as $t'\bullet$ forms the max-cut and none of its successors belongs to $\mathcal{S}$.

Example. Consider an approximation of the partial cut covers for cuts consisting of single places belonging to $\mathcal{S}(+a') = \langle (p'_1), \{(p'_7, p'_8, p'_9), (p'_6, p'_8, p'_{10}), (p'_5, p'_9, p'_{10})\}\rangle$ in the example shown in Figure 6.16(b). The approximation cover for $\mathsf{c} = \{p'_4\}$ is found from $\xi_{\lceil +a' \rceil}$ as $\mathcal{C}^*(p'_4) = \{1--0--0\} = a\overline{d}\overline{g}$. Place p'_{10}, on the other hand, belongs to at least one max-cut. Thus the approximation cover for $\mathsf{c}^* = \{p'_{10}\}$ is found using two approximations corresponding to two different sets of concurrent transitions:

$$\begin{aligned} \mathcal{C}^*(p'_{10}) &= \mathcal{C}^*_{f'}(p'_{10}) \;\; (\text{for } \{b', c', e'\}) \\ &+ \mathcal{C}^*_{e'}(p'_{10}) \;\; (\text{for } \{b', c', f'\}) \\ &= \{1--1-01\} \cup \{1--10-1\} = ad\overline{f}g + ad\overline{e}g. \end{aligned}$$

■

Property 6.6.4 Let a partial cut c^* consist of one place p' belonging to the on-slice (off-slice) $\mathcal{S}_{On}(+a'_i) : \; +a'_i \prec p' \; (\mathcal{S}_{Off}(+a'_i) : \; -a'_i \prec p')$ in the STG-unfolding segment of an STG satisfying the general specification correctness criteria (Section 6.2). The cover approximation $\mathcal{C}^*(\mathsf{c}^*)$ always has the literal (compliment of the literal) corresponding to a_i.

Proof: Follows from the fact that the on-set (off-set) slice contains no instances of a_i except for $+a'_i$ $(-a'_i)$, which is used to define it. Thus there are no instances of a_i in the slice which are concurrent to p'. Since p' is sequential to $+a'_i$ $(-a'_i)$, the value of the corresponding element will remain the same as $\xi_{\lceil +a'_i \rceil}[i] = 1$ $(\xi_{\lceil -a'_i \rceil}[i] = 0)$, corresponding to the minimal stable cut of $+a'_i$ $(-a'_i)$.

□

6.6.2 Finding $\mathcal{P}$-set and $\mathcal{N}$-set Approximations

Since the Negative cover approximation strategy requires approximation covers for both $\mathcal{P}$-set and $\mathcal{N}$-set, consider finding those for each implementation architecture. The Positive set evaluation strategy only uses the $\mathcal{P}$-set approximation. For simplicity only the on-set covers and covers for $GER(+a_i)$ are considered further. The complimentary cover approximations are found by using the instances of $-a_i$.

ACGPS Implementation

After the On- and Off-set partitioning w.r.t signal a_i was found, the slices $\mathcal{S}_{\mathcal{O}n}(+a_i')$ and $\mathcal{S}_{\mathcal{O}n}(-a_i')$ represent the cuts starting from those enabling each instance $*a_i'$ up to (but not including) the cuts enabling instances from $next(*a_i')$. To approximate the cuts enabling $*a_i'$ the approximation cover $\mathcal{C}_e(*a_i')$ for the partial cut $\mathsf{c}^* = \bullet(*a_i')$ is used. To find the approximation for the rest of cuts in $\mathcal{S}_{\mathcal{O}n}(+a_i')$ and $\mathcal{S}_{\mathcal{O}n}(-a_i')$ partial cover approximations are found for cuts consisting of place instances which belong to $\mathcal{S}_{\mathcal{O}n}(+a_i')$ and $\mathcal{S}_{\mathcal{O}n}(-a_i')$, respectively.

A cover approximation for a partial cut $\mathsf{c}^* = \{p_l'\}$ will cover all states in which p_l' is marked together with any other concurrent place p_j'. Therefore, only mutually non-concurrent subset of places belonging to $\mathcal{S}_{\mathcal{O}n}(+a_i')$ can be considered. A set of such places is found in the STG-unfolding segment.

Definition 6.6.2 A maximal set P_a' of places belonging to a slice $\mathcal{S}_{\mathcal{O}n}(+a_i')$ such that $\forall p_k' \in P_a' : +a_i' \prec p_k'$ (if such $+a_i'$ exists) and $\forall p_k', p_l' \in P_a' : p_k' \not\| p_l'$ is called an *approximation set of the slice* $\mathcal{S}_{\mathcal{O}n}(+a_i')$. □

The approximation set for a slice is a "skeleton" of places for this slice which are either sequential or in conflict with each other. Any cut encapsulated by $\mathcal{S}_{\mathcal{O}n}(+a_i')$ will contain a place from its approximation set.

Definition 6.6.3 The cover $C^*_{\mathcal{O}n}(a_i)$ is called the *initial on-set cover approximation* and is calculated from the on-slices $\mathcal{S}_{\mathcal{O}n}(+a_i')$ as:

$$C^*_{\mathcal{O}n}(a_i) = \sum_k C^*_{\mathcal{O}n}(+a_i^k).$$

where

$$C^*_{\mathcal{O}n}(+a_i^k) = C^*_e(+a_i^k) + \sum_l C^*(p_l'),\ p_l' \in P'_{+a_i^k}$$

for each $\mathcal{S}_{\mathcal{O}n}(+a_i^k)$ defined using $\mathsf{c}_e^{min}(+a_i^k)$ as its its min-cut and

$$C^*_{\mathcal{O}n}(+a_i^k) = \sum_l C^*(p_l'),\ p_l' \in P'_{+a_i^k}$$

if $\mathcal{S}_{\mathcal{O}n}(+a_i^k)$ was defined using $\mathsf{c}_s^{min}(\perp)$. □

Definition 6.6.4 The cover $C^*_{\mathcal{O}ff}(a_i)$ is called the *initial off-set cover approximation* and is calculated from the off-slices $\mathcal{S}_{\mathcal{O}ff}('-a_i')$ as:

$$C^*_{\mathcal{O}ff}(a_i) = \sum_k C^*_{\mathcal{O}ff}(-a_i^k).$$

where

$$C^*_{Off}(-a_i^k) = C_e^*(-a_i^k) + \sum_l C^*(p_l'),\ p_l' \in P'_{-a_i^k}$$

for each $\mathcal{S}_{On}(-a_i^k)$ defined using $\mathsf{c}_e^{min}(-a_i^k)$ as its its min-cut and

$$C^*_{On}(-a_i^k) = \sum_l C^*(p_l'),\ p_l' \in P'_{-a_i^k}$$

if $\mathcal{S}_{On}(-a_i^k)$ was defined using $\mathsf{c}_s^{min}(\perp)$. □

The initial cover approximations for the on- and off-sets are taken as the initial cover approximations for the $\mathcal{P}$-set and $\mathcal{N}$-set.

Proposition 6.6.1 For any STG satisfying the general specification correctness criteria (Section 6.2), the on-set cover approximation $C^*_{On}(a_i)$, obtained from its STG-unfolding segment, covers all reachable states in $On(a_i)$.

Proof: Since there are two types of states in $On(a_i)$, the proof is split into two parts: for those states where $+a_i$ is excited and for those states where a_i is stable. The proof for the former (represented by $\mathcal{S}_e(+a_i')$ which is included in $\mathcal{S}_{On}(+a_i')$) is given later in Proposition 6.6.2; only the proof that a cut corresponding to any state from $On(a_i)$ in which a_i is stable is covered by its $C^*_{On}(a_i)$ is given here.

Every reachable state of an STG is represented in the STG-unfolding segment. Therefore, for every state s in $On(a_i)$ there exists a corresponding cut in the segment and no instance in c is a successor of a cutoff transition (Lemma 5.2.1). Every such cut c is sequential to the minimal stable cut of some $+a_i'$ (or $\perp$). Consider the set of places belonging to this cut; one of these places, say p', will be chosen in to the approximation set P_a'. Furthermore, c is reached from $\mathsf{c}_s^{min}(\bullet p')$ through the firing of transition instances which are concurrent to p' and belong to $\mathcal{S}_{On}(+a_i')$. The cover approximation $C^*(p')$ will have "-" for all literals corresponding to the signals whose instances are concurrent to p' and will, therefore, include the exact cover $C(\mathsf{c})$. Hence $C^*_{On}(a_i)$ includes $C(\mathsf{c})$ and covers s.

□

A similar statement holds for the off-set cover approximation and the set of states in the off-set.

ACGPEF and ACGPER Implementations

It was discussed earlier that the $\mathcal{P}$-set and $\mathcal{N}$-set cover approximations for the ACGPER can only be chosen the same as for the ACGPEF implementation architecture. Also consider only the calculation of the set function. For the reset function the instances of $-a_i$ are used. The $\mathcal{P}$-set and $\mathcal{N}$-set in these architectures are $\mathsf{GER}(+a_i)$ and the rest of reachable states, respectively. In the discussion above, for the ACGPS implementation, two separate approximations were used for the states in which $+a_i$ is excited and in which a_i is stable. Thus the $\mathcal{P}$-set and $\mathcal{N}$-set cover approximations can be found by obtaining the On- and Off-set partitioning of the STG-unfolding segment.

Definition 6.6.5 The cover approximation $C^*_S(a_i)$ is called the *initial set cover approximation for signal transition* $+a_i$ and is calculated from the excitation slices of $+a_i^k$ as:

$$C^*_S(a_i) = \sum_k C^*_e(+a_i^k)$$

□

Proposition 6.6.2 For any STG satisfying the general specification correctness criteria (Section 6.2), the set cover approximation $C^*_S(a_i)$, obtained from its STG-unfolding segment, covers all reachable states where $+a_i$ is excited.

Proof: Every reachable state s enabling $+a_i$ is represented in the STG-unfolding segment and the cut c corresponding to s does not contain any successors of cutoff transitions (Lemma 5.2.1). Furthermore, there exists a corresponding instance $+a_i^l$. Hence there will exist a slice $\mathcal{S}_e(+a_i^l)$ defined by the minimal excitation and minimal stable cuts of $+a_i^l$ and, also c is encapsulated by $\mathcal{S}_e(+a_i^l)$.

c is reached from $\mathsf{c}_e^{min}(+a_i^l)$ through the firing of transition instances which are concurrent to $+a_i^l$ and belong to $\mathcal{S}_e(+a_i^l)$. The cover approximation $C^*(+a_i^l)$ will have "-" for all literals corresponding to the signals whose instances are concurrent to $+a_i^l$ and will, therefore, include the exact cover $C(\mathsf{c})$. Hence $C^*_e(a_i)$ includes $C(\mathsf{c})$ and covers s.

□

Definition 6.6.6 The cover approximation $C^*_{\mathcal{N}}$ is called the *initial $\mathcal{N}$-set cover approximation for* ACGPEF *(*ACGPEF*) architecture* for the set cover approximation and is found as:

$$C^*_{\mathcal{N}} = \sum_l C^*(p_l') + \sum_n C^*_e(-a_i^n) + \sum_k C^*(p_k')$$

where p_l' and p_k' belong to the approximation sets found for On- and Off-set partitioning, respectively, as in the previous architecture. □

Obviously, similar definitions exist for the reset cover of $*a_i$ and a statement similar to Proposition 6.6.2 holds.

The set and reset cover approximations for the ACGPER architecture are found by interpretation of the cover approximations for each excitation slice after the approximations for ACGPEF architecture are found.

6.6.3 Correctness of Negative Set Approximation Strategy

It is demonstrated here that the Negative set approximation strategy produces a correct implementation for any STG satisfying CSC.

Proposition 6.6.3 Let the $\mathcal{P}$-set and the $\mathcal{N}$-set be chosen as on- and off-sets of a_i, respectively. Let the cover approximations $C^*_{On}(a_i)$ and $C^*_{Off}(a_i)$ be calculated from the segment for a CSC-compliant STG as per Definitions 6.6.3 and 6.6.4. Then the Negative set approximation procedure given in Figure 6.13 produces correct covers for ACGPS architecture iff the refinement procedure in the worst case restores the exact covers for On- and Off-set partitioning after a finite number of iterations.

Proof: (1) According to Proposition 6.6.1 the above approximation method produces covers for the on- and off-sets of a_i. (2) The procedure in Figure 6.13 exits only if the covers are disjoint or terminates if the STG does not have CSC. For a CSC-compliant STG the on- and off-sets of a_i are disjoint. Therefore, if the refinement procedure in the worst case restores exact covers for each on- and off-slice of the partitioning, then the states covered by these restored covers will belong only to on- and off-set, respectively. Thus the refined covers will satisfy the correctness criterion for the ACGPS architecture (Definition 6.2.1).

□

The next proposition illustrates that covers found for ACGPEF and ACGPER using our strategy are also correct.

Proposition 6.6.4 Let the $\mathcal{P}$-set and the $\mathcal{N}$-set be chosen as $GER(+a_i)$ and the rest of reachable states, respectively. Let the cover approximations $\mathcal{C}^*_e(+a_i)$ and $\mathcal{C}^*_\mathcal{N}$ be calculated from the segment for a CSC-compliant STG as per Definitions 6.6.5 and 6.6.6. Then the Negative set approximation procedure given in Figure 6.13 produces correct covers for ACGPEF and ACGPER architectures iff the refinement procedure in the worst case restores the exact covers for all cuts including $\bullet(+a'_i)$ and the cuts including the place instances used to find $\mathcal{C}^*_\mathcal{P}(a_i)$ and $\mathcal{C}^*_\mathcal{N}(a_i)$ in a finite number of iterations.

Proof: (1) According to Proposition 6.6.2 the above approximation method produces covers for $GER(+a_i)$. (2) There are three components in the $\mathcal{N}$-set cover approximation $\mathcal{C}^*_\mathcal{N}$. From the properties of transition and place instance cover approximations it follows that the cover approximations $\sum \mathcal{C}^*(p'_l)$ and $\sum \mathcal{C}^*_e(-a'_i)$ will have an element corresponding to a_i set to "1". This element will always be set to "0" in the $\mathcal{P}$-set cover $\mathcal{C}^*_e(+a_i)$. Therefore, the only intersection between the $\mathcal{P}$-set and the $\mathcal{N}$-set cover approximations may happen if some $\mathcal{C}^*_e(+a'_i)$ intersects with $\sum \mathcal{C}^*(p'_k)$. The former corresponds to the states where signal transition $+a_i$ is excited and the latter to those where a_i is stable at "0". In a CSC-compliant STG intersection between these two sets of states is empty. Thus, if the refinement procedure in the worst case produces exact covers, then these covers satisfy the correctness criteria for these architectures (Definitions 6.2.3 and 6.2.5, respectively).

□

Note that the above proposition also showed that only intersection between the cover approximations for the excitation slices and the places of the Off-set partitioning needs to be checked when the set function is found for the implementation in the ACGPEF or ACGPER architectures. This reduces the number of cover intersections to be checked and, hence, reduces the synthesis time.

6.7 Cover Refinement

The purpose of the refining procedure is to restore some of the relations between concurrent transitions belonging to a slice S which were ignored during the approximation.

The refinement is used in both cover derivation strategies. In the first one, Negative set approximation, the refinement procedure is called when cover approximations $C^*_{\mathcal{P}}$ and $C^*_{\mathcal{N}}$ for the $\mathcal{P}$-set and $\mathcal{N}$-set, respectively, have a non-empty intersection. Furthermore, the cover approximations $C^*_{\mathcal{P}}$ and $C^*_{\mathcal{N}}$ are built from the covers of transition and place instances. Thus when the intersection is non-empty, at least two instances x' and x'' causing it can be identified. An intersection check can be performed on the "per cube" basis, which narrows the intersection to a particular pair of cubes from which the set of offending signals Sig is found. These are the signals whose corresponding literals are missing from a particular cube of the cover.

In the second strategy the cover approximation is evaluated. The $\mathcal{P}$-set cover approximation $C^*_{\mathcal{P}}$ is also formed from the cover approximations of particular instances. If $C^*_{\mathcal{P}}$ covers some states from the $\mathcal{N}$-set, then it is possible to find at least one instance whose cover approximation covers these states. The set of offending signals in this case will be all signals whose literals are missing from at least one of the cubes of the cover.

Obviously, there may be several choices of instances whose cover approximations cause a non-empty intersection in the first strategy or whose cover approximation covers states from the $\mathcal{N}$-set in the second. Refining one of them at a time will, however, correct the approximated cover.

The refinement procedure, therefore, needs to be able to refine a cover for an element of unfolding x' with a given set of offending signals. The instance x' can be either a place or a transition instance. If x' is a place instance, then, by definition, this place forms a partial cut. If x' is a transition instance, then its approximated cover is in fact a cover for a partial cut including instances of places in $\bullet x'$. Hence the refinement procedure should work with partial cuts.

Recall that each instance x' belongs to some slice of the appropriate partitioning. The refinement procedure needs to find some of the places which can be marked together with the places already in a partial cut c^* and produce new partial cuts for each combination, i.e. *restoring partial cuts.* When a new partial cut is formed, a new cube is introduced into the cover which represents this partial cut. Eventually, when all partial cuts are restored to ordinary cuts, the refinement procedure terminates with the cover consisting of the cubes found for each cut. Any existing minimisation technique can be used to optimise this cover by standard boolean transformations. Special attention is required for those partial cuts which are part of at least one max-cut of $\mathcal{S}$. Their initial cover approximation consists of several cubes $B_{t'}$ each of which does not allow some instance t' to be used in the approximation. Thus in any further refinements of these covers any cubes derived from $B_{t'}$ may not be refined using t'.

The refinement of the cover for element x' is based on finding the refinement set for x' with respect to an offending signal a_j.

Definition 6.7.1 A maximal set of places P'_r that belong to the slice $\mathcal{S}$ is called the *refining set of x' with respect to a_j* iff it satisfies the following: $\forall p'_k \in P'_r : x' \| p'_k$, $\forall p'_l \in P'_r : p'_k \not\| p'_l$ and $\forall *a'_j \in \mathcal{S} : (*a'_j) \bullet \cap P'_r \neq \emptyset$ □

In other words, the refining set is a set of mutually non-concurrent places belonging to $\mathcal{S}$ which are pair-wise concurrent with x'. The *inclusion of the successors* of each $*a'_j$ into P'_r

is required for the progress of the refinement procedure. Thus, if a_j was causing the cover to fail the correctness condition, the literal corresponding to a_j will be present in all cubes of the cover approximation of x' after this iteration. In addition, at least one signal will be refined at each iteration.

Suppose the refinement set P'_r was found for x'. Suppose also that the cover for x' consists of a set of cubes B_j with corresponding partial cuts c^*_j. From each partial cut c^*_j new partial cuts are built using instances in P'_r:

$$\mathsf{c}^{*\prime}_j = \mathsf{c}^*_j \cup \{p'\}, \quad \text{where } (p' \in P'_r) \wedge (\forall p'' \in \mathsf{c}^*_j : p' \| p'')$$

and for each newly created partial cut $\mathsf{c}^{*\prime}_j$ its new cover approximation $C^*(\mathsf{c}^{*\prime})$ is found as before, using the cumulative states of local configurations of instances $t' \in \bullet p' : p' \in \mathsf{c}^{*\prime}_j$. Note, that this cover approximation will be a single cube. The newly created cover approximation for $\mathsf{c}^{*\prime}_j$ will only have those literals set to "- ", whose instances belong to the slice $\mathcal{S}$ and are concurrent to all places in $\mathsf{c}^{*\prime}_j$.

The refined cover $C^*_{new}(x')$ is obtained as:

$$C^*_{new}(x') = \sum_j \left(\sum_k C^*_j(\mathsf{c}^{*\prime});\ \forall p'_k \in P'_r \right);\ \forall \mathsf{c}^*_j$$

from the set of old partial cuts c^*_j.

The pseudo-code of the algorithm for cover refinement is shown in Figure 6.17.

```
proc Refine covers($C^*(x')$, $Sig$)
  Set $C^*_{new}(x') = \emptyset$
  Choose an offending signal $a_j$ from $Sig$
  Find refining set $P'_r$ for $x'$ w.r.t. $a_j$
  for each $\mathsf{c}^*_j$ used to find $C^*(x')$ do
    for each $p' \in P'_r$ do
      $\mathsf{c}^{*\prime} = \mathsf{c}^* \cup \{p'\}$
      Calculate $C^*(\mathsf{c}^{*\prime})$
      $C^*_{new}(x') = C^*_{new}(x') + C^*(\mathsf{c}^{*\prime})$
    end do
  end do
  return $C^*_{new}(x')$
end proc
```

Figure 6.17: Procedure for refining cover approximations.

Property 6.7.1 The cover approximation for x' obtained after an iteration using the refining set with respect to a_i adds a literal corresponding to a_i to each cube.

Proof: By definition the refining set P'_r contains place instances which are sequential to all instances $*a'_i$ belonging to $\mathcal{S}$. Furthermore, all places in P'_r are mutually non-concurrent and, thus, none of them can be concurrent to any $*a'_i$. Hence, no $*a'_i$ will be concurrent to all places of any newly constructed partial cut. Therefore, literal corresponding to a_i will be defined in all cover approximations for newly built partial cuts.

□

Proposition 6.7.1 The refinement procedure in Figure 6.17 terminates in a finite number of steps.

Proof: Since each step refines the value of at least one variable and the set of signals is finite, the refinement procedure will terminate after at most $|A| \times |P'_a|$ iterations.

□

Corollary 6.7.1 The cover derivation procedure for the Negative set approximation strategy (shown in Figure 6.13) terminates in a finite number of steps. □

Proposition 6.7.2 The fully refined (in the worst case) cover of x' in the STG-unfolding segment covers only states corresponding to the cuts which are encapsulated by $\mathcal{S}$ and covered by the exact cover for x'.

Proof: The approximated cover for x' represents partial cuts: those where input places of x', if x' is a transition, and those where x' is marked, if x' is a place. At each step the refinement procedure restores partial cuts of $\mathcal{S}$. Thus at the end of the refinement procedure all cuts are fully restored to the ordinary cuts. No transitions can be concurrent to all places in c. The cover approximation for each fully restored cut c will, therefore, be equal to the signal state of the configuration producing it. Hence, the fully refined cover will be obtained as the union of the binary vectors found for each cut in $\mathcal{S}$ and includes $\bullet x'$ or x' depending on whether x' is a transition or place instance, i.e. the cover will be the exact cover for x'.

□

Corollary 6.7.2 The fully refined cover for an excitation slice is equal to the exact cover for this slice. □

Corollary 6.7.3 The fully refined cover for an on- (off-) slice is equal to the exact cover for this slice. □

From the above corollaries it follows that Propositions 6.6.3 and 6.6.3 about the correctness of the Negative set cover derivation strategy hold. Moreover, the refinement procedure preserves the covering of the $\mathcal{P}$-set. Therefore, after each iteration step a newly obtained cover also covers $\mathcal{P}$-set. Hence, if after some iteration the new $\mathcal{P}$-set and $\mathcal{N}$-set covers have an empty intersection, then they will satisfy the implementation specific correctness criterion, set out in Definitions 6.2.1-6.2.5 for each implementation architecture.

Example. Consider a fragment of an STG-unfolding segment shown in Figure 6.18. Suppose that on-set cover approximation $\mathcal{C}^*_{On}$, found with the approximation set P'_a =

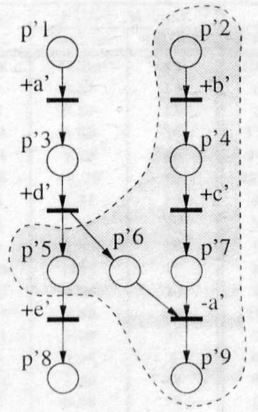


Figure 6.18: Illustration of cover approximation and refinement.

$\{p'_1, p'_3, p'_5, p'_8\}$, intersects with C^*_{Off} for some signal. Suppose also that some cube which is a cover approximation of place p'_5 causes this non-empty intersection. The set of offending signals is found as $Sig = \{a, b, c\}$. Let a be the signal chosen for refinement. Its only instance which should be used in refinement is $-a'$. A refinement set is chosen as $P'_r = \{p'_2, p'_4, p'_7, p'_9\}$. The new partial cuts are: $\mathsf{c}_1 = (p'_5, p'_2)$, $\mathsf{c}_1 = (p'_5, p'_4)$, $\mathsf{c}_1 = (p'_5, p'_7)$, $\mathsf{c}_1 = (p'_5, p'_9)$. Therefore, the new cover approximation for the place instance p'_5 is: $C^*_{new}(p'_5) = \{10010\} \cup \{11010\} \cup \{11110\} \cup \{01110\} = a\bar{c}d\bar{e} + bcd\bar{e}$. The resulting cover is an exact cover of for place p'_5. ■

6.8 Experimental Results

The approximation method using the Negative set approximation strategy was implemented on the basis of the STG verification tool "PUNT" which constructs an STG-unfolding segment. The Negative set approximation strategy requires On- and Off-set partitioning for each of the implementation architectures. One of the architectures, ACGpS, can be considered as an indicator of the method's performance. While testing the novel approach the following targets were aimed at:

- To demonstrate the practicality of the approach on a set of moderately sized examples by a comparison of the obtained implementation to those obtained from other existing tools.

- To illustrate the increased feasibility of the synthesis process using the suggested approach, by an investigation of the dependency of the time taken to synthesise a circuit from the size of specifications.

6.8.1 Practicality

To demonstrate the practicality of the approach a set of publicly available benchmarks was used. All STGs in this set of benchmarks have a low number of signals (max. 25). Thus the size of the state space is moderate and it is possible to synthesise these STGs with other existing tools. The synthesis results are presented in Table 6.1. The table presents the total

Benchmark	Sigs	PUNT ACGpS		Other tools		
		TotTim	LitCnt	Petrify	SIS	LitCnt
imec-master-read.csc	18	77.00	83	125.66	630.52	69
nowick.asn	7	0.97	17	1.44	0.51	20/17
nowick	6	0.57	15	1.10	0.23	14
par_4.csc	14	3.63	36	12.31	168.55	36
sis-master-read.csc	14	5.78	48	27.09	130.66	48
tsbmSIBRK	25	42.70	72	299.90	141.51	72
pn_stg_example	6	1.77	19	4.20	6.84	19
forever_ordered	8	1.46	20	5.24	.8.81	16
alloc-outbound	9	0.85	16	1.75	1.53	16
mp-forward-pkt	20	0.83	17	1.50	0.22	17
nak-pa	10	0.96	20	2.28	0.29	20
pe-send-ifc	17	2.53	68	19.50	1.16	75/72
ram-read-sbuf	11	1.08	25	3.28	0.26	22
rcv-setup	5	0.25	8	0.72	0.14	8
sbuf-ram-write	12	1.48	23	4.04	0.38	23
sbuf-read-ctl.old	8	0.86	15	1.29	0.19	15
sbuf-read-ctl	8	0.71	15	0.99	0.16	15
sbuf-send-ctl	8	0.88	19	1.95	0.21	19
sbuf-send-pkt2	9	0.99	19	2.16	0.23	19
sbuf-send-pkt2.yun	9	1.07	31	3.43	0.26	31
sendr-done	4	0.23	6	0.33	0.14	6
Total	228	146.78	592	520.16	1092.77	580/574

Table 6.1: Experimental results for circuit synthesis.

time spent (in seconds) on the synthesis of SI circuits from their STG specifications in the ACGPS architecture ("PUNT ACGpS"). On average, about 1% of this time was spent on building the STG-unfolding segment and about 15% was spent on Espresso minimisation. For comparison, the same set of benchmarks was synthesised using `petrify` and SIS [15, 82]. Their total timings are grouped in the column "Other tools".

The literal count was used for comparison of the implementations obtained by different tools (columns "LitCnt"). The literal count shows the total number of literals used in all cubes of logic functions implementing all signals. As it can be observed, the synthesis technique based on the STG-unfolding segment using the Negative set approximation strategy produces implementations comparable with those produced by other tools. The timing results show that the approximation technique compares favourably to `petrify` [15]. PUNT is also comparable with SIS on the benchmarks with a low count of signals and it becomes increasingly better with the growth of the signal count. These results show that for small sized benchmarks, the overheads of constructing and traversing the STG-unfolding segment may outweigh the time spent on constructing a small reachability graph with an efficient implementation. The slightly worse literal count for two benchmarks (`imec-master-read.csc` and `nowick`) is attributed to the fact that the DC-set is partitioned due to the pessimistic condition on the empty intersection between cover approximations in the Negative set approximation strategy.

6.8.2 Feasibility

The Muller pipeline benchmark was chosen to demonstrate the increased feasibility of the approximation method. The graph interpretation of the results is shown in Figure 6.19. As can be observed, existing tools cannot cope well with the growing size of the specification, either running out of memory or taking a prohibitively long time. Double exponential dependency of SIS and `petrify` is attributed to (1) exponential explosion of the state space, and (2) exact cover calculation methods. In addition, the new tool was used to synthesise a real life specification of the Counterflow pipeline specification [97] which has 34 signals. Of the existing

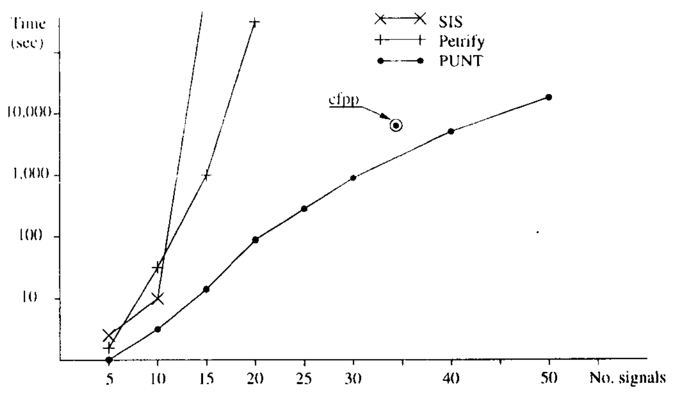


Figure 6.19: Experimental results for Muller pipeline.

tools, only `petrify` was able to synthesise it but it took more than 24 hours. At the same time PUNT was able to synthesise it in under 2 hours, thus giving an order of magnitude gain in speed. This result is shown on the graph as a circled dot.

6.9 Conclusions

This chapter introduced a methodology for the synthesis of SI circuits from the STG-unfolding segment. It presented a review of the major implementation architectures for SI circuits along with the conditions of implementability of an STG in a particular architecture. It introduced cuts and slices of the STG-unfolding segment, which were used to identify fragments of the STG-unfolding segment corresponding to sets of reachable states of the original STG.

A new synthesis procedure from the STG-unfolding segment was developed, first for exact, and then for approximated covers. The practicality and applicability of the suggested technique was demonstrated for the Negative set approximation cover strategy by a comparison with implementations obtained from the existing tools.

This work leaves open for future research the problems of developing better techniques for cover evaluation and detecting cuts that belong to the negative set in the Positive set evaluation strategy. Once the solution for these problems is found, the comparative study of two strategies can be performed to indicate which classes of STGs and implementation architectures benefit most from the application of different strategies.

Chapter 7

Applications of PN-unfolding

This chapter describes two new applications of the PN-unfolding segment. The first one is related to the problem of circuit analysis. The second, suggests a new use for PN-unfolding as a pre-processing mechanism for PN symbolic traversal.

PN-unfolding may suffer from an unnecessary exponential explosion when it is applied to the verification of PN models of asynchronous systems if the model contains bi-directional arcs. Sections 7.1–7.5 of this chapter propose the use of *contextual* nets instead, by presenting an algorithm for constructing a finite contextual net unfolding segment. This method is applied to the verification of an asynchronous control structure implementing a four-slot asynchronous communication mechanism, intended for use in real-time systems.

The rest of this chapter proposes the use of PN-unfolding for obtaining a good variable ordering for analysis methods based on PN-symbolic traversal. These recently proposed verification methods often yield better performance than using the conventional reachability graph analysis. They employ a Binary Decision Diagram (BDD) representation of the boolean functions characterising the state space of the model. PN-symbolic traversal may, however, suffer from bad ordering of BDD variables. The method suggested here combines two approaches for PN verification: PN-unfolding and BDD-based traversal. The results of unfolding construction are used for obtaining a better ordering of BDD variables thereby helping the PN symbolic traversal approach.

7.1 Motivation for Contextual Net Unfolding

Chapter 5 examined the analysis of four-phase asynchronous circuits by means of Circuit PNs. The circuit schematic is converted into a set of STG fragments composed together. A characteristic feature of STGs which fall into the category of Circuit PNs is the presence of a *contextual* dependence between places and transitions associated with different logic elements of the circuit. The intuitive meaning of such a dependence happens to be exactly the same as that of the *context flow-relation* in contextual nets [52]. However, the "standard" STG uses the "ordinary" PN model, and those contextual links are generally represented in terms of the usual flow-relation as *self-loop* arcs (often represented by bi-directional arcs). The application of the "ordinary" PN semantics to PNs and STGs with self-loops, as in all existing unfolding algorithms, often results in the same sort of exponential blow-up as with the use of RGs and SGs (see an example below).

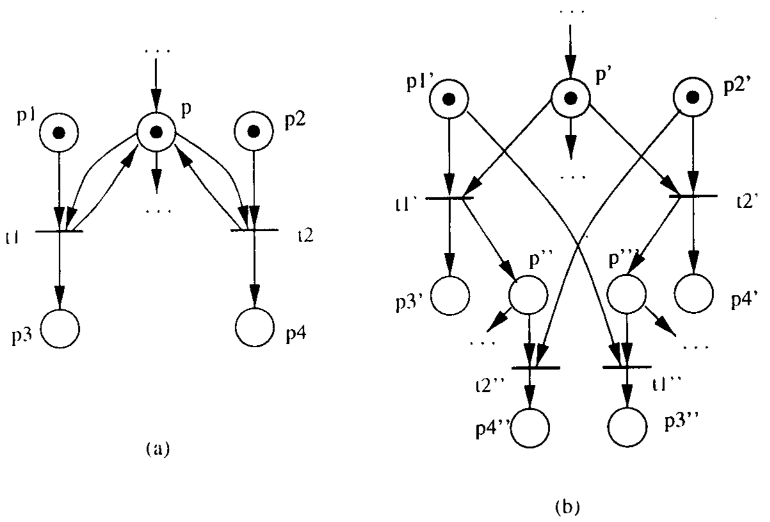


Figure 7.1: Example of a Petri net with self-loops (a) and its unfolding according to the requirement of structural conflict (b)

The root of this problem, informally, is in the different semantic interpretations applied to self-loops in ordinary PNs and in contextual nets. This appears to be critical, from the size point of view, for the partial order (or non-sequential) semantics [12]. Indeed, the PN-unfolding is a partial order technique which proves to be efficient for a large class of PN models – it can keep concurrency in its natural, "non-interleaving", form. Unfortunately, the classical partial order semantics, underlying the existing unfolding methods, loses it efficiency when the paradigms of concurrency and conflict are mixed up.

Consider a well-known example of a pair of transitions t_1 and t_2 sharing a single place p by means of a self-loop flow relation as shown in Figure 7.1.(a). The place p is thus both a predecessor and a successor to transitions t_1 and t_2. The classical partial order semantics does not allow those transitions to fire simultaneously, since the token in the shared place is treated as a *single resource* (structural conflict on p). It will unfold this net into two interleaving sequences (Figure 7.1.(b)), resulting in a combinatorial blow-up.

In asynchronous system modelling, transitions may also be connected by self-loops with shared places but these connections have a different meaning. Their meaning is not resource-based but rather *condition-based* – a transition, associated with a gate's output, can fire if a place, associated with another gate's output, is marked with a token. In that sense, the above example should be interpreted as the fact that two gates share the same input wire (the "high" or "low" state of the wire is modelled by place p). When the value is set to logical 1 (or 0) it enables both gates (the events on the gates' outputs are represented by transitions $t1$ and $t2$), and so the latter can fire completely independently of each other.

The new CN-unfolding algorithm suggested here allows explicit contextual arcs in the underlying PN. This algorithm may find its theoretical justification in the work on non-sequential semantics of contextual nets by Busi and Pinna [12]. The algorithm was applied to a number of characteristic examples of models with contextual arcs. Those examples included the verification of asynchronous circuits and asynchronous communication mechanism for coherence (checking *freedom from simultaneous access to data for read and write*). In the latter example, it was impossible to construct the truncated unfolding for an ordinary PN interpretation, due to the large number of self-loop arcs, while the contextual net unfolding appeared to be fairly compact. Note that it is often the case that even initial STG specifications of asynchronous systems (see the example in Section 7.4) may contain contextual arcs and therefore the use of the new unfolding algorithm will be productive. The use of contextual nets in the modelling of asynchronous systems substantially increases the efficiency of the unfolding (i.e. partial order) approach, and thus enhances the power of PNs as a modelling/verification tool in general.

7.2 Positive Contextual Nets

This section introduces the basic concepts required to develop the algorithm for a contextual net unfolding. For simplicity, a subclass of contextual nets in which only the so-called *positive* [12] context is allowed is considered here and the underlying PN is safe. This restriction is mainly for practical reasons since this class is sufficient to deal with net models of asynchronous systems, which would otherwise be represented by ordinary PNs with self-loops. For example, Circuit PNs, mentioned before, are a class of 1-safe nets with self-loops. Furthermore, one can imagine that it is possible to extend this technique to nets with *negative* context (on the basis of contextual interpretation of inhibitor arcs). This can be done by the (standard) transformation of a bounded inhibitor net to an ordinary PN with complementary places and self-loop arcs [65].

Further reading on contextual Petri nets can be found, e.g., in [52, 12].

7.2.1 Contextual Net Definition

Since contextual nets differ from ordinary PNs, several notions, e.g. transition enabling and firing, must be redefined. First, a contextual net is defined.

Definition 7.2.1

1. A **(positive) contextual net** (CN) is a tuple $\langle P, T, F, \mathsf{C}\rangle$, where $\langle P, T, F\rangle$ is a safe Petri net and $\mathsf{C} \subseteq P \times T$ is the **context relation** such that $\mathsf{C} \cap F = \emptyset$.

2. A **marked (positive) contextual net** is CN augmented with an initial marking M_0 of its places in P, i.e. it is a tuple $\langle P, T, F, \mathsf{C}, M_0\rangle$.

□

For a transition $t \in T$ a *context set* of t is defined as $\hat{t} = \{y \in P | (y, x) \in \mathsf{C}\}$. Places in the context set of t are called *contextual places*. In the graphical representation of CNs, the context relation will be depicted with a line without arrows.

A transition t is called *enabled* at marking M if $\bullet t \subseteq M$ and $\hat{t} \subseteq M$. Thus, for a transition to be enabled in a CN, in addition to all its predecessor places, all its contextual places must also be marked.

The firing of a transition t enabled at marking M is defined as an *instantaneous action* producing the new marking $M' = (M \setminus \bullet t) \cup t\bullet$. Such a marking M' is called *directly reachable* from marking M. Note that transition firing in CN does not affect tokens which mark contextual places. The notions of feasible (firing) sequence and reachability are defined for PNs on markings and transitions between them and, therefore, are directly applicable to CNs.

A CN is said to be *finite* if sets P and T are finite.

A transition t is said to be *disabled by* another transition t' of a CN, if there exist a marking M in which both t and t' are enabled and t is not enabled in the marking M' reached from M by firing t'. In this case t is said to be a *non-persistent* transition or CN is *non-persistent with respect to* t. Otherwise CN is called persistent with respect to t. Finally, a CN is called *persistent* if it is persistent with respect to all its transitions.

In order to be able to work with STG models of asynchronous circuits, a contextual STG is defined.

Definition 7.2.2 A **Contextual Signal Transition Graph** (CSTG) is a tuple $\langle P, T, F, \mathsf{C}, M_0, A, \Lambda\rangle$, where $\langle P, T, F, \mathsf{C}, M_0\rangle$ is a CN, A is a set of signals and $\Lambda : T \to (A \times \{+, -\})$ is the **signal labelling function**. □

7.2.2 Role of Conflict Relation

A pair of transitions t_1 and t_2 in a CN are said to be in *symmetric structural conflict* iff $\bullet t_1 \cap \bullet t_2 \neq \emptyset$. Transitions t_1 and t_2 are said to be in *asymmetric structural conflict* iff $\bullet t_1 \cap \bullet t_2 = \emptyset$ but $\bullet t_1 \cap \widehat{t_2} \neq \emptyset$. A pair of transitions t_1 and t_2 are said to be in *symmetric dynamic conflict* (*asymmetric dynamic conflict*) if they are in *symmetric structural conflict* (*asymmetric structural conflict*) and there exist a marking in which they are both enabled.

Note that the definition of dynamic conflict is fairly strong (it is brought to match to the notion of conflict in the unfolding see below).

The conventional "PN-based" definition of reachability graph defines the so-called *interleaving semantics* of a CN. This type of semantics cannot for example distinguish between two behaviours of the following kind. In the first case, a pair of transitions can fire independently (in either order or simultaneously). In the second case, two transitions are in symmetric dynamic conflict where they share a predecessor place with one token. They cannot fire simultaneously; however they are allowed to fire one after the other in either order. It is possible to apply a more refined type of semantics, called *step semantics* [32], which can differentiate between those behaviours. In this case, a transition step, consisting of *both* transitions, is allowed which will be explicitly represented in the *step reachability graph*. The step semantics of CNs is defined in [52, 12].

Example. Figure 7.2.(a) shows an example of a PN with a self-loop flow relation between place p and transitions t_1 and t_2. Its reachability graph and step reachability graph are the same – they are shown in part (b) of the figure. For a "similar" contextual net, shown in Figure 7.2.(c), where the relation between p and t_1 and t_2 is contextual, the step reachability graph is different – it is depicted in part (d). Note that the same step reachability graph would be produced by the above CN (or PN) if place p and its context edges are deleted. ■

Note that considering steps of two or more *non-conflicting* transitions, in addition to simple, single-transition steps does not create new markings in the reachability set of a CN.

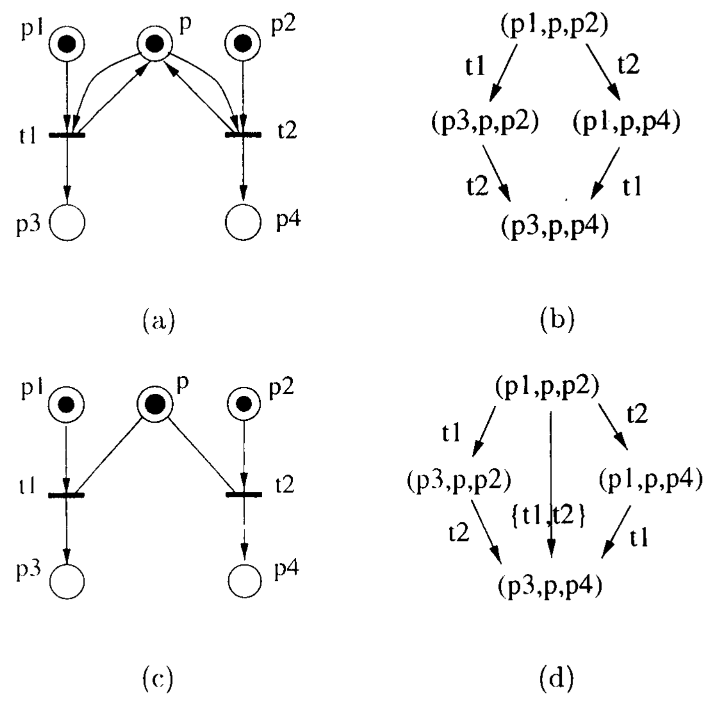


Figure 7.2: Petri net with self-loops (a) with its step reachability graph (b) and contextual net (c) with its step reachability graph (d)

It produces only new firing sequences, called *step sequences*. As has been noted, the partial order models, such as net unfoldings, are sensitive to such semantical distinctions as well, but their representation reflects the above distinction in the structure of the semantic model. For example, the presence of dynamic conflicts in the original CN generates choice between alternative branches of the net unfolding, which consists of the instances (or occurrences) of places and transitions taken from the original CN.

It is obvious that the necessary condition for a disabling to occur (and hence non-persistency) between transitions is the presence of dynamic conflicts between them. The opposite is not true – indeed, in the CN shown in Figure 7.2.(a) transitions t_1 and t_2 are in dynamic conflict but they do not disable each other. The property of transition persistence (or non-persistence) is crucial in the analysis of asynchronous circuit behaviour. It corresponds to the behavioural quality of a circuit to be either free from hazards on a given signal or not. In the use of PNs for the analysis of hazard-freedom it is convenient to identify hazards with some local properties of the semantical model, such as conflicts – the latter are easier to check than those properties which are based on the global information such as markings and their reachability. Therefore the cases like the one shown in Figure 7.2.(a) appear to be difficult for analysis because from the circuit semantics such behaviour can be correct (the signal transition corresponding to say t_1 is not disabled by firing t_2) while its analysis through the notion of dynamic conflicts raises an alarm[1]. CNs distinguish the cases of contextual dependence from those which are real dynamic conflicts and imply hazardous behaviour in the circuit terms. The CN-unfolding, introduced later, supports this distinction. It fills the

[1]This alarm can certainly be checked and identified as false by means of explicit analysis for disabling – but this would affect the overall efficiency of the analysis process and hence it is preferable to avoid it.

semantic gap between the notion of a dynamic conflict in the net and that of nón-persistence, the gap characteristic for ordinary PNs.

7.3 Contextual Net Unfolding

7.3.1 Basic Relations in Acyclic Contextual Nets

Similar to PN-unfolding, CN-unfolding is defined using the notion of a labelled occurrence net. In order to define the corresponding notions for contextual nets, several preliminary definitions are required. These will impose additional conditions on the structure of a (P,T) labelled positive occurrence CN when it is augmented with the context relation between places and transitions. Those conditions must ensure that every transition in the occurrence CN can be fired once and only once if all its minimal places are initially marked.

Consider a (P,T) labelled positive CN $CN = \langle P',T',F',\mathsf{C}',L'\rangle$, where $\langle P',T',F',L'\rangle$ is a (P,T) labelled occurrence net and $\mathsf{C}' \subseteq P' \times T'$ is a context relation, satisfying $\mathsf{C}' \cap F' = \emptyset$. The following relations in this net are defined.

Definition 7.3.1 (Weak and Strong Precedence)

1. Two elements x' and x'' of CN are said to be in the **weak precedence relation**, denoted $x' \prec x''$, iff $(x',x'') \in (F' \cup \mathsf{C}')^*$.

2. Two elements x' and x'' of CN are said to be in the **strong precedence relation**, denoted $x' \prec\!\!\prec x''$, iff $\exists t : (x',t) \in F'^* : t \prec x''$.

□

It is clear from these definitions that:

- $\prec\!\!\prec \subseteq \prec$

- if the first element in a pair of elements $(x',x'') \in \prec$ of CN is a transition (i.e. $x' \in T'$), then $(x',x'') \in \prec\!\!\prec$. This is a simple corollary of the definition of C', which is defined on $P' \times T'$.

Note that the set-theoretic difference between the weak and strong precedence is a relation which reflects the specific nature of acyclic CNs. If a place p is in such a relation with another element x, this means that p must be marked with a token before x is either marked (if $x \in P'$) or fired (if $x \in T'$). However, this does not state that when x is marked or fired, the token must have been removed from p.

The following definitions of conflict and semi-conflict are also required.

Definition 7.3.2 (Conflict and Semi-conflict)

1. Two elements x' and x'' of CN are said to be in the **conflict relation**, denoted $x'\#x''$, iff there exist two distinct transitions t' and t'' such that $\bullet t' \cap \bullet t'' \neq \emptyset$ and $t' \prec x'$ and $t'' \prec x''$.

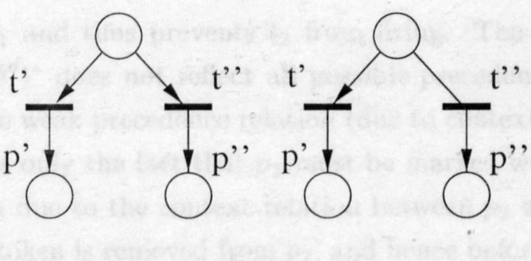


Figure 7.3: Conflict and semi-conflict.

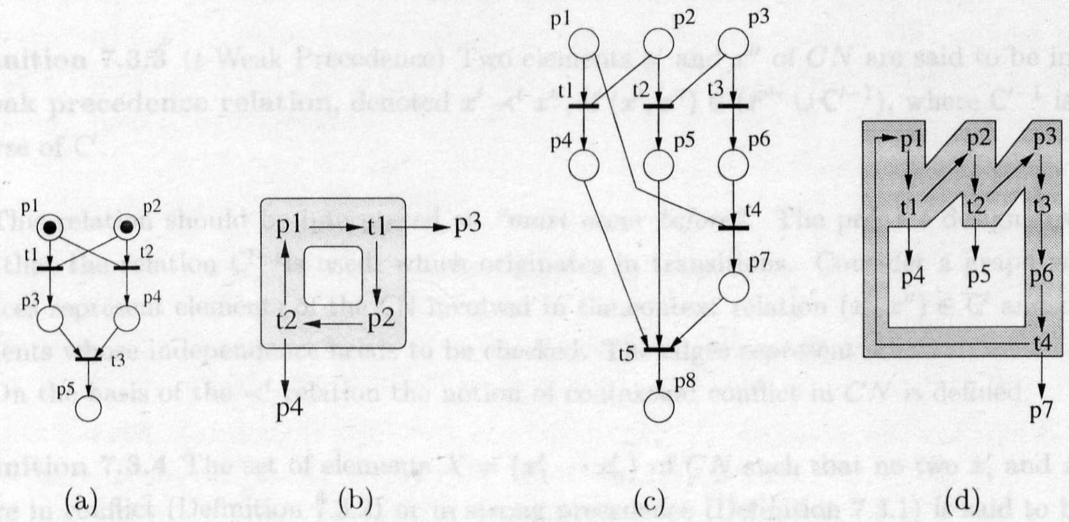


Figure 7.4: Examples of CNs (a),(c) which cannot be contextual occurrence nets and their corresponding graphs (b),(d) of *must occur before* relation.

2. Two elements x' and x'' of CN are said to be in the **semi-conflict relation**, denoted $x'\vec{\#}x''$, iff there exist two distinct transitions t' and t'' such that $\bullet t' \cap \hat{t''} \neq \emptyset$ and $t' \prec x'$ and $t'' \prec x''$.

□

The difference between the two conflict relations is illustrated in Figure 7.3. In the left part of the figure t' and t'' are in conflict, while in the right part in semi-conflict. The semi-conflict relation in CN corresponds to the asymmetric conflict in PNs discussed earlier. The firing of t' can disable t''. In this case, p' and p'' will not be marked together. However, if t'' manages to fire *before* t', then both p' and p'' can have tokens in the same reachable marking. Obviously, if p' and p'' are in conflict, there is no reachable marking in which they can both have tokens.

Example. Consider the CN shown in Figure 7.4(a). This example satisfies the conditions of Definition 3.3.1, applied to the CN taken above. At the same time this CN is not a valid contextual occurrence net since its transition t_3 cannot be fired. This is despite the fact that t_3 satisfies the "no self-conflict" condition of Definition 3.3.1. ■

Indeed, in order to fire transition t_3 both p_3 and p_4 must be marked in the same marking. In order to mark p_3, t_1 must fire before the token is removed from $p_2 \in \bullet t_2$. However, firing

t_1 removes token from p_1 and thus prevents t_2 from firing. The weak precedence relation defined as the set $(\mathsf{C}' \cup F')^*$ does not reflect all possible precedences that are expressed by the CN. For example, the weak precedence relation (due to context relation) between p_2 and t_1 in Figure 7.4(a) reflects only the fact that p_2 must be marked with a token before t_1 fires. On the other hand, again due to the context relation between p_2 and t_1, it can be said that t_1 must occur before the token is removed from p_2, and hence before the occurrence of t_2, the successor of p_2. This new type of precedence combined with the transitive closure of F' gives rise to a new relation which will be important for the definition of the independence relation.

Definition 7.3.3 (t-Weak Precedence) Two elements x' and x'' of CN are said to be in the **t-weak precedence relation**, denoted $x' \prec^t x''$, if $(x', x'') \in (F'^* \cup \mathsf{C}'^{-1})$, where C'^{-1} is the inverse of C'. □

This relation should be interpreted as *"must occur before"*. The prefix t designates the fact that the relation $\mathsf{C}'^{-1}$is used, which originates in transitions. Consider a graph where vertices represent elements of the CN involved in the context relation $(x', x'') \in \mathsf{C}'$ and those elements whose independence needs to be checked. The edges represent $\prec^t$.

On the basis of the $\prec^t$ relation the notion of contextual conflict in CN is defined.

Definition 7.3.4 The set of elements $X = \{x'_1 \cdots x'_n\}$ of CN such that no two x'_i and x'_j in X are in conflict (Definition 7.3.2) or in strong precedence (Definition 7.3.1) is said to be in **contextual conflict**, denoted $\{x'_1 \cdots x'_n\} \in \widehat{\#}$, iff there exist two distinct transitions t' and t'' such that $(t', t''), (t'', t') \in (\prec^t)^*|_Y$, where $Y = \{y'_1 \dots y'_k\}$ is the set of elements that belong to the backward transitive closure (with respect to $F' \cup \mathsf{C}'$) from X, and $(\prec^t)^*|_Y$ is the transitive closure of $\prec^t$ with the domain restricted to Y. □

That is, a set of elements X is in contextual conflict if there exists a cyclic path in the graph of the $\prec^t$ relation such that this path goes through the elements that precede the elements in X. Observe that unlike the acyclic PN case this relation is defined for a set of elements and is not generalisable from a pairwise relation, i.e. in an acyclic CN the absence of pairwise contextual conflicts in X does not imply that X is free from contextual conflicts.

Example. The graph showing $(\prec^t)^*|_Y$ relation for the example in Figure 7.4(a) is shown in Figure 7.4(b). As can be observed, there exists a loop in this graph which indicates that it is impossible to fire transitions belonging to this loop due to the contextual dependence between them.

The example in Figure 7.4(c) (similar to discussed in [94]) illustrates that the contextual conflict relation is not generalisable. Indeed, from the graph in Figure 7.4(d) it can be seen that there is no contextual conflict between any pair of places p_4, p_5, p_7, however, all three places are in contextual conflict (and hence cannot ever be marked together). ■

Thus the independence relation is introduced as:

Definition 7.3.5 A set of elements $X = \{x'_1 \cdots x'_n\}$ of CN is said to be in the **independence relation**, denoted $\{x'_1 \cdots x'_n\} \in \|$, iff the following three conditions are satisfied:

1. no two instances x'_i and x'_j are in conflict $((x'_i, x'_j) \notin \#)$;

2. no two instances x'_i and x'_j are in strong precedence with each other $(x'_i \prec\!\!\prec x'_j)$;

3. the elements of X are not in contextual conflict $(\{x'_1 \cdots x'_n\} \notin \widehat{\#})$.

□

The first two conditions are trivial extension of those used in defining independence in occurrence nets. The third condition shows that the elements of X may be independent even if there exists a semi-conflict between them but no a contextual conflict. For example, places p' and p'' are in semi-conflict in Figure 7.3 but are also independent. Therefore these two places can be marked together.

7.3.2 Contextual Net Unfolding

It is now possible to define formally a contextual occurrence net.

Definition 7.3.6 A (P,T) **labelled (positive) contextual occurrence net** is a CN $CN = \langle P', T', F', \mathsf{C}', L' \rangle$, where $\langle P', T', F', L' \rangle$ is a (P,T) labelled occurrence net, $\mathsf{C}' \subseteq P' \times T'$ is a context relation, satisfying $\mathsf{C}' \cap F' = \emptyset$, and for any $t \in T' : \forall p_1, p_2 \in \bullet t \cup \widehat{t} : p_1 \| p_2$ (i.e. all preceding and context places for every transition are independent). □

The definition of a contextual occurrence net is different from the one in [12] (Definition 6). It is, however, possible to prove that the conditions defined in items 3 to 5 of Definition 6 in [12] follow from the conditions based on the independence relation in Definition 7.3.6.

The occurrence net derived by unfolding a CN is defined as follows (in a similar way as the PN-unfolding in Chapter 3).

Definition 7.3.7 If $\langle P, T, F, \mathsf{C}, M_o \rangle$ is a marked CN, then the CN-**unfolding** of this net is the *maximal* (P,T) labelled contextual occurrence net (up to isomorphism) $\langle P', T', F', \mathsf{C}', L' \rangle$, satisfying the following properties:

- $\forall t' \in T' : L'$ restricted to $\bullet t'$ and $\bullet L'(t')$ is a bijection, L' restricted to $t'\bullet$ and $L'(t')\bullet$ is a bijection and L' restricted to $\widehat{t'}$ and $\widehat{L'(t')}$ is a bijection;

- L' restricted to the set P'_{min} of minimal places in P' is a bijection between P'_{min} and M.

□

The notion of independence relation plays a critical part in the algorithm of the unfolding as it allows one to determine if a particular transition is instantiated or not. A new instance t' is constructed when there exists a set of independent instances of places in $(\bullet t) \cup \widehat{t}$. Similar to the PN-unfolding, the following "cornerstones" are defined:

- A *configuration* C of a CN-unfolding is defined as a transitively backwards closed (with respect to $F' \cup \mathsf{C}'$) set of transitions (i.e. if $t' \prec t''$, then $t'' \in C$ implies $t' \in C$) such that no two instances t' and t'' in C are in conflict or contextual conflict. A local configuration of t', denoted as $\lceil t' \rceil$, is a minimal configuration including t'.

- The *post-set of configuration* C, denoted as $C\bullet$, is the set of places which are successors but not predecessors of the transition instances in C. The post-set consists of instances of places reachable after firing of all transitions in C.
- The *final state of* C, denoted as $F_s(C)$, is the set of places which is obtained by mapping the instances in $C\bullet$ onto places of the original CN.

A local configuration $\lceil t' \rceil$ also has a post-set and a final state. Note that places in $\widehat{t'}$ are not consumed by t' and, therefore, instances in $\widehat{t'}$ are always included into the post-set of its local configuration.

In a PN-unfolding, if a configuration C_1 is a subset of another configuration C_2, then the $F_s(C_2)$ is always reachable from $F_s(C_1)$ by firing all transitions whose instances are in $C_2 \backslash C_1$. Firing all such transitions is always possible because, by the definition of configuration of a PN-unfolding [46], all their instances are not in conflict with any transitions in $C2$ and hence, if enabled cannot be disabled.

In a CN-unfolding, it is possible that transition instances that are in semi-conflict belong to one configuration. Let the above-mentioned C_2 be such a configuration and let $t_2 \in C_2$ and $t_1 \in C_1$ be in semi-conflict $t_2 \vec{\#} t_1$. Then transition t_1 may be fired before t_2, and thus disable t_2. At the same time, if t_2 fires before t_1, the latter is not disabled, and thus configuration C_2 becomes valid. There always exists at least one firing sequence which includes all transitions of any configuration in the contextual CN-unfolding.

7.3.3 Contextual Net Unfolding Segment

The CN-unfolding can be infinite. To obtain a finite fragment of CN-unfolding (similar to a truncated unfolding) a cutoff condition is introduced.

Definition 7.3.8 A transition instance t'_c is called a *cutoff point* if there exits another instance t' such that :

- $|\lceil t' \rceil| < |\lceil t'_c \rceil|$, and
- $F_s(\lceil t' \rceil) = F_s(\lceil t'_c \rceil)$.

□

Note that the cutoff definition imposes the same conditions on the newly generated instance t' as in [46]. The choice of these conditions may introduce redundancy into the unfolding fragment, i.e. some transitions will be added although their successors do not add any information about the CN behaviour. However, for the sake of simplicity, this redundancy is ignored here. The CN-unfolding algorithm in given in Figure 7.5.

The difference between this algorithm and the truncated PN-unfolding algorithm is in the line marked $**$. Instead of simply finding a set of independent instances for places in $\bullet t$, the new algorithm needs to find a set of instances for $\bullet t \cup \widehat{t}$ which are independent according to Definition 7.3.5.

Example. Consider an example in Figure 7.6. In order to instantiate t'_8 the independence of p'_8 and p'_9 is checked, and a graph of context relations is constructed as shown in

```
proc Build segment($N = \langle P, T, F, C, M_0 \rangle$)
  Initialise $N'$ with instances of places in $M_0$
  Initialise QUEUE with $t$ enabled at $M_0$
  while QUEUE not empty do
    Pull $t$ from QUEUE
    Add $t'$ and $t'\bullet$ to $N'$
    if $t'$ is a cutoff then do
      Mark $t'$ and $t'\bullet$ as cutoff points
    end do
    for each $t$ in $T$ do
**      Find unused set of CN independent instances of places in $\bullet t \cup \hat{t}$
      if such set exists then do
        Add $t$ to QUEUE in order if its $|[t']|$
      end do
    end do
  end do
  return $N'$
end proc
```

Figure 7.5: Algorithm for obtaining CN-unfolding segment.

Figure 7.6. As can be observed, these places are not independent due to a contextual conflict between them. Hence t'_8 will not be instantiated. ■

The work of Vogler *et. al.* [95] presents a rigorous proof of the completeness of the CN-unfolding segment for read-persistent subclass of contextual nets, i.e. contextual nets where $\bullet t_1 \cap \bullet t_2 \neq \emptyset$ for all $t_1, t_2 \in T$ in conflict under some reachable marking. This work, however, highlights that this simple cutoff condition cannot be applied to all contextual nets. It presents an example of a contextual net with an acyclic contextual fragment which is not synchronised later with any future behaviours of the CN. In this particular case, the CN-unfolding segment fails to represent all reachable markings. Nevertheless, the CN-unfolding servers as a powerful experimental verification vehicle for this work. The examples illustrating its application are cyclic and demonstrate the great potential of the CN-unfolding segment.

The CN-unfolding can be extended to CSTG unfolding in a similar way as the PN-unfolding was extended to STG-unfolding. The CSTG-unfolding keeps track of the signal interpretation of transitions and terminates when all states of the SG are represented in the truncation.

7.4 Application: Modelling a Communication Mechanism

A good example for the application of the CSTG-unfolding technique can be found in [84]. This paper describes an asynchronous communication mechanism, which is used in designing real-time systems for safety-critical applications. The mechanism is defined at the high abstraction level as a concurrent system consisting of a reader process, a writer process and a four-slot

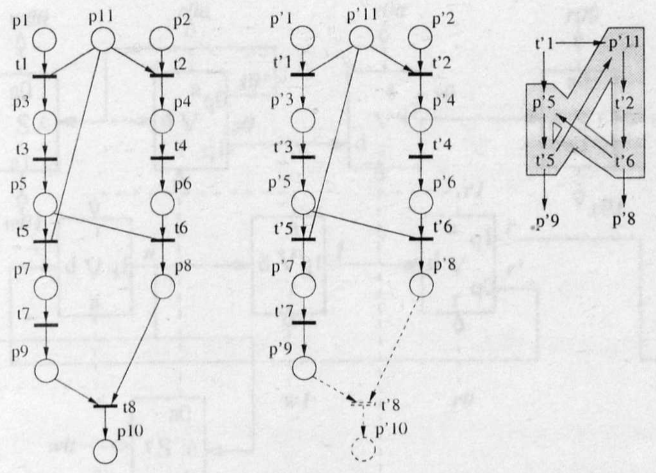


Figure 7.6: Another example of a CN and its unfolding.

data array and a set of shared control variables:

Writing		*Reading*	
$wr:$	$d[n, \overline{s[n]}] := input$	$r0:$	$r := l$
$w0:$	$s[n] := \overline{s[n]}$	$r1:$	$v := s$
$w1:$	$l := n \| n := \overline{r}$	$rd:$	$output := d[r, v[r]]$

The four slot data array is denoted by variable $d[i, j]$, which has a two-dimensional structure. The first subscript i of d defines one of the two pairs of slots (numbered as pair 0 and pair 1), while the second subscript j defines one of the two slots in each pair (numbered as slot 0 and slot 1). The values of subscripts are obtained through the auxiliary shared control variables. Variables n (for "next"), l (for "latest") and r (for "reading") are binary. The variables $s[i]$ and $v[i]$ are single dimensional arrays, each consisting of two elements; these arrays give pointers to the slots which are currently being written and read, respectively.

The primary aim of this communication mechanism is to organise the process of writing to and reading from data slots by the writer and reader in such a way that:

- (1) they can proceed from one operation to another, cyclically executing their respective sequences $wr, w0, w1$ and $r0, r1, rd$, in their own pace - that is, *completely independently* of each other; and
- (2) they never "clash" on writing to and reading from the same slot.

The author of the mechanism claimed (and proved in a fairly laborious way [85], analogous to manual reachability analysis) that the system satisfies the above stated requirements. For the first condition, he assumed that the writer may write data into a slot which has been written to but those data have not been read by the reader. Similarly, the reader is allowed to read from the same slot twice or more without new data being written into the slot. With this assumption[2] both the writer and the reader may access data slots directly, without checking any sort of semaphore or other blocking (mutual exclusion) mechanisms. This would guarantee

[2]In this work Simpson's "freshness" conditions [85] are not considered.

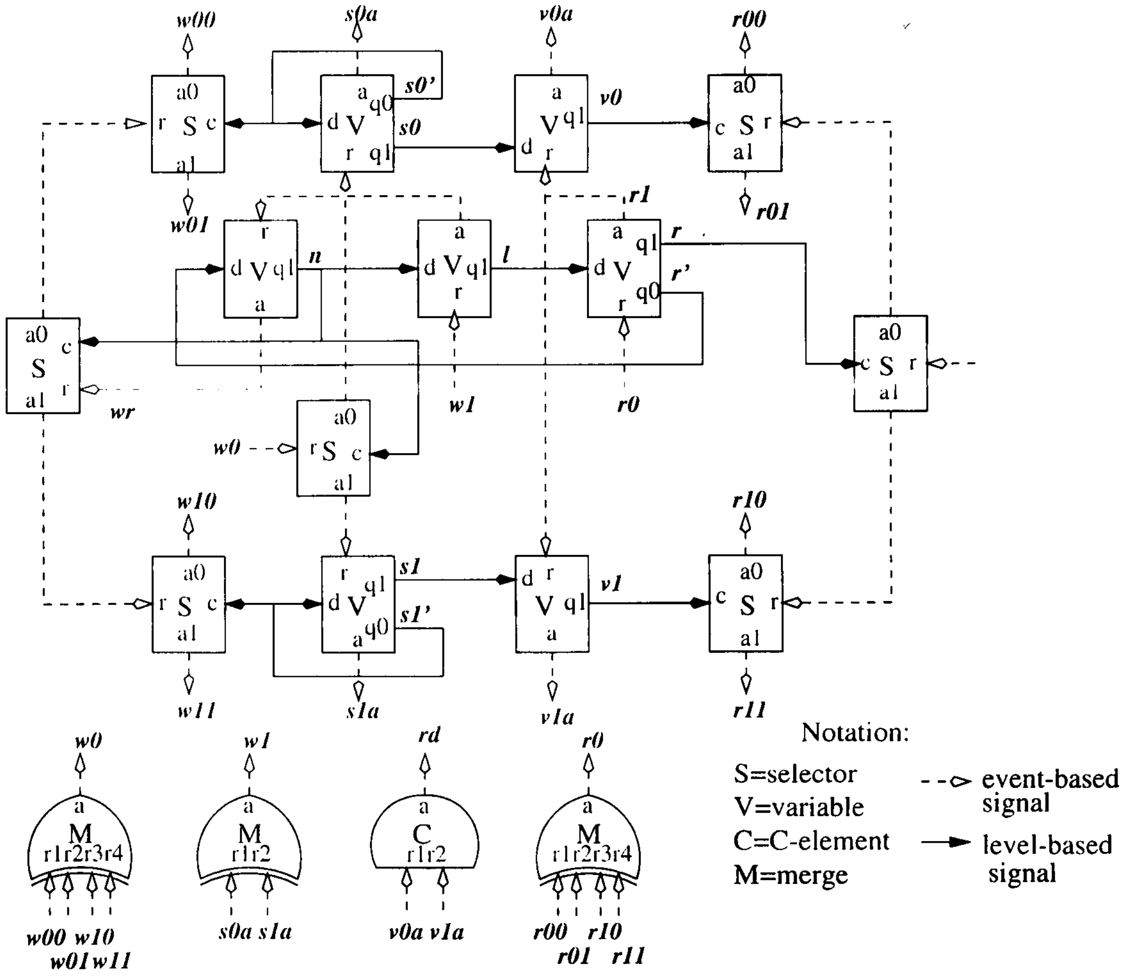


Figure 7.7: Schematic for Simpson's communication mechanism.

that both processes can act in a *truly* asynchronous and mutually independent fashion. This is critical from the viewpoint of *real-time* applications.

The second requirement is important from the *functional correctness* point of view. Any reading or writing operation applied to a slot of data is not atomic because the structure of data can be complex. Hence the reading and writing operations on the same slot may not interleave in time, i.e. every access operation must be separated from the next by an operation of some other sort. Thus the system must provide mutual exclusion only at the level of data slots, without using such methods on control variables.

The repetitive access of the writer and the reader to the pairs and the slots with those pairs is controlled entirely by means of the shared variables, which can be implemented in hardware as bistables (i.e. flip-flops). The order in which these variables are assigned new values and probed by the processes is outlined in the above procedures. Note that the notation || in operation wl stands for parallel execution of its left and right hand side subaction.

Assume that the initial state of all binary variables n, r, l is 0 and that $s = v = (0, 0)$. The writing process is repeated as follows. The writer process first puts data into the data slot pointed to by the current value of n (next pair to write) and the inverse of the $s[n]$. Then the writer toggles $s[n]$ to its inverse state. Finally, the writer assigns l (latest) to the value of n (next) and at the same time puts into n (next) the inverse value of r (reading).

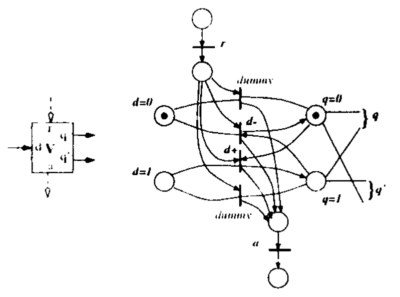


Figure 7.8: Illustration of the Variable element.

The reading process begins with the assignment of r (pair for reading) to l (latest pair to have been written to). Then the reader copies the value of the writer's slot pointer s to its slot pointer v. Finally, the reader reads a value from the data slot pointed by its current value of r and $v[r]$, after which the reading process is repeated.

The above control mechanism was implemented [84] in the form of a hardware structure which can be modelled as an ACS shown in Figure 7.7. The nodes of this structure are components of four main types: "Selector", "C-element", "Merge" and "Variable". The former three elements were described previously in Chapter 4. The latter, called **Variable** (also called Flip-flop), is a node which stores a level-based value on its output q (its complement q' is also available if needed). This module can copy the value of its level-based input d into q upon the arrival of a Request on the event-based input r. The result of copying is acknowledged on the event-based output a. If the previous value of q matched the new value of d, then the node retains its value, responding only with its acknowledgement on a. This behaviour is reflected in the CSTG model of Variable shown in Figure 7.8.

The system has both event-based (shown with dashed lines) and level-based (shown with solid lines) signals. Note that the system was slightly simplified in this implementation by serialising the parts of the $w1$ operation – first, $l := n$, and then $n := \overline{r}$. This does not affect the basic idea of independent interaction between the writer and the reader.

The CN for the whole system is constructed by composing the fragments for each element together as it was described earlier. Transitions of the resulting CN are labelled with actions corresponding to the switching signals. The actual resulting net is not shown here due to its fairly large size. This net has fourteen places with contextual arcs from them – those places correspond to seven level-based signals (variables $n, l, r, s0, s1, v0$ and $v1$.)

Due to the relatively large number of contextual arcs and a high degree of concurrency in the CN, its corresponding ordinary PN model produces a very large size unfolding segment. It was impossible to build such a segment due to memory blow-up[3]. However, the application of the new CN-unfolding algorithm allowed building the segment (up to cut-off transition instances) for the CN, and its size is rather modest.

In the process of building the unfolding both of the above correctness conditions were verified. The first condition, mutual independence and non-blocking, was checked in terms of persistence of transitions of the CN (e.g., the act of disabling a transition involved in the writing process by a transition involved in the reading process and vice versa). It was found that the unfolding was free from ordinary conflicts. The only remaining source of non-

[3]CN-unfolding was built on a SPARC4.

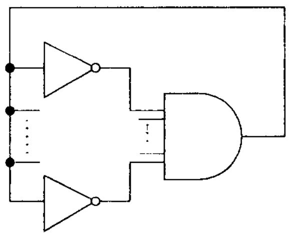


Figure 7.9: Illustration of the benchmark circuit.

persistence is due to the presence of semi-conflicts. For example, when the reader changes the value of variable r in operation $r0$, the writer may concurrently 'listen to' the value of r in its operation $w1$. Such a listening is realised via the contextual arcs between the places $r = 0$ and $r = 1$ and the transitions in its fragment corresponding to variable n (transitions labelled $n+$, $n-$ and two dummies – see Figure 7.8). It is thus possible to imagine that when the writer enables the transition labelled $n+$ (under the state $n = 0$ and $r = 1$), the reader switches r from 1 to 0, and thus disables $n+$. But this disabling of $n+$ will always be simultaneous with the enabling of a dummy transition in the same CN fragment (for n), and thus the progress of the writer is not affected. The CN-unfolding verification showed that **only** such types of non-persistence are possible, thus proving that the non-blocking (real-time) requirement is satisfied.

Consider the "no-clash" condition. This condition, in terms of the properties of the CSTG model can be stated as follows:

- for all four pairs of the read and write signals $(w00, r00), (w01, r01), (w10, r10)$ and $(w11, r11)$, referring to the same data slot, it is not possible to reach a marking in the CSTG in which a corresponding pair of transitions is simultaneously enabled and is not in conflict.

In terms of the unfolding, it is sufficient to verify whether any pair of transition instances in the CN-unfolding segment which map onto their transitions in the CN in the above mentioned pairs, belong to the independence relation. The result of the check proved that the mechanism satisfies its "no clash" condition.

7.5 Circuit Verification Results

The CN-unfolding method described earlier was implemented within the PN/STG analysis tool "PUNT" which uses the PN-unfolding approach. An enhanced cutoff condition, described previously in Chapter 4, was used which allows the avoidance of possible redundancy in the McMillan's unfolding and the construction of a PN-unfolding segment for larger PNs.

The CN model for Simpson's communication mechanism had 47 places, 56 transitions, 142 ordinary arcs and 44 contextual arcs, which in an ordinary PN would correspond to a total of 230 ordinary arcs. The constructed CN-unfolding segment had 1299 transition instances and showed that there are no independent instances of transitions for read and write operations on the same data slot. The computer ran out of memory trying to build the PN-unfolding segment for the ordinary PN model when the size exceeded 3000 transition instances.

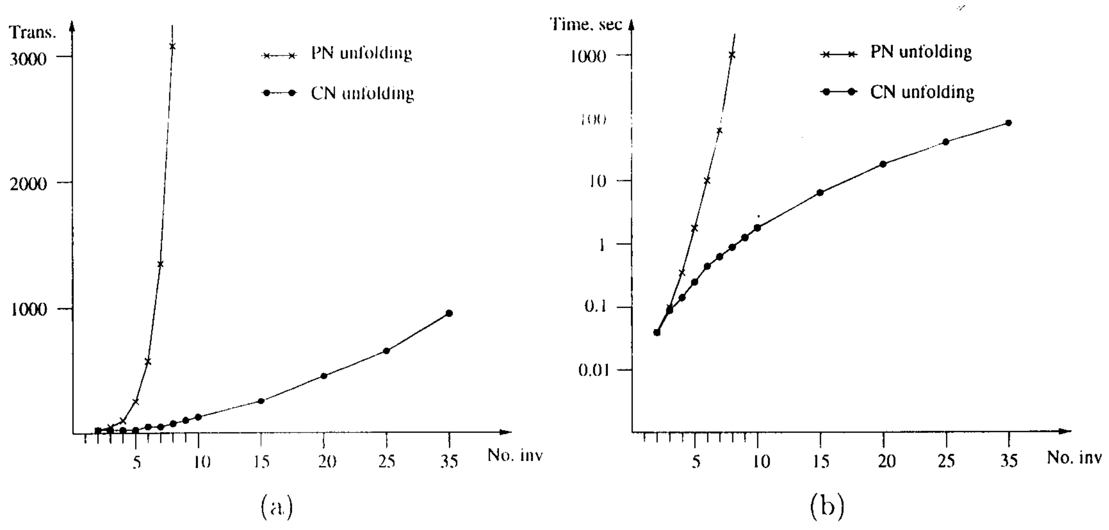


Figure 7.10: Experimental results: (a) Size of the segment in transition instances; (b) Time taken.

To illustrate the performance of the CN-unfolding approach a simple scalable example was chosen, consisting of an AND gate and inverters in the feedback loop of the AND gate. The circuit is shown in Figure 7.9. The number of inverters can vary, making this benchmark easily scalable. This circuit is known to have a hazard which exhibits itself as non-persistency of the output of the AND gate. This, of course, can be detected during the construction of the CN-unfolding and the construction can be terminated with an error report. Such a termination can only be made due to the hazard-freedom requirements of asynchronous circuits. However, examples such as the communication mechanism, described in the previous section, may require that the unfolding segment is built for the PN (or CN) even though some signals are non-persistent. Therefore, for illustrative purposes, the whole PN- (or CN-) unfolding segment was built.

The results are presented in Figure 7.10. The graph in Figure 7.10(a) illustrates the growth in size of the PN and CN-unfolding segments with the increase of the number of inverters. The size of the unfolding includes cutoff transitions. The graph in Figure 7.10(b) shows the time spent on the construction of the segment. This time was spent on the unfolding of the STG rather than the PN, i.e. the algorithm kept track of the binary states of the STG. The superiority of the CN-unfolding for this benchmark is obvious. For comparison, the original McMillan's algorithm builds a truncated PN-unfolding with 913 transition instances in 13.15 sec. for the benchmark with only 4 inverters.

The results demonstrate that the size of the PN-unfolding segment and the time it takes to construct it explodes double-exponentially. The first exponential dependency is caused by the exponential growth in the number of transition instances due to the exponential number of interleavings. The second dependency is caused by the combinatorial complexity during the selection of a set of independent instances for places in $\bullet t$. When a new instance of t is generated, the algorithm needs to find a set of independent instances of places in $\bullet t$, i.e. it may try all combinations in the worst case. The number of place instances is large due to the exponential number of transitions. Therefore, this selection process greatly affects the performance of the PN-unfolding construction algorithm.

Notably, the original McMillan's algorithm becomes triple-exponential for this set of benchmarks. The third exponent is caused by the redundant instances of transitions which cannot be made cutoffs due to the overly strong cutoff condition.

In the CN-unfolding the exponential explosion attributed to interleavings is avoided. Since this drastically reduces the number of place instances, the effect of the independent instances selection process is reduced to a negligible value. Thus the performance of the CN-unfolding algorithm on this set of benchmarks is close to polynomial. It was also observed that using the McMillan's cutoff condition yields a CN-unfolding of the same size as the one obtained using the enhanced cutoff condition. Due to the benefits of exploring the contextual dependency the algorithm did not generate redundant instances of transitions which were causing the blow-up.

7.6 Use of Unfolding for Variable Ordering

As it was discussed earlier, the analysis of PN models for state-based properties, such as deadlocks, is hard on the PN-unfolding segment. It was also noted in [47] that if a PN contains a deadlock, then the branch-and-bound algorithm [47] detects this deadlock in a reasonable time. However, if a PN is deadlock free, then the deadlock detection algorithm may need to examine all final configurations which is a time consuming operation. Recent methods [61, 36], based on Binary Decision Diagrams (BDDs), have demonstrated that they are capable of handling large state spaces at a relatively low cost. This feature has made this method attractive and its application is currently being investigated.

Despite being powerful, BDD-based verification techniques may suffer from the problem of bad ordering of the BDD variables. Using a proper variable ordering can yield significant reduction in the size of BDD, which in the worst case can be exponential to the number of variables. Usually, it is assumed that the designer, using additional knowledge about the system, can provide proper variable ordering. This obviously cannot be assumed in general, especially if the PN description has been generated automatically.

The rest of this chapter suggests a technique combining two approaches to the verification of PN-based models: the PN-unfolding approach, based on partial order and the PN reachability symbolic traversal approach. The overall verification framework still employs many useful properties of the PN-unfolding algorithm, such as boundedness and safety checks as well as the properties of its signal transition interpretation, relevant to the hazard-freedom of the analysed circuit. However, in the new framework, the new role for the PN-unfolding is to produce a more efficient variable ordering.

7.7 Symbolic Traversal of Petri Net State Space

There are several methods of representing logic functions such as truth tables, Karnaugh maps, minterm canonical form or the sum of products form. Operating with these representations is inefficient for relatively big logic functions.

Binary Decision Diagrams (BDDs) were proposed as a means of canonical representation of logic functions in a graphical form. For a detailed introduction to BDDs and their basic

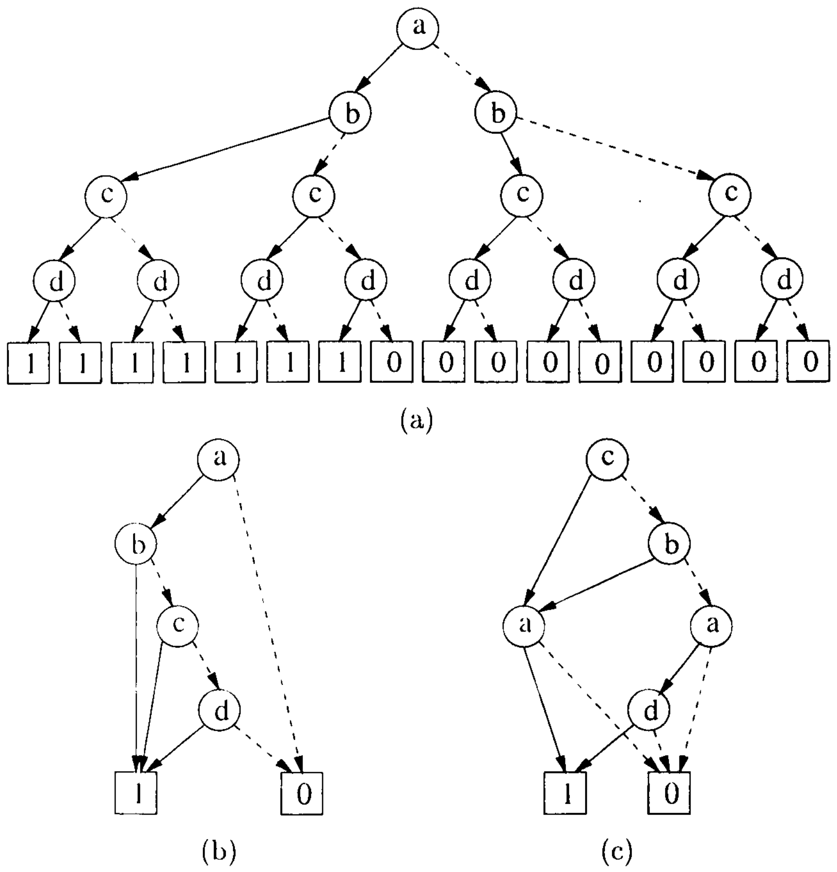


Figure 7.11: Example of different ordering of variables.

manipulations, the reader is referred to [11]. We will only briefly introduce BDDs and explain how they can be used for PN analysis.

Consider a logic function given below:

$$f = ab + ac + ad$$

A Binary Decision Tree (BDT) is constructed for this function using a straightforward algorithm. The BDT has two types of leaves labelled with 0 and 1. Leaves labelled with 0 indicate that if the values of variables are equal to those in the nodes along the path leading to such a node, then the function evaluates to FALSE. Leaves labelled with 1 have the opposite meaning. The BDT is constructed for a specific order of variables; in general the number of different BDTs is equal to the number of permutations in the variable order. The BDT for function f is shown in Figure 7.11(a), where dashed lines represent the value FALSE for each variable.

After performing transformational operations on the BDT [11] (merging equivalent nodes and eliminating the redundant ones) a BDD is obtained in the form shown in Figure 7.11(b). Evaluating all three representations (the Boolean function given above and its corresponding BDT and BDD) of this function will show that all these representations are equivalent. The number of nodes in the BDD for this variable ordering is 4 while the number of nodes in the BDT is 16.

BDDs were shown to be very powerful for representing Boolean functions. Boolean binary operations, such as conjunction and disjunction, on two functions represented by BDDs can be performed in polynomial time in the size of BDDs [11]. It has, however, been noted [11] that the size of BDDs strongly depends on the order of the variables in the function. For example using another order ($c < b < a < d$) for the same function will give a BDD shown in Figure 7.11(c). In general, the size of a BDD can be exponential in the number of variables, however, in practical examples the BDD usually has a smaller size when the appropriate ordering of its variables is used.

The use of BDDs for the PN analysis has been explained in [61]. The interested reader is referred to that work for a more elaborate description. In essence, a marking of a safe[4] PN N can be represented by means of a Boolean vector $V \in 2^{|P|}$. Then the fact that a place p_i is marked is denoted by asserting the value of corresponding element $V[i]$ to TRUE. A reachable marking M_n corresponds to a vector V_n and a Boolean function $R_n(V)$ which evaluates TRUE for V_n. Hence the reachability set of a given PN can be represented symbolically as the Boolean *reachability function*

$$R = \prod_{j=1}^{n} R_j.$$

Another set of variables $V' \in 2^{|T|}$ is introduced where each variable corresponds to one and only one transition of the PN. Using V and V' several other functions, which describe the behaviour of the PN, are obtained using the structural information about a PN, i.e. its flow relation. These include the *enabling function*, which has the domain $V \cup V'$ and evaluates to TRUE when the input places for a particular transition t are marked. The *firing function* evaluates to TRUE for the set of places which become marked after firing t.

From this representation, using standard Boolean operations such as quantification and substitution, all markings are obtained which are reachable from the initial one via firing the transitions enabled at M_0. Furthermore, since these functions evaluate to TRUE for a set of markings $\{M_1 \ldots M_m\}$, the set of all markings reachable from this set via the firing of all transitions enabled at all markings in $\{M_1 \ldots M_m\}$ is found in the same way.

This gives rise to an efficient algorithm constructing R for a given PN. A detailed algorithm was developed [61] which uses the BDD representation of the reached markings and iteratively constructs the symbolic representation in the form of Boolean function. This method is called the PN *symbolic traversal*, and the function R is called the *traverse function.* For clarity, this algorithm is reproduced here in a slightly simplified form. The pseudo-code of the algorithm is shown in Figure 7.12. $\mathcal{T}(f)$ denotes a function which returns a BDD representing the set of markings reachable from the markings represented by f. The time of traversing a PN and the size of the BDD constructed during traversal strongly depends on the order of variables which are defined on the places of the PN. If this ordering is unsatisfactory, then the symbolic traversal procedure may even never complete. Thus, finding an optimal ordering is a crucial task for the symbolic traversal approach. It is impossible[5] to find a good ordering without referring to some extra information available from the net. The algorithm introduced in the next section uses the PN-unfolding segment built from the original PN to obtain an ordering yielding a BDD whose size is reasonable.

[4]Existing PN symbolic traversal methods deal only with safe PNs; analysis of unsafe PNs requires additional encoding of tokens in unsafe places.

[5]Unless a completely greedy enumeration of all possible orderings is performed.

```
proc Construct BDD(PN = $\langle P, T, F, M_0 \rangle$)
    NEW = BDD($M_0$)
    REACHED = BDD(FALSE)
    while NEW $\neq$ BDD(FALSE) do
        REACHED = REACHED + NEW
        NEW = $\mathcal{T}$(NEW)
    end do
end proc
```

Figure 7.12: Pseudo-code for construction of the BDD representation of the state space of a PN.

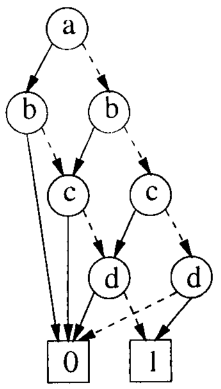


Figure 7.13: Example of a BDD.

7.8 Variable Ordering by Means of Unfolding

Consider the BDD built for a formula:

$$f = a\bar{b}\bar{c}\bar{d} + \bar{a}b\bar{c}\bar{d} + \bar{a}\bar{b}c\bar{d} + \bar{a}\bar{b}\bar{c}d$$

which is given in Figure 7.13. Each term of this formula has one variable in normal form and all the others in complementary form. In general, these type of functions can be written as:

$$f = \sum_{j=1}^{n}[a_j \cdot \prod_{i=1}^{n}(\overline{a_i} \; : \; i \neq j)]$$

If each variable is associated with only one place in a PN, then such a formula will evaluate to TRUE for the markings which have *only* one place marked.[6] Note that the size of the BDD does not depend on the order of the variables used in it. The size and the structure of the BDD remain the same for any possible orderings; it is essentially the same BDD which has some nodes renamed.

According to the PN symbolic traversal algorithm, each of the places of the original PN corresponds to a variable in the traverse function. Hence, any subset of places which can never be marked at the same time will have its traverse function in the form of function f given above. An *ME-cluster* (or simply a cluster) is defined as a set of places of the original PN

[6]This type of formulas is also used in the *one-hot encoding* technique [36].

```
proc Get clusters $(N, N')$
  for each place $p$ in $P$ do
    find a cluster $C$ in CLUSTERS such that $p$ is orthogonal to every $p_i$ in $C$
    if such cluster $C$ found then do
      add $p$ to $C$
    else
      add new cluster $C_n = \{p\}$ to CLUSTERS
    end do
  end do
  return CLUSTERS
end proc
```

Figure 7.14: Pseudo-code for clustering algorithm.

that cannot be marked simultaneously. The problem of dividing places of a PN into clusters is, in fact, NP-hard. A heuristic suggested here allows a more efficient calculation of clusters to be used for variable ordering.

The result in Chapter 4 (Proposition 4.1.1) shows that the PN-unfolding segment represents the concurrency relation between elements of the original PN. Hence, for two places it is said that they are in an *orthogonality relation* (or simply orthogonal to each other) iff they are not concurrent. Using the orthogonality relation, places of a PN are split into clusters with the algorithm similar to the graph colouring algorithm given in [26]. The pseudo-code of this algorithm is shown in Figure 7.14.

This is a greedy algorithm which does not check all the possible clustering of the PN. As it can be easily seen, not every clustering of places will yield the smaller size of BDD. It was observed that if the clusters are *balanced* in number of places contained in them, then such a clustering produces better results. One of the possible ways to reach a balanced clustering within this approach is to order the places of the PN in the ascending order of the number of their outgoing arcs. Then the places which can be included into the largest number clusters will be considered last [68].

It is convenient to represent the orthogonality relation between places as a table, Δ, where each element is found as follows:

$$\Delta[i,j] = \begin{cases} 0 & \text{if } p_i \| p_j \\ 1 & \text{otherwise} \end{cases}$$

Obtaining clusters themselves is obviously not enough. Clusters should also be ordered with the same goal to minimise the size of BDDs. In order to achieve this the *degree of orthogonality*, δ_{ij}, is defined between two clusters P_i and P_j using the matrix of orthogonality relations Δ as:

$$\delta_{ij} = |\{(p_k, p_n) : \Delta[k,n] = 1\}| \quad \forall p_k \in P_i, \forall p_n \in P_j$$

In other words, the degree of orthogonality is calculated as the number of mutually exclusive pairs of places between two clusters.

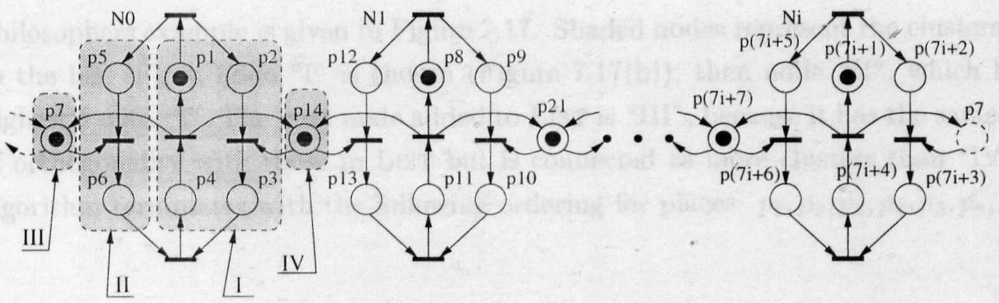


Figure 7.15: Dining philosophers benchmark.

```
proc Order clusters (CLUSTERS)
    choose cluster with highest degree of orthogonality
    add chosen cluster to the LIST
    while not all clusters are in the LIST do
        choose cluster which has greatest $\delta$ with most of clusters in LIST
        add the chosen cluster at the end the LIST
    end do
    return LIST
end proc
```

Figure 7.16: Pseudo-code for cluster ordering algorithm.

Example. To illustrate the clustering algorithm, consider a well-known example of dining philosophers. Consider first a subnet, N_0, taken alone from the PN shown in Figure 7.15. After clustering there are four ME-clusters as shown by the shaded areas. The degree of orthogonality is represented as a graph, in Figure 7.17(a) where each node corresponds to a cluster and the arcs are inscribed with the degree of orthogonality between two clusters. The arcs representing the 0 degree of orthogonality, i.e. independence of places belonging to different clusters, are omitted. ■

The next step is to order the ME-clusters between themselves. At this point a greedy algorithm which orders the clusters according to their degree of orthogonality is used. The algorithm itself is given in Figure 7.16.

Example. The illustration of application of the cluster ordering algorithm to the dining

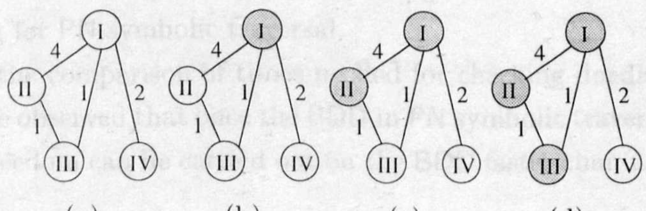


Figure 7.17: Steps of cluster ordering algorithm.

philosophers example is given in Figure 7.17. Shaded nodes represent the clusters added to the list. First, node "I" is chosen (Figure 7.17(b)), then node "II", which has the highest δ with "I". The next node added to LIST is "III", because it has the same degree of orthogonality with those in LIST but is connected to more clusters than "IV". The algorithm terminates with the following ordering for places: $p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_{14}$. ∎

Consider the complexity of the algorithms suggested above. The complexity of the algorithm deriving the ME-clusters is $O(|P'|^2)$, i.e. quadratic in the number of instances in the PN-unfolding segment. Finding the degree of orthogonality between clusters also has $O(|P'|^2)$ complexity. Finally, the ordering of clusters has complexity $O(|P'|)$. Thus, the overall complexity of the clustering procedure is $O(|P'|^2)$.

7.9 Experimental Results

The clustering procedure was implemented within PUNT and run on a Sun SPARC5. The PN symbolic traversal software was developed at UPC[7] using the CMU[8] BDD package.

In order to demonstrate the performance of the developed algorithm a set of benchmarks [68] was chosen which included such examples as dining philosophers (Figure 7.15) and Muller pipelines. Both types of models are scalable and can be easily grown by instantiating an additional number of generic fragments. The results of experiments for PN symbolic traversal and for the ordering algorithm using PN unfolding can be observed from Table 7.1.

Table 7.1 presents the results of PN symbolic traversal for three different variable orderings. The table presents the time (in second) on the construction of BDD representation (columns "Time") and the size of the BDD (columns "BDD size"). The first two columns present the benchmark name and the size of the state space. The next two columns presents the results for PN-symbolic traversal for an arbitrary ordering[9] which existed in the description of the benchmark and the *sift* dynamic reordering procedure supplied with CMU BDD package. The last two columns present the results for the combined technique suggested in the previous section. The timing results for BDD traversal (column "Cluster order") include the time spent on construction of the PN-unfolding segment and obtaining the ordering for the BDD. The performance of the algorithm obtaining the variable ordering only is given in column "PUNT" which show the fraction of time and the size of the required segment. The size of the segment is given in the number of transition instances which indicates the number of reachable states actually visited during the construction of the segment.

The results demonstrate the gains achieved by using the PN-unfolding segment for deriving the variable ordering for PN-symbolic traversal.

Table 7.2 shows the comparison of times needed for checking deadlock freedom, for both approaches. It can be observed that once the BDD in PN symbolic traversal has been built, the check for deadlock freedom can be carried out on the BDD faster than using the PN-unfolding segment.

[7]Universitat Politècnica de Catalunya, Spain

[8]Carnegie-Mellon University, USA

[9]There exists an ordering with which the PN symbolic traversal fails to construct the BDD.

Benchmark	States	Arbitrary order		Dynamic reorder		Clustering order		PUNT	
		Time	BDD size	Time	BDD size	Time	BDD size	Time	Size [tr]
10 phil	4.86×10^{6}	3.99	554	4.45	357	2.39	357	0.60	50
20 phil	2.19×10^{13}	18.33	1174	19.39	737	10.03	737	1.73	100
30 phil	1.03×10^{20}	42.93	1794	44.48	1117	23.30	1117	3.66	150
40 phil	$\approx 10^{27}$	76.93	2414	79.18	1497	41.13	1497	6.68	200
50 phil	$\approx 10^{34}$	122.05	3034	126.37	1877	64.11	1877	10.74	250
15 pipe	6006	22.49	1639	22.87	1153	12.17	715	1.11	70
30 pipe	6.01×10^{7}	754.81	6694	786.38	4518	352.13	2635	11.61	240
45 pipe	6.90×10^{11}	6827.03	15149	6988.88	10665	2753.51	5755	76.91	510

Table 7.1: Experimental results for PN symbolic traversal.

Benchmark	PN unfolding	PN traversal
10 phil	0.18	0.17
20 phil	1.32	0.85
30 phil	4.34	2.01
40 phil	10.40	3.66
50 phil	20.58	6.04

Table 7.2: Experimental results (deadlock detection)

The above comparison shows that both PN-unfolding analysis and PN symbolic traversal should be used in conjunction, complementing each other. For relatively small examples, the PN-unfolding segment can be used both for obtaining the variable ordering for future PN symbolic traversal and deadlock detection. When the examples grow larger it is more efficient to obtain the variable ordering and then proceed to PN symbolic traversal. At the same time, the PN-unfolding segment can be kept for later "backtracking" in the case a deadlock has been found during symbolic traversal, for the identification of the offending trace.

While building the PN-unfolding segment, the PN is checked for boundedness. In the case of an unbounded PN the trace leading to unboundedness is reported at the pre-processing period without wasting time on more expensive traversal. Using the STG-unfolding segment it is also possible to check the validity of the STG while building the segment itself. Thus, all the traces invalidating the behaviour described by the STG are reported at an earlier stage of the analysis. Once a good ordering is obtained, symbolic traversal methods [61, 70, 15] can be used to solve the CSC problems or problems of decomposition. In this case, STG-unfolding pre-processing allows a further extension of automated synthesis methods.

7.10 Conclusions

This chapter proposed a new algorithm for the analysis of contextual nets. This approach is based on the partial order approach and constructs a CN-unfolding segment which represents the set of reachable markings (or states for STGs). It attempts to overcome the problems of exponential explosion of the PN-unfolding which show up during the analysis of PN model of a CN, i.e. a PN with bi-directional arcs to represent contextual dependency. This explosion is caused by the difference in the firing semantics of two models. The PN-unfolding explicitly examines all possible interleavings of transitions whose firings lead to the same marking. The CN-unfolding, suggested here, preserves the step semantics of the transition firings.

The new method was implemented and used on a realistic example of a four slot communication mechanism. The behavioural trend of the new approach was demonstrated using a scalable benchmark. Experimental results demonstrate the power of the new CN-unfolding algorithm.

The second suggested application of the PN-unfolding technique is based on the combination of partial order (PN unfolding) and symbolic traversal techniques. This approach uses an algorithm for obtaining the ordering of variables in BDDs employed for symbolic traversal of the PN state space. Experimental results show that this approach is practical for the known set of benchmarks.

Chapter 8

Conclusions

This chapter presents the summary of contributions offered by this work, namely the improvement of existing PN analysis techniques based on the partial order approach, novel techniques suggested for the verification of PN and STG models of specifications and implementations of asynchronous circuits, synthesis of SI circuits from STGs. It also outlines potential areas of further research which are opened by this work.

8.1 Summary

This thesis presented the theory and application of partial order approach to asynchronous circuit design. It follows the top-down design methodology and presents new methods for automated verification of event-based descriptions and synthesis of SI circuits from their STG specifications.

Extension of the existing PN-unfolding technique This work presented a comparative study of the existing methods for PN analysis based on the PN-unfolding method. This work examined potential redundancy in the PN-unfolding truncation constructed by the original McMillan's algorithm and presented the cutoff condition for safe PNs due to [22]. In addition, this thesis presented a promising pruning technique (and a new termination condition) which avoids the redundancy of the PN-unfolding truncation caused by the unsafeness of places in a PN. This condition uses temporal relations which are available in the unfolding. A new version of the algorithm constructing a finite fragment of the PN-unfolding was suggested. This fragment is called a PN-*unfolding segment.*

Verification of PN models PNs are used to model a variety of asynchronous systems. This work suggested new algorithms for the verification of properties of PN models such as safeness, persistency and liveness using a PN-unfolding segment. These properties are crucial for establishing the correctness of behavioural models represented by PNs. This thesis also described a modelling approach for the verification of already existing circuits. This approach is based on representing each element of the circuit by a fragment of a Labelled PN. The fragments are composed together along with the model of the environment. The resulting circuit model is verified for absence of hazards using the PN-unfolding segment technique.

Verification of STG models Conventional STG verification techniques construct a State Graph which represents all reachable states of an STG. This work revisited this verification approach and showed that it may produce false alarms for STGs with acyclic parts. To overcome this problem, this thesis proposed a new concept, *Full State Graph*, which represents adequately the behaviour of an arbitrary STG. In addition, this work presented an adaptation of the PN-unfolding analysis for the analysis of STG models. The new representation, called STG-*unfolding segment*, takes into account the signal interpretation of transitions in an STG and allows verification of the STG properties. The STG-unfolding segment approach was applied to the analysis of four-phase circuits, whose behaviour can be described using STG fragments.

Synthesis of SI circuits The problem of SI circuit synthesis from STGs is of paramount importance to the whole process of SI circuit design. The conventional approach requires the construction of the SG and only then the implementation can be obtained. Unfortunately, often the SG cannot be built due to computational limits, even though the verification process shows no problems with the STG description. This work builds upon recent research in structural synthesis methods. It uses a compact implicit representation of the SG in the form of the STG-unfolding segment. Thus, once the specification has been verified, the STG-unfolding segment is reused in the synthesis process. Two approaches were presented in this work: exact and approximate. Although the implementation can be obtained using the exact method, the approximation approach produces comparable implementations in a fraction of time needed for any other method to synthesise the same set of benchmarks.

Analysis of CNs Analysis of CN models falls into the category where the high degree of concurrency makes their analysis intractable for the Reachability Graph analysis methods, and the modelling of contextual dependency using bi-directional arcs in PNs makes it hard for the PN-unfolding approach. The RG-based approach runs into computational limits. At the same time, the unfolding approach is too powerful in distinguishing the state of a place before and after a contextually dependent transition fires. However, CN models proved to be useful in modelling of asynchronous systems and circuits. This thesis suggested a new method of CN analysis which is based on the PN-unfolding but preserves the contextual dependency between transitions and places.

Combining partial orders and symbolic traversal Some of the problems in asynchronous circuit design, e.g. resolving coding conflicts, are still solved using the RG representation of the PN or STG model. Recently proposed PN symbolic traversal methods represent state spaces using Binary Decision Diagrams. They demonstrated the ability to handle large state spaces at a relatively low cost. However, these methods may suffer from a bad variable ordering in BDDs. Since variables in these methods are associated with places of a PN, the knowledge about the behaviour of these places assists in reducing the size of the BDD representation of its state space. This work proposes a combined approach where the PN-unfolding segment is used for obtaining a good variable ordering and the PN symbolic traversal is used for representing the state space.

Experimental results The proposed approaches and algorithms were implemented in an experimental tool. The tool was applied to a wide selection of benchmarks which includes PN and STG specifications as well as circuit models.

Three realistic examples were used. The first two illustrated the motivation and need for the new PN analysis method based on the PN-unfolding segment. Using the PN-unfolding segment it was possible to detect the behaviour in a production cell example which could not have been identified using any previously existing PN-unfolding approaches due to the large size of the model. The third example illustrated the application of CN unfolding to the analysis of a communication mechanism used in a real communication system. The verification of PN and STG models included control circuits from the AMULET1 microprocessor and a selection of other available circuits.

In addition, a number of publicly available benchmarks were used to show advantages of the new synthesis method based on the STG-unfolding segment. The synthesis procedure was also used to produce an implementation from the counterflow pipeline processor, one of the proposed challenges in the asynchronous community. The performance of the new methods was also tested on well-known scalable examples such as PN descriptions of dining philosophers and Muller pipeline.

Experimental results presented in this thesis confirm the advantages of using the partial order approach in the verification, analysis and synthesis of PN and STG models.

8.2 Areas of Further Research

The potential for future research in application of partial order methods, such as on PN-unfolding technique, in asynchronous circuit design is enormous. Experimental results observed during this work prompted several new applications.

Timing analysis Analysis of models which include some information about the duration of events is gaining recognition among formal method developers. In order to produce a better implementation designers often take shortcuts by making assumptions about propagation delays in different parts of the circuit. Such low-level optimisation is often done by hand and therefore requires verification to guarantee the correctness of the circuit. In addition, taking realistic delay assumptions into the high-level model may show that in reality the system behaves correctly. For example, realistic timing assumption in the production cell example may change the limit on the number of initially available blanks.

Several works, e.g. [72, 78], have indicated that it is possible to apply partial order techniques for the analysis of models with timing information. This area, however, requires further theoretical investigation.

Solving CSC problem One of the problems being currently investigated in SI circuit synthesis is how to resolve coding conflicts for STGs which do not satisfy the Complete State Coding condition. This work proposed a refinement technique for resolving the intersection between approximated covers. If refined covers still intersect, the implementation is impossible. An different approach is to attempt to resolve coding conflicts by means of inserting additional signals. Solving this problem would require identification of insertion points on the STG-unfolding segment.

Technology mapping The problem of technology mapping for asynchronous circuits deals with the implementation of a circuit in a given gate library, i.e. it requires an implementation of the circuit using a restricted set of gates. The synthesis procedure suggested here assumed that almost any gate exists in the library. This is obviously not the case for many real designs. Solving this problem requires decomposition of the boolean logic into simpler components preserving the speed-independence. The technology mapping problem can often be reformulated as the problem of inserting additional signals into the specification and then resolving the coding conflicts. Thus it is closely related to the CSC problem, discussed above.

Synthesis using extended covers The synthesis procedure proposed in this work produces basic implementations in Atomic Complex Gate per Excitation Function and Atomic Complex Gate per Excitation Region architectures. It is known that covers implementing output signals may be allowed to cover states at which these signals are stable. Such cover may be obtained by extending excitation slices with new cuts, thus expanding the logic optimisation domain. Extended covers are required to be monotonous, i.e. the set of covered states may not contain a state which causes an opposite transition of the signal. The problem of cover monotony verification is closely related to the cover evaluation problem and can be also solved on the STG-unfolding segment.

Testing of asynchronous circuits The last stage of the design process is testing. All current testing machines (and techniques) assume synchronous mode of operation. To enable testing of asynchronous circuits in synchronous test machines a new approach is needed.

The testing procedure requires constructing test sequences, which consist of input stimuli and output responses for a circuit. The testing of fundamental mode circuits, which have well-defined sets of changing inputs, are very close to the testing of synchronous ones. For fundamental mode circuits, permutations of input signal changes in one set lead to the same response on the outputs.

SI circuits may respond differently to different sequences of inputs changes. Obviously it is impossible to test all possible combinations of inputs allowed by an SI circuit specification. However, it is possible to identify those states in which the outputs of a circuit are stable and the circuit is awaiting for a change in the input signals. A set of these states can be used in a synchronous test machine to test the circuit. Identification of such states can be made on the STG-unfolding segment constructed for an STG model of the circuit (or its specification). Since each state in the SG (or FSG) of an STG corresponds to a cut in the STG-unfolding segment, the approach should be similar to the one discussed in this thesis for the synthesis of SI circuits.

Use of BDDs in STG-unfolding Finally, the problem of synthesis requires efficient implementation and operation on binary vectors. One of the most efficient methods to represent them is BDDs. Therefore, a combined approach which constructs the BDD representation of the states represented by an STG-unfolding segment is required. The objective of this combination is efficient derivation of implementations from an STG-unfolding segment rather than using it only as a pre-processing method for variable ordering, which was suggested in this thesis.

Bibliography

[1] Amulet1 group workshop: Presentation materials. Technical report, Manchester University, Department of Computer Science, Amulet Group, Lake District, England, July 18-22 1994.

[2] P. A. Beerel. *CAD Tools for the Synthesis, Verification, and Testability of Robust Asynchronous Circuits*. PhD thesis, Stanford University, 1994.

[3] P. A. 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.

[4] K. van Berkel. *Handshake Circuits: An Intermediary between Communicating Processes and VLSI*. PhD thesis, Eindhoven University of Technology, 1992.

[5] K. van Berkel, R. Burgess, J. Kessels, Ad Peeters, M. Roncken, and F. Schalij. Asynchronous circuits for low power: A DCC error corrector. *IEEE Design & Test of Computers*, 11(2):22–32, Summer 1994.

[6] K. van Berkel, R. Burgess, J. Kessels, Ad Peeters, M. Roncken, and F. Schalij. A fully-asynchronous low-power error corrector for the DCC player. In *International Solid State Circuits Conference*, pages 88–89, February 1994.

[7] K. van Berkel, J. Kessels, M. Roncken, R. Saeijs, and F. 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] E. Brunvand. *Translating Concurrent Communicating Programs into Asynchronous Circuits*. PhD thesis, Carnegie Mellon University, 1991.

[9] E. Brunvand. The NSR processor. In *Proc. Hawaii International Conf. System Sciences*, volume I. IEEE Computer Society Press, January 1993.

[10] E. Brunvand and R. F. Sproull. Translating concurrent programs into delay-insensitive circuits. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 262–265. IEEE Computer Society Press, November 1989.

[11] R. E. Bryant. Graph-based algorithms for boolean function manipulations. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[12] N. Busi and G. M. Pinna. Non-sequential semantics for contextual P/T nets. In *Proc. of 17th Int. Conf. on Applications and Theory of Petri Nets, Osaka*, pages 113–132, June 1996.

[13] T. A. Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-theoretic Specifications.* PhD thesis. MIT, 1987.

[14] 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.

[15] J. Cortadella. M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. In *Proc. of the 11th Conf. Design of Integrated Circuits and Systems*, pages 205–210. Barcelona. Spain. November 1996.

[16] J. Cortadella. A. Yakovlev, L. Lavagano, and P. Vanbekbergen. Designing asynchronous circuits from behavioral specifications with internal conflicts. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 106–115, November 1994.

[17] P. Day and J. V. Woods. Investigation into micropipeline latch design styles. *IEEE Transactions on VLSI Systems*, 3(2):264–272, June 1995.

[18] J. Desel and J. Esparza. *Free Choice Petri Nets.* Cambridge University Press, Cambridge, 1995.

[19] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* The MIT Press, Cambridge, Mass., 1988. An ACM Distinguished Dissertation 1988.

[20] J. C. Ebergen. *Translating Programs into Delay-Insensitive Circuits*, volume 56 of *CWI Tract.* CWI. Amsterdam, 1989.

[21] J. Esparza. Model checking using net unfoldings. Technical Report 14/92, Universität Hildesheim, October 1992.

[22] J. Esparza, S. Römer. and W. Vogler. An inprovement of mcmillan's unfolding algorithm. Technical Report TUM-19599, Insitute Für Informatik, Technische Universität München, 1995.

[23] S. Furber. Computing without clocks: Micropipelining the ARM processor. In Graham Birtwistle and Al Davis, editors, *Asynchronous Digital Circuit Design*, Workshops in Computing, pages 211–262. Springer-Verlag, 1995.

[24] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods. AMULET1: A micropipelined ARM. In *Proceedings IEEE Computer Conference (COMPCON)*, pages 476–485, March 1994.

[25] S.B. Furber, P. Day, J.D. Garside, N.C. Paver, and J.V.Woods. A micropipelined ARM. In *VLSI'93, Grenoble*, 1993. Conference best paper award.

[26] F. Gavril. Algorithms for minimum colouring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SIAM Jornal on Computing*, 1(4):180–187, December 1972.

[27] H. J. Genrich and R. M. Shapiro. Formal verification of an arbiter cascade. In *Proceedings of the 13th International Conference on Application and Theory of Petri Nets*, pages 205–223. 1992.

[28] P. Godefroid and P. Wolper. Using partial orderes for efficient verification of deadlock freedom and safety properties. *Formal Methods in System Design*, 2(2):149–164, April 1993.

[29] S. Hauck. Asynchronous design methodologies: An overview. *Proceedings of the IEEE*, 83(1), January 1995.

[30] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, New York, 1985.

[31] D. A. Huffman. The synthesis of sequential switching circuits. In E. F. Moore, editor, *Sequential Machines: Selected Papers*. Addison-Wesley, 1964.

[32] R. Janicki and M. Koutny. On equivalent execution semantics of concurrent systems. In *Lecture Notes in Computer Science, Vol. 266*. Springer-Verlag, 1987.

[33] M. B. Josephs, C. A. R. Hoare, and He Jifeng. A theory of asynchronous processes. Technical Report PRG-TR-6-89, Oxford Univ., Computing Laboratory, 1989.

[34] M. B. Josephs and J. T. Udding. An algebra for delay-insensitive circuits. In R. P. Kurshan and E. M. Clarke, editors, *Proc. International Workshop on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 343–352. Springer-Verlag, 1990.

[35] M. Kishinevsky, A. Kondratyev, A. Taubin, and V. Varshavsky. *Concurrent Hardware: The Theory and Practice of Self-Timed Design*. John Wiley and Sons, London, 1993.

[36] A. Kondratyev, J. Cortadella, M. Kishinevsly, E. Pastor, O. Roig, and A. Yakovlev. Checking Signal Transition Graph implementability by symbolic BDD traversal. In *EDTC-95*, pages 325–332, 1995.

[37] A. Kondratyev, M. Kishinevsky, B. Lin, P. Vanbekbergen, and A. Yakovlev. Basic gate implementation of speed-independent circuits. In *Proceedings of ACM/IEEE Design Automation Conference*, pages 56–62, June 1994.

[38] A. Kondratyev, M. Kishinevsky, and A. V. Yakovlev. On hazard-free implementation of speed-independent circuits. In *Proceedings of Asia and South Pasific Design Automation Conference (ASP-DAC'95)*, pages 241–248, 1995.

[39] A. Kondratyev and A. Taubin. On verification of the speed-independent circuits by STG unfoldings. In *International Symposium on Advanced Research in Asynchronous Circuits and Systems, Salt Lake City, Utah, USA*, pages 64–75, November 1994.

[40] L. Lavagno. Personal communications.

[41] L. Lavagno and A. Sangiovanni-Vincentelli. *Algorithms for Synthesis and Testing of Asynchronous Circuits*. Kluwer Academic Publishers, Massachusetts, 1993.

[42] C. Lewerentz and T. Linder. Formal development of reactive systems - Case study production cell. *LNCS 891*. 1995.

[43] A. J. Martin. Collected papers on asynchronous VLSI design. Technical Report Caltech-CS-TR-90-09, California Insitute of Technology, 1990.

[44] A. 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.

[45] A. J. Martin. Synthesis of asynchronous VLSI circuits. In J. Staunstrup (ed.), editor, *Formal Methods for VLSI Design*, chapter 6. North Holland, Amsterdam, 1990. IFIP WG 10.5 Lecture Notes.

[46] K. L. McMillan. Using unfolding to avoid the state explosion problem in the verification of asynchronous circuits. In *Proceedings of the 4th Workshop on Computer Aided Verification*, pages 164–177, Montreal, 1992.

[47] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, 1993.

[48] Teresa H.-Y. Meng, Robert W. Brodersen, and David G. Messerschmitt. Automatic synthesis of asynchronous circuits from high-level specifications. *IEEE Transactions on Computer-Aided Design*, 8(11):1185–1205, November 1989.

[49] R. E. Miller. *Switching theory*, chapter 2, pages 192–244. Wiley and Sons, 1965.

[50] T. Miyamoto and S. Kumagai. An efficient algorithm for deriving logic functions of asynchronous circuits. In *Proc. of the Second International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC'96)*, pages 30–35, Aizu-Wakamatsu, Fukushima, Japan, March 1996.

[51] C. E. Molnar, T.-P. Fang, and F. 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.

[52] U. Montanari and F. Rossi. Contextual nets. *Acta Informatica*, 36:545–596, 1995.

[53] D. 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.

[54] T. Murata. Petri nets: properties, analysis and applications. *Proceedings of IEEE*, 77(4):541 580, April 1989.

[55] T. Nanya. A quasi-delay-insensitive microprocessor: Titac-I. In *Proceedings of 1995 Israel Workshop on Asynchronous VLSI*, March 1995.

[56] M. Nielsen, G. Plotkin, and G. Winskel. Petri nets, event structures and domains. part I. *Theoretical Computer Science*, 13:85–108, 1981.

[57] S. M. Nowick. *Automatic Synthesis of Burst-Mode Asynchronous Controllers*. PhD thesis, Stanford University, Department of Computer Science, 1993.

[58] S. M. Nowick and D. L. Dill. Practicality of state-machine verification of speed-independent circuits. In *Proceedings of ICCAD'89*, Santa Clara, CA, November 1989.

[59] S. M. Nowick and D. 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.

[60] S. M. Nowick and D. L. Dill. Synthesis of asynchronous state machines using a local clock. In *Proc. International Conf. Computer Design (ICCD)*, pages 192–197. IEEE Computer Society Press, October 1991.

[61] E. Pastor, O. Roig, J. Cortadella, and R. M. Badia. Petri net analysis using boolean manipulations. In *Proceedings of 15th International Conference on Application and Theory of Petri Nets, Zaragoza, Spain, June 1994*, pages 416–435, 1994.

[62] P. Pastor, J. Cortadella, A. Kondratyev, and O. Roig. Structural methods for the synthesis of speed-independent circuits. In *Proc. European Design and Test Conference (EDAC-ETC-EuroASIC)*, pages 340–347, Paris(France), March 1996.

[63] S. S. Patil. Cellular arrays for asynchronous control. In *Proceedings of the ACM 7th Annual Workshop on Microprogramming*, 1974. (CSG Tech. Memo. 122, Project MAC, MIT, April 1975).

[64] N.C. Paver. *The design and implementation of an asynchronous microprocessor*. PhD thesis, University of Manchester, 1994.

[65] J. L. Peterson. *Petri Net Theory and the modeling of systems*. Prentice-Hall, Inc., 1981.

[66] I. Reicher and M. Yoeli. Net-based modelling of communicating parallel processes with applications to VLSI design. Technical Report 532. Technion. Haifa. 1988.

[67] W. Reisig. *Petri Nets: an Introduction*. EATCS Monographs in Computer Science. Springer-Verlag, Berlin, 1985.

[68] O. Roig. Personal communications.

[69] O. Roig. *Formal Verification and Testing of Asynchronous Circuits*. PhD thesis, Universitat Politècnica de Catalunya (UPC), 1997.

[70] O. Roig, E. Pastor, and J. Cortadella. Symbolic model checking for speed independent circuits. In *Proceedings of ACiD Workshop on Asynchronous Low Power VLSI, Lyngby, Denmark*, April 1993.

[71] Per Torstein Roine. Asynchronous FIFO buffer for multicomputer applications. Master's thesis, Department of Informatics, University of Oslo, 1994.

[72] T.G. Rokicki. *Representing and modeling digital circuits.* PhD thesis, Stanford University, 1993.

[73] L. Ya. Rosenblum and A. V. Yakovlev. Signal graphs: from self-timed to timed ones. In *Proceedings of International Workshop on Timed Petri Nets, Torino, Italy, July 1985*, pages 199–207. IEEE Computer Society, 1985.

[74] C. L. Seitz. System timing. In C. A. Mead and L. A. Conway, editors, *Introduction to VLSI Systems*, chapter 7. Addison-Wesley, 1980.

[75] A. Semenov, A. M. Koelmans, L. Lloyd, and A. Yakovlev. Designing an Asynchronous Processor Using Petri Nets. *IEEE Micro*, pages 54–64, Mar/Apr 1997.

[76] A. Semenov and A. Yakovlev. Event-based framework for verification of high-level models of asynchronous circuits. Technical Report 487, University of Newcastle upon Tyne, 1994.

[77] A. Semenov and A. Yakovlev. Combining partial orders and symbolic traversal for efficient verification of asynchronous circuits. In *Proceedings of CHDL'95, Chiba, Japan*, pages 567–573, 1995.

[78] A. Semenov and A. Yakovlev. Verification of asynchronous circuits using time Petri-net unfolding. In *Proc. ACM/IEEE Design Automation Conference*, pages 59–63, 1996.

[79] A. Semenov and A. Yakovlev. Contextual net unfolding and asynchronous system verification. Technical Report 572, University of Newcastle upon Tyne, 1997.

[80] A. Semenov, A. Yakovlev, E. Pastor, M. Peña, J. Cortadella, and L. Lavagno. Partial order based approach to synthesis of speed-independent circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems.* IEEE Computer Society Press, April 1997.

[81] A. Semenov, A. Yakovlev, E. Pastor, M. A. Peña, and J. Cortadella. Synthesis of speed independent circuits from STG-unfolding segment. In *Proc. ACM/IEEE Design Automation Conference*, pages 16–21, 1997.

[82] E.M Sentovich *et. al.* SIS: A system for sequential circuit synthesis. Memorandum No. UCB/ERL M92/41, University of California, Berkeley, 1992.

[83] P. Siegel and G. De Micheli. Decomposition methods for library binding of speed-independent asynchronous designs. In *Proceedings of the International Conference on Computer-Aided Design*, pages 558–565, 1994.

[84] H. R. Simpson. Four-slot fully asynchronous communication mechanism. *IEE Proceedings-E*, 137(1):17–30, January 1990.

[85] H. R. Simpson. Correctness analysis for class of asynchronous communication mechanisms. *IEE Proceedings-E*, 139(1):35–49, January 1992.

[86] R. F. Sproull and I. E. Sutherland. *Asynchronous Systems.* Sutherland, Sproull and Associates, Palo Alto, 1986. Vol. I: Introduction, Vol. II: Logical effort and asynchronous modules, Vol. III: Case studies.

[87] K. S. Stevens. *Practical Verification and Synthesis of Low Latency Asynchronous Systems.* PhD thesis. Dept. of Computer Science, University of Calgary, Canada, September 1994.

[88] I. E. Sutherland. Micropipelines. *Communications of ACM*, 32(6):720–738, 1989.

[89] S. H. Unger. *Asynchronous Sequential Switching Circuits.* Wiley-Interscience, New York, 1969.

[90] S. H. Unger. Hazards, critical races, and metastability. *IEEE Transactions on Computers*, 44(6):754–768, June 1995.

[91] A. Valmari. Stubborn Sets for Reduced State Space Generation. *Advances in Petri Nets 1990, ed.G.Rozenberg. LNCS 483*, pages 491–515, 1991.

[92] 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.

[93] V. I. Varshavsky *et. al.*, 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.

[94] W. Vogler. Personal communications.

[95] W. Vogler, A. Semenov, and A. Yakovlev. Partial order semantics and read arcs. Technical Report 634, University of Newcastle upon Tyne, February 1998.

[96] J. Xu, B. Randell, A. Romanovsky, C. Rubira, R. Stroud, and Z. Wu. Fault tolerance in concurrent object-oriented software through coordinated error recovery. In *Proceedings of the Twenty Fifth Annual International Symp. on Fault-Tolerant Computing.*, pages 499–509, 1995.

[97] A. Yakovlev. Designing control logic for counterflow pipeline processor using Petri nets. Technical Report 522, University of Newcastle upon Tyne, 1995.

[98] A. Yakovlev, M. Kishinevsky, and A. Koelmans. Petri net based modelling and analysis of micropipelines. in [1].

[99] A. Yakovlev, L. Lavagno, and A. Sangiovanni-Vincentelli. A unified signal transition graph model for asynchronous control circuit synthesis. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 104–111. IEEE Computer Society Press, November 1992.

[100] A. Yakovlev, V. Varshavsky, V. Marakhovsky, and A. Semenov. Designing an asynchronous pipeline token ring interface. In *Asynchronous Design Methodologies*, pages 32–41. IEEE Computer Society Press, May 1995.

[101] A. V. Yakovlev, A. M. Koelmans, A. Semenov, and D. J. Kinnement. Modelling, analysis and synthesis of asynchronous control circuits using Petri nets. *Integration, the VLSI journal*, 21(3):143–170, December 1996.

[102] Ch. Ykman-Couvreur, B. Lin, and H. DeMan. ASSASSIN: A synthesis system for asynchronous control circuits. Reference manual, IMEC, 1995.

[103] Ch. Ykman-Couvreur, B. Lin, G. Goossens, and H. 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.

[104] K. Y. Yun. Automatic synthesis of extended burst-mode circuits using generalized C-elements. In *Proc. European Design Automation Conference (EURO-DAC)*, September 1996.

[105] K. Y. Yun and D. L. Dill. Automatic synthesis of 3D asynchronous state machines. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 576–580. IEEE Computer Society Press, November 1992.

[106] K. Y. Yun, D. L. Dill, and S. M. Nowick. Synthesis of 3D asynchronous state machines. In *Proc. International Conf. Computer Design (ICCD)*, pages 346–350. IEEE Computer Society Press, October 1992.