Technical Report

UCAM-CL-TR-587
ISSN 1476-2986

Number 587

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Verification of asynchronous circuits

Paul Alexander Cunningham

April 2004

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted January 2002 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Gonville and Caius College.

ISSN 1476-2986

To My Parents

Declaration

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration. In accordance with the regulations stated in the Memorandum to Graduate Students, this dissertation is less than 60,000 words in length, including tables and footnotes, but excluding appendices, bibliography, photographs and diagrams.

P. A. Cunningham

Acknowledgements

I would like to thank Peter Robinson, my supervisor, for his support and wisdom over the last three years. Without Peter I would never have become interested in asynchronous design or even in pursuing a PhD at Cambridge. I would also like to thank Simon Moore, Steev Wilcox, George Taylor, and Bob Mullins for their ideas and encouragement both at work and in the pub. In particular, I want to thank George and Bob for putting up with my many monologues on formal methods and for sharing in my frustration with our in-house chip design flow.

Further thanks go to Ivan Sutherland, Jo Ebergen, and the rest of the Asynchronous Design Team at Sun Microsystems for bringing me the three most enthralling months of my PhD. I would especially like to thank Scott Fairbanks for broadening my horizons and reminding me that shorting supply to ground can actually improve performance.

Finally, I would like to thank Sarah Haydon for her undying commitment and moral support at times when I needed it most.

This work was funded by the Engineering and Physical Sciences Research Council.

Abstract

The purpose of this thesis is to introduce proposition-oriented behaviours and apply them to the verification of asynchronous circuits. The major contribution of proposition-oriented behaviours is their ability to extend existing formal notations to permit the explicit use of both signal levels and transitions.

This thesis begins with the formalisation of proposition-oriented behaviours in the context of gate networks, and with the set-theoretic extension of both regular-expressions and trace-expressions to reason over proposition-oriented behaviours. A new trace-expression construct, referred to as biased composition, is also introduced. Algorithmic realisation of these set-theoretic extensions is documented using a special form of finite automata called proposition automata. A verification procedure for conformance of gate networks to a set of proposition automata is described in which each proposition automaton may be viewed either as a constraint or a specification. The implementation of this procedure as an automated verification program called Veraci is summarised, and a number of example Veraci programs are used to demonstrate contributions of proposition-oriented behaviour to asynchronous circuit design. These contributions include level-event unification, event abstraction, and relative timing assumptions using biased composition. The performance of Veraci is also compared to an existing event-oriented verification program called Versify, the result of this comparison being a consistent performance gain using Veraci over Versify.

This thesis concludes with the design and implementation of a 2048 bit dual-rail asynchronous Montgomery exponentiator, MOD_EXP, in a $0.18\mu m$ standard-cell process. The application of Veraci to the design of MOD_EXP is summarised, and the practical benefits of proposition-oriented verification are discussed.

Contents

List of figures

List of tables

List of algorithms

Chapter 1
Introduction

1.1 Synchronous and Asynchronous Design

MOS transistors are voltage controlled devices, and the state of a digital circuit can therefore be determined directly from the voltage levels on all its wires. To change the state of a digital circuit at least one wire must change level. A change in level is called an *event* and it is the sequence in which events occur that determines the behaviour of a digital circuit.

Since it is events that determine behaviour but levels that control transistors, the designers of digital MOS circuits are faced with an inherent level-event conflict that must be resolved before relating algorithm to logic. *Synchronous* design is a technique that resolves this conflict by asserting that time be quantised according to only one event on one wire, the clock. With every clock event the level on other circuit wires is sampled and that sample remains fixed until the following clock event. In this sense synchronous design eclipses all references to events not on the clock, and enables synchronous designers to adopt a logic-centric computation model that accurately relates computation to transistor functionality.

Asynchronous design is a term used to classify all digital circuits that are not synchronous, and therefore do not employ the use of global clocking. The resulting lack of a quantised timeline makes asynchronous circuits potentially sensitive to events on any wire, which the designers of asynchronous circuits typically resolve by adopting a channel-centric computation model. Channel-centric computation differs from logic-centric computation in that circuit operation is dependent on the movement of data between functional units, and level-event conflicts are resolved using dedicated communication protocols across channels.

Channel-centric computation complements logic-centric computation in that more attention is drawn to the connectivity between logic, and less to the logic itself. For example, the FLEET architecture of Sun Microsystems [23] permits only one instruction `MOVE A,B` where A is the output of one functional unit and B is the input to another functional unit. Computation in a FLEET machine is based entirely on which values are moved where.

Thirty years ago, in the early days of integrated circuit manufacture, it was the transistors and not the interconnect that dominated cost and performance. Synchronous designs produced faster, cheaper chips than their asynchronous counterparts, and were therefore deemed to be superior. The persistent and exponential growth of the semiconductor industry since those early days has resulted in a world where commercial asynchronous design is almost extinct. However, this is not to say that the extinction is justified, and in particular, the underlying relationship between transistors and interconnect is now inverted with respect to the past: Interconnect is now slow and takes substantially more chip area than transistors.

Although it is unclear whether asynchronous design will ever receive the same commercial success as its synchronous counterpart, the case in favour of asynchronous circuits is now strong, and research into asynchronous design methods remains active.

1.2 Formal Verification

A circuit is considered correct if the sequences of events that it performs result in the intended behaviour on its outputs, provided its inputs also behave as intended. Since intent is something internal to the mind of a designer, until intentions have been documented in some tangible form, be it on paper, or by word of mouth, correctness is by necessity something private between designer and circuit. To *verify* that a circuit is correct its intended behaviour must first be articulated in some unambiguous way, referred to as a *specification*. Once a specification has been made a well-defined procedure can then be executed to determine whether that circuit conforms to its specification.

When the specification and the conformance checker have a formal foundation, verification is akin to a mathematical proof that the circuit will always behave as intended. Such a proof is in contrast to simulation where it is merely demonstrated that a circuit responds in a certain way to a specific set of input stimuli. Unfortunately, formal verification is both computationally complex and its formal foundation unnatural for many hardware engineers. Consequently, the commercial cost of formal verification is often high, making its use uncommon when compared to simulation.

1.3 Objectives of this Thesis

The foundation of this thesis is the observation that resolution of the level-event conflict inherent to digital MOS circuit design is not absolute. Existing notations for circuit design adopt either a level-oriented or an event-oriented methodology. Although both these methodologies are sufficient for describing circuit behaviour, hardware engineers are free to reason with either levels or events, and in practice use both, even within the same design style.

The objective of this thesis is to embrace both levels and events under a common umbrella, called proposition-oriented behaviour, and in doing so to offer hardware engineers a greater degree of freedom than is possible using either their level or event-oriented counterparts. This freedom is also intended to be practical, and a proposition-oriented verification program, Veraci, is therefore evolved and used to demonstrate the benefits of applying proposition-oriented behaviours to the field of asynchronous circuit design.

1.4 Structure of this Thesis

This thesis consists of eight chapters. Chapter 2 presents an overview of asynchronous design, including a summary of existing formal methods, notations, and tools from the literature. Chapter 3 introduces proposition-oriented behaviours and evolves a set-theoretic semantics for two notations over proposition-oriented behaviours. Chapter 4 describes a special type of finite automata called proposition automata, into which the proposition-oriented notations from Chapter 3 can be translated. Chapter 5 builds on Chapters 3 and 4 to define a verification procedure for proposition-oriented behaviours using Binary Decision Diagrams. Chapter 6 documents the implementation of this procedure in the verification program Veraci, which is then used to outline key benefits of proposition-oriented behaviours to the field of asynchronous design. Chapter 7 applies Veraci to the design and implementation of an asynchronous public-key cryptographic unit, MOD_EXP, and Chapter 8 concludes the thesis, including a discussion of possible further work.

Chapter 2
Background

2.1 Introduction

The design of asynchronous circuits is a diverse, well documented field, and research into techniques for asynchronous design is active in both academic and industrial institutions worldwide. The major contribution of this thesis is proposition-oriented behaviour, and the purpose of this chapter is therefore to develop a framework within which proposition-oriented behaviours can be related to previously published work.

This chapter comprises of two sections: Section 2.2 presents two different classifications of asynchronous design, and Section 2.3 overviews existing formal methods applicable to asynchronous design. Although neither of these sections is exhaustive in its coverage of published work, it is hoped that sufficient material has been referenced to relate proposition-oriented behaviours to the literature. Further information on asynchronous design can be found in survey papers by Sutherland [97], Hauck [45], Davis [29, 30], and a generic survey of formal methods in hardware design can be found in Gupta [43]. Reference books on asynchronous design and its associated formal methods have also been published by both Springer-Verlag [16] and Wiley [75].

2.2 Asynchronous Design Styles

There is no unique way to design an asynchronous circuit. Different styles of asynchronous design can be classified according to different metrics, each of which quantifies certain design assumptions or architectural techniques that are adopted by the underlying circuit. The purpose of this section is to present two alternative classifications of asynchronous design, and to summarise some practical differences between the elements of each of these two classifications.

2.2.1 Modes of Operation

Global clocking permits synchronous designers to assert that chip function be sequential with respect to a quantised timeline. Asynchronous circuits lack such a quantised timeline, and must therefore generalise circuit operation according to different *modes*, each of which equates to a set of assumptions about the possible orders in which events can occur. Once a mode of operation is defined, it is possible to restore sequential function using further circuit-level techniques such as handshake protocols.

Fundamental Mode

An asynchronous circuit is said to be operated in fundamental mode if the time between input changes is no shorter than the maximum response time of that circuit to any input change [102]. Fundamental mode operation relates closely to synchronous design where the terms *setup* and *hold*-time are used to assert that a flip-flop has sufficient time to latch its input data value on each rising clock edge.

Speed Independent

An asynchronous circuit is said to be speed independent if its correct operation depends only on the assumption that wire delays are zero. Speed independence was first formalised by Muller in 1955 [73]. The only way to find out whether a speed independent circuit has finished a computation is to have it indicate completion on one of its outputs, often referred to as an *acknowledge*.

Semi-Modular

Semi-modular circuits form a proper subset of speed independent circuits in which no gate output can ever glitch. Semi-modularity can be formalised as the assertion that excited gate outputs can only become quiescent if they also change value [74]. Semi-modularity is useful since it quantifies a circuit-level property that can be checked independently of any functional requirements for that circuit. Semi-modularity also relates closely to hazard freedom [30].

Delay Insensitive

An asynchronous circuit is said to be delay insensitive if its correct operation depends neither on the delay of any gate nor the delay on any wire [68, 101]. Delay insensitive circuits form the most robust class of asynchronous circuit since their correctness is guaranteed for arbitrary gate and wire delay. However, delay insensitive circuits cannot be built using conventional single output logic gates alone, and require a more complex set of primitive

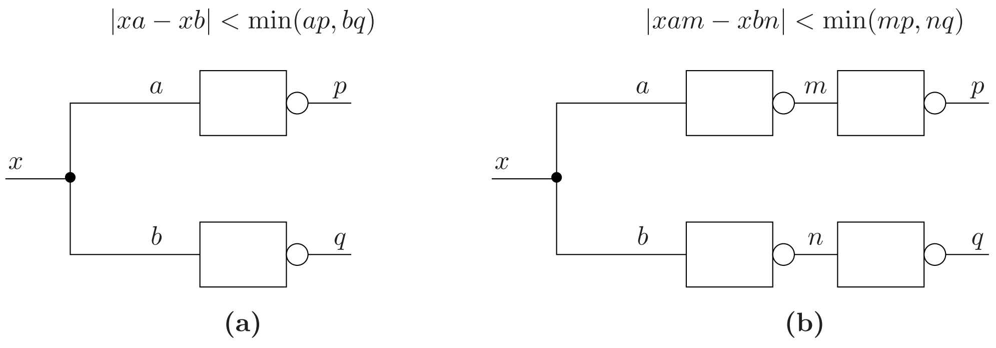


Figure 2.1: Isochronic forks of depth one (a), and two (b).

components, some of which have more than one output [35, 59]. Physical construction of these primitive components using single output logic gates can only be achieved by asserting alternative modes of operation internally.

Quasi-Delay Insensitive

An asynchronous circuit is said to be quasi-delay insensitive if its correct operation depends only on the existence of one or more *isochronic forks* [60]. A fork is said to be isochronic if the difference in delay on each branch is no more than the minimum delay of the two components to which each branch leads, see Figure 2.1(a) [5]. Quasi-delay insensitive operation is superior to the assumption that wire delay be zero since isochronic forks can be validated using static timing analysis tools, whereas zero gate delay is in practice impossible to achieve. However, quasi-delay insensitive circuits in which every fork is isochronic are operationally equivalent to speed independent circuits, and speed independent operation can be modelled mathematically using a smaller state space than quasi-delay insensitive circuits.

Q*DI

Q*DI circuits generalise the class of quasi-delay insensitive circuits to include isochronic forks of arbitrary depth [7]. For example a Q^{2}DI circuit asserts the existence of one or more isochronic forks of depth 2, see Figure 2.1(b). Assertion of a Q*DI mode of operation can simplify circuit design considerably and yet the underlying isochronic forks can all still be validated using static timing analysis tools.

2.2.2 Handshake Protocol Schemes

A digital circuit is distinct from an analogue circuit in that the voltage on every wire is assumed to be at one of two discrete levels: Logic High or Logic Low. The transmission of

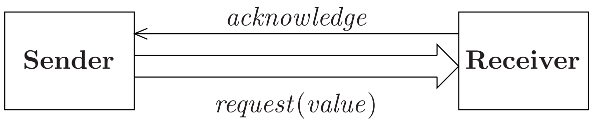


Figure 2.2: A generic handshake protocol.

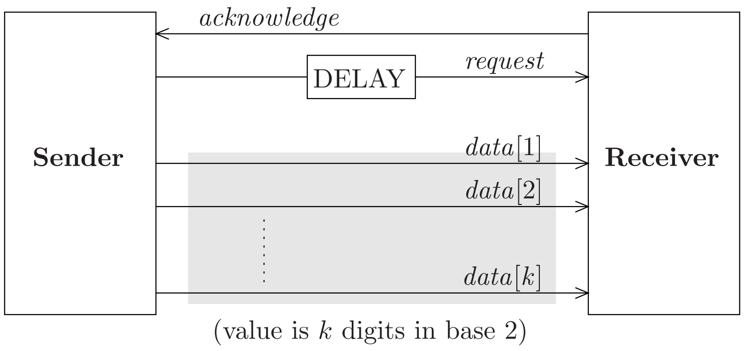


Figure 2.3: Bundled-data encoding.

data along a digital wire requires that these two levels be used to indicate both value and sequence: value is required to give data meaning and sequence is required to distinguish one data value from the next.

In synchronous design sequence is separated from value by a global drumbeat called a clock: every time the clock ticks the voltage level on each data wire is sampled, and this level used to denote a binary value from the set $\{0, 1\}$. In asynchronous design no global drumbeat is available, so value and sequence are often combined into a handshake protocol, see Figure 2.2. The purpose of a handshake protocol is to implement the point-to-point transmission of data from a Sender to a Receiver across a virtual channel. Handshake protocols can be classified according to two orthogonal metrics, their data encoding scheme and their signalling convention.

Data-Encoding Scheme

Data-encoding is the term used to describe how value is associated with a request in a handshake protocol. A data-encoding scheme may be either bundled-data or delay-insensitive. Bundled-data encoding splits request and value into separate wires [39]. Value is encoded as in a synchronous circuit using k wires to denote a k-bit number, and request is encoded using a dedicated request wire, *request*, see Figure 2.3.

Bundled-data encoding requires the explicit insertion of delay in the request wire to ensure that a request is never received before the bundled value is valid. Conversely, delay-

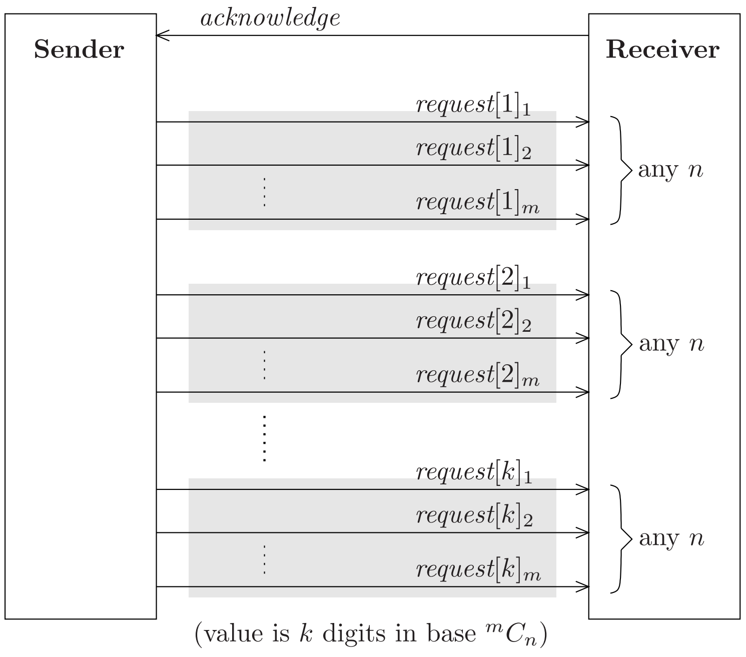


Figure 2.4: Delay insensitive *n*-of-*m* encoding.

insensitive encoding makes value implicit in the request and no delay insertion is therefore required. Delay-insensitive encoding can be described using an n-of-m notation [38, 105] to indicate that the transmission of data consists of making n requests out of a possible m, see Figure 2.4. Since there are ${}^mC_n$ ways to make n requests out of m, n-of-m encoding can be likened to communication in base ${}^mC_n$. The most common delay insensitive encoding is 1-of-2 or *dual-rail*, in which two request wires are used to send a single bit of data [12].

Although delay-insensitive encoding does not require the explicit insertion of delay, the use of multiple request lines to send a single data value necessitates completion detection to determine when all these requests have arrived at the Receiver.

Signalling Convention

Signalling is the term used to describe a sequence of actions sufficient to implement a single data transfer across a channel using digital wires. A signalling convention may be either two-phase or four-phase. Four-phase signalling associates handshake actions with voltage levels, whereas two-phase signalling associates handshake actions with transitions in voltage levels, see Figure 2.5.

Every four-phase handshake consists of two two-phase handshakes: an ‘asserting’ handshake and a ‘clearing’ handshake. In this sense a four-phase handshake can be likened to the alternating presence and absence of data in a channel, whereas a two-phase handshake can be likened to the transient transmission of data using transitions.

Although two-phase handshaking uses fewer transitions than four-phase handshaking, this efficiency need not translate to a faster implementation: transistors are level-sensitive devices, and a transition-sensitive handshake protocol therefore requires dedicated circuitry to alternate level-sensitivity from one handshake to the next. However, if interconnect delay is dominant, as for example in inter-chip bus communication, then delay-insensitive two-phase handshaking can offer significant benefits over alternative four-phase handshaking schemes [3].

Both two and four-phase signalling assume that every wire is driven by a unique logic gate. Single-track signalling [6] combines request and acknowledge actions onto a single tri-state wire, alternately driven by the Sender and Receiver, see Figure 2.6. Single-track handshaking can be viewed as either two-phase or four-phase since it combines the transition efficiency of two-phase signalling with the level-sensitivity of four-phase signalling. Single-track bundled-data handshaking has been successfully used by Sun Microsystems to develop a high performance asynchronous design style called GasP [96].

2.3 Formal Methods in Asynchronous Design

A technique is considered formal if its meaning and criteria for use are bounded by rules that obey the principles of reason. The use of formal methods in hardware design is motivated by a desire to eliminate human error and improve designer productivity through use of unambiguous notations that prevent mis-communication between designers.

This section begins by introducing some generic formal terminology, and by overviewing some generic formal methods called logic systems. This section continues to classify further formal methods in asynchronous design according to two generic concepts: sequence and concurrency. The motivation behind this classification is to observe that most notations for concurrency build directly on a simpler notation which reasons over sequential behaviours alone. Furthermore, this precipitation of sequence and concurrency also approximates the distinction between notations that have been applied to synchronous design from notations that have been applied to asynchronous design: Asynchronous circuits do not enforce sequence according to a global clock and concurrency is often fine-grained and

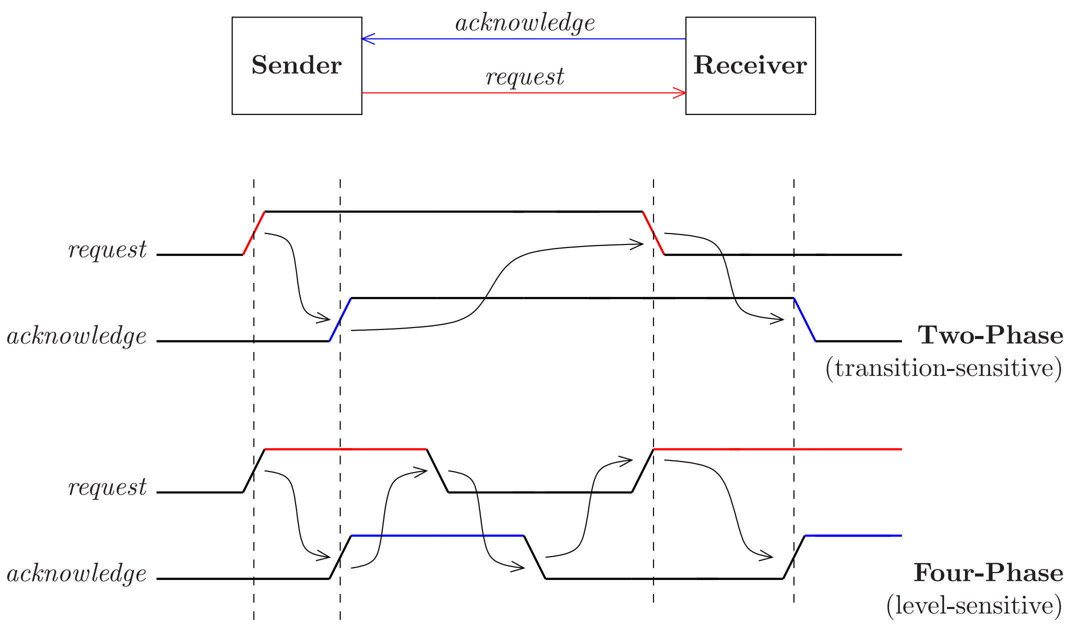


Figure 2.5: A comparison between two and four-phase signalling.

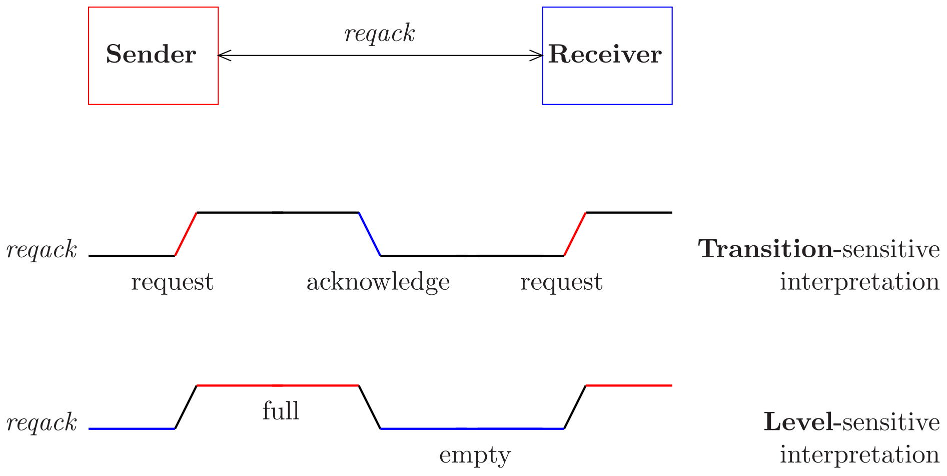

Sender
Receiver
reqack
reqack
request
acknowledge
request
Transition-sensitive interpretation
reqack
full
empty
Level-sensitive interpretation

Figure 2.6: Single-Track signalling.

explicit in circuit functionality. Conversely, it is well-known that synchronous circuits can always be abstracted to sequential machines of either the Moore or Mealy type [113].

This section concludes by summarising some different notions of equivalence between formal models, and by introducing a data-structure called Binary Decision Diagrams, whose application to the field of formal verification is discussed further in Chapter 5.

2.3.1 Problem Dimensions

The application of formal methods to hardware design typically consists of three basic problem dimensions: specification, model, and satisfaction criterion.

Specification. A specification, S, is a formal definition of a circuit property or behaviour described using a formal notation. Formal notations differ from informal notations in that any expression given in a formal notation has an associated mathematical meaning or *semantics*.

Model. A model, M, is a formal notation from which the behaviour of a circuit can be described mathematically. Given any circuit C, the model $M(C)$ of C denotes an abstract entity which mimics the physical behaviour of C, but which can also be reasoned with mathematically.

Satisfaction criterion. A satisfaction criterion is a mathematical relationship between specification and circuit model. A satisfaction criterion may be used for verification, where a circuit model and specification are shown to 'conform' according to a certain satisfaction criterion. A satisfaction criterion may also be used for synthesis to demonstrate that the synthesised circuit will function as specified. Satisfaction criteria can be categorised into two general types:

- $M(C) \Rightarrow S$. The circuit model *implies* the specification.
- $M(C) \equiv S$. The circuit model is *equivalent* to the specification.

2.3.2 Logic Systems

Logic is generally regarded as the study of the principles of reason. A logic system consists of a formal language and a set of axioms and rules for deducing proofs. A logic system is *sound* if all things provable in it are logically true. Conversely, a logic system is *complete* if all logically true formulas within it are provable. Logic is central to all formal methods

		b		
	$a \wedge b$	0	Φ	1
	0	0	0	0
a	Φ	0	Φ	Φ
	1	0	Φ	1

Table 2.1: Truth table for the ternary AND operator.

in that it provides the foundation on which both formal models and satisfaction criteria can be described. A detailed introduction to logic systems can be found in Chang and Lee [19].

Propositional Logic

Propositional logic is a notation of reasoning with the boolean constants True (T) and False (F). Propositional logic assumes the existence of an infinite set of propositional *variables*, each of which may take on either of these two constant values. Formulae in propositional logic are constructed from propositional variables and the constants T,F using the boolean operators AND ($\wedge$), OR ($\vee$), and NOT ($\neg$). Propositional logic is both sound and complete.

Ternary Logic

Ternary logic extends propositional logic to reason with three-valued truths [16]. Ternary logic begins with the notion of the three constant values False (0), Uncertain (Φ), and True (1). Ternary logic extends each of the boolean operators in propositional logic to apply to these three constant values, see for example Table 2.1. Ternary logic is significant to asynchronous design since it can be efficiently used to determine the outcome of an Extended Multiple Winner analysis on gate networks, see "Gate Networks" in Section 2.3.4.

Quaternary Logic

Quaternary logic is an extension of propositional logic proposed by Gaubatz [40]. Quaternary logic asserts existence of the four constant values, Low (ll), Rising (lh), High (hh), Falling (hl), and extends each of the boolean operators in propositional logic to apply to these four constant values, see for example Table 2.2.

Quaternary logic is significant because it treats both levels and events as explicit logic constants. However, given any pair of different levels it is always possible to infer the intermediate edge and in this sense quaternary logic offers no theoretical benefit over propositional logic in the context of circuit design.

			b		
	$a \wedge b$	ll	lh	hh	hl
	ll	ll	ll	ll	ll
a	lh	ll	lh	lh	ll
	hh	ll	lh	hh	hl
	hl	ll	ll	hl	hl

Table 2.2: Truth table for the quaternary AND operator.

First-Order Logic

First-order logic is an extension of fixed-constant logics, including propositional logic, ternary logic, and quaternary logic, to permit reason over members of any non-empty universe of constant symbols. It introduces the concept of *functions* between any two universes, and *predicates* as special functions from any universe to the propositional universe $\{\mathsf{T}, \mathsf{F}\}$. First-order logic also permits universal ($\forall$) and existential ($\exists$) quantification over the elements in any universe. First-order logic is both sound and complete.

Higher Order Logic

Higher-order logic extends first-order logic so that function arguments can range over constants, functions and predicates. Higher-order logic is the most powerful of all logics, and all formal methods described in this chapter can be embedded in higher-order logic. However, higher-order logic is not sound. For example, consider the expression $P(P)$ where $P(x) \stackrel{\text{def}}{=} \neg(P(x))$. Higher-order logic is also incomplete and fully-automated proof of higher-order logic theorems therefore impossible.

Higher-order logic can be made sound by introducing a notion of *type* to each formula and requiring that valid formulae adhere to a set of rules referred to as a type system. The *HOL* system [42] is a mechanised tool for deriving proofs in typed higher-order logic. Since typed higher-order logic is still incomplete the use of HOL to derive a proof may require human assistance, although in practice this requirement is rare.

The application of higher-order logic to circuit design been discussed at length by Camilleri, Gordon and Melham [18].

Temporal Logics

Temporal logics extend first-order logic to reason over a notion of states in time. Linear Time Logics (LTL) assert that time is linear whereas Branching Time Logics (BTL) assert that time is branching [54, 58]. Linear and Branching Time Logics can be distinguished by considering a relation $<$ between pairs of states in time. If $<$ is a total order, meaning that for any two different states s, t in time, either $s < t$ or $t < s$, then time is said to be linear. If $<$ is a partial order, meaning that for any three states s, t, u in time with $t < s$ and $u < s$, either $t < u$ or $u < t$ or $u = t$, then time is said to be branching. Temporal logics reason over states in time using explicit temporal predicates:

- **Safety**, denoted $\Box P$. Means that P holds at all times in the future.
- **Liveness**, denoted $\Diamond P$. Mean P will eventually hold.
- **Precedence**, denoted $[P\mathbf{U}Q]$. Means that P will hold until Q does.

Temporal logics are less expressive than higher-order logic, but they are complete, and automated proof of temporal formulae is therefore possible. Computation Tree Logic (CTL) is a branching time logic defined by Clarke and Emerson that has received significant attention in the field of circuit design [21, 22, 32]. An efficient algorithm for the automatic proof of CTL formulae using Binary Decision Diagrams has been implemented in a tool called SMV [62]. Linear Temporal Logic has also been successfully applied to circuit design using a verification tool called SPIN [104].

2.3.3 Sequential Systems

State Graphs and Transition Systems

A state graph or labelled transition system is a labelled directed graph $G = \langle \Sigma, V, E \rangle$ with a finite set of vertices, V, and a finite set of labelled edges $E \subseteq V \times \Sigma \times V$ over a finite alphabet of symbols Σ. A state graph is a generic graphical model of sequential behaviour in which sequences of symbols in Σ can be visually equated to paths in a labelled directed graph. If $G = \langle \Sigma, V, E \rangle$ is a state graph and $(v_1, a, v_2) \in V \times \Sigma \times V$ then $v_1 \xrightarrow{a} v_2$ is often used as shorthand for the assertion that $(v_1, a, v_2) \in E$.

Finite Automata

Finite automata extend state graphs to include a notion of start and finish states. A finite automaton is a 5-tuple $M = \langle Q, \Sigma, S, T, F \rangle$ where Q is a finite set of states, Σ is a finite alphabet of input symbols, $S \subseteq Q$ is a set of start states, $F \subseteq Q$ is a set of finish states, and $T \subseteq (Q \times \Sigma) \times Q$ is a labelled transition relation. If $|S| = 1$ and T is injective then M is deterministic, otherwise M is non-deterministic. Finite state automata are often depicted by a labelled directed graph in which start states are identified by a free floating incoming arc, and finish states are identified by a double ringed vertex, see Figure 2.7.

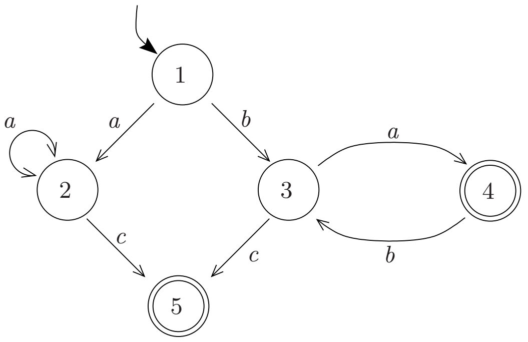


Figure 2.7: An example of a finite automaton.

If M is a finite automaton then any sequence of symbols matched by a path in M from an initial state to a finish state is said to be *accepted* by M. The set of all sequences accepted by M is the *language*, $L(M)$, of M. If $M = \langle Q, \Sigma, S, T, F \rangle$ is a non-deterministic finite automaton then M may always be translated into a deterministic finite automaton D with $O(2^{|Q|})$ states such that $L(M) = L(D)$. This translation is known as the powerset construction [112], and is important since it demonstrates that non-determinism does not increase the expressivity of finite automata.

Finite automata can be extend to include *empty* transitions, denoted by the special label $\varepsilon \notin \Sigma$. The purpose of an empty transition $s \xrightarrow{\varepsilon} s'$ is to permit the instantaneous transition from state s to state s'. The use of empty transitions can simplify finite automata considerably, however empty transitions do not increase the expressivity of finite automata since every finite automaton M can always be translated into a ε-free finite automaton D such that $L(M) = L(D)$ [112].

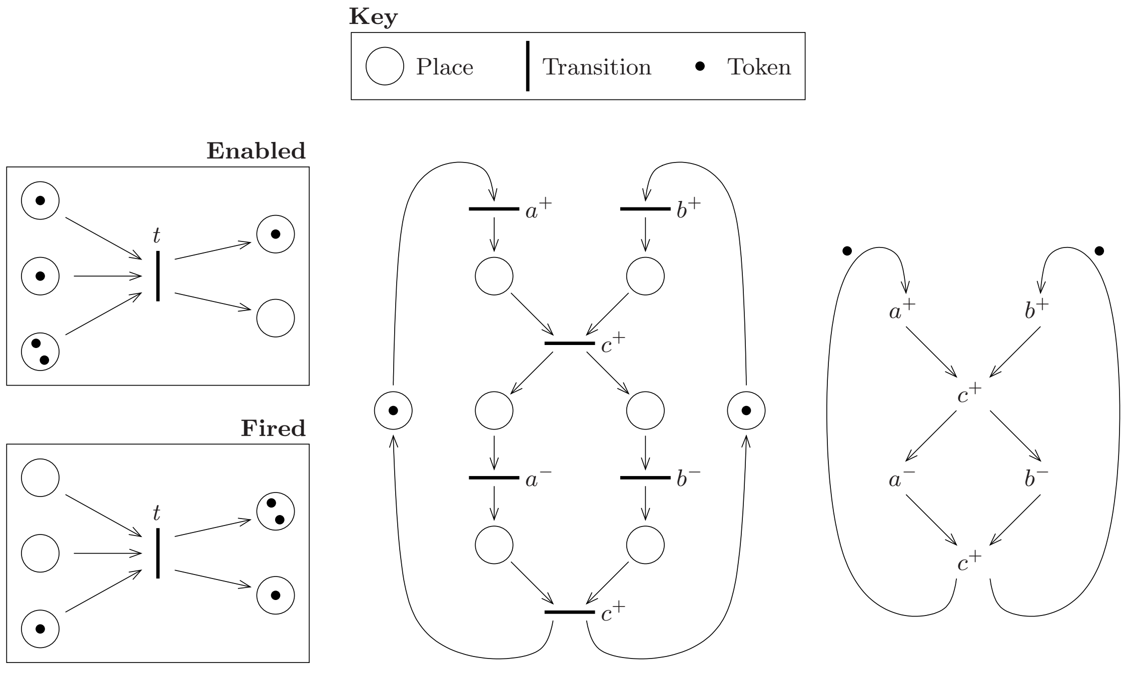
Finite automata and their associated languages have been extensively classified, modified and specialised to particular problems. For a detailed introduction to finite automata see Hopcroft and Ullman [48] or Booth [9].

Regular-Expressions

A regular-expression over a finite alphabet Σ is any expression R that can be formed from one or more applications of the following grammar:

$$\begin{array}{lcl} R & ::= & \varepsilon \mid a \mid RR \mid R+R \mid R^* \\ a & ::= & \text{any symbol in } \Sigma \end{array}$$

Regular-expressions are a generic textual notation for describing sequences of symbols from Σ. If R is a regular-expression then the language, $L(R)$ of R is a set of sequences



(a) transition firing **(b) Petri-net for a C-element** **(c) STG for a C-element**

Figure 2.8: A brief summary of Petri-nets and STGs.

from Σ. If $R = \varepsilon$ then $L(R) = \{\varepsilon\}$ contains the empty sequence, ε, and if $R = a$ then $L(R) = \{a\}$ contains the singleton sequence, a. $L(R_1R_2)$ contains sequences that are concatenations of sequences from R_1, R_2, and $L(R_1 + R_2)$ contains any sequence from either $L(R_1)$ or $L(R_2)$. $L(R^*)$ contains any sequence formed from the concatenation of zero or more sequences in $L(R)$. If a sequence $\sigma \in L(R)$ then R is said to *accept* σ.

Regular-expressions and finite automata have much in common. In particular, it can be shown that the language of every regular-expression is also the language of a finite automaton, and vice-versa [48].

2.3.4 Concurrent Systems

Petri-nets

A Petri-net is a directed graph $N = \langle P, T, F, m_0 \rangle$ with vertices $P \cup T$ formed from two disjoint sets P, T, and edges $F \subseteq (S \times T) \cup (T \times S)$. Vertices $v \in P$ are known as *places* and vertices $v \in T$ are known as *transitions*. The purpose of a Petri-net is to extend finite automata with an explicit notion of concurrency, represented by tokens. Each state of a Petri-net is defined by a marking function $m \in P \to \mathbb{N}$, denoting a number, $m(p)$, of tokens for each place $p \in P$. The initial state of N is defined by its initial marking, m_0.

A transition $t \in T$ is *enabled* in state m_i if all its predecessor places contain at least one token. If t is enabled then it may *fire*, causing one token to be removed from each of its predecessor places and causing one token to be added to each of its successor places, see Figure 2.8(a). Any valid sequence of transitions from the initial marking is called a feasible trace. The set of all feasible traces for N is known as the feasible trace-set of N.

A marking m is *reachable* if it can be obtained by a finite number of transitions from the initial marking. If all reachable markings have $m(p) \leq k$ for $k \in \mathbb{N}$ then N is said to be k-bounded. If N is k-bounded then there can only be a finite number of markings, and these markings can be used to construct a reachability state graph with paths equivalent to feasible traces in N.

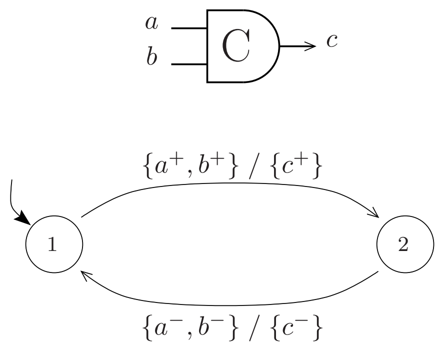
Petri-nets provide a very general model of concurrency that need not describe a digital circuit. The application of Petri-nets to circuit design is achieved by equating Petri-net transitions to *events* or *edges* on circuit wires. An event is a symbol, x, used to denote a level change on a single circuit wire, x. An edge is a qualified event, denoted by one of two symbols, x^+ or x^-. An interpreted Petri-net where transitions correspond to events or edges on circuit wires is called a Signal Transition Graph [20, 87]. STGs are often drawn in shorthand form, where places with a unique input and output transition are omitted, see Figure 2.8(b,c).

STGs are well suited to the synthesis of speed-independent circuits, see for example the synthesis tool Petrify [24]. In particular, the markings of a 1-bounded STG can be related to the logic values on each of the wires to which its transitions refer. If this relation is one-to-one then the STG is said to have Unique State Coding. If CSC does not hold, then it can be satisfied automatically by inserting new wires in the circuit to distinguish previously indistinguishable states [26].

A detailed introduction to Petri-nets, STGs, and their application to asynchronous design, verification and synthesis can be found in Kondratyev [53].

Burst-Mode Specifications

A burst-mode specification is a five-tuple $M = \langle X, Y, Z, z, \delta \rangle$ where X is a finite set of input actions, Y is a finite set of output actions, Z is a finite set of states, $z \in Z$ is an initial state, and $\delta \in Z \times (2^X \times 2^Y) \rightarrow Z$ is a labelled transition function between states in Z. The purpose of a burst-mode specification is to label each edge of a state graph



$$\begin{aligned}
X &= \{a^+, a^-, b^+, b^-\} \\
Y &= \{c^+, c^-\} \\
Z &= \{1, 2\} \\
z &= 1 \\
\delta &= \{ \quad (1, (\{a^+, b^+\}, \{c^+\}), 2), \\
&\qquad\quad (2, (\{a^-, b^-\}, \{c^-\}), 1) \ \}
\end{aligned}$$

Figure 2.9: An example of a burst-mode specification.

with a pair of actions $(x, y) \in 2^X \times 2^Y$, where $x \subseteq X$ denotes an input-burst and $y \subseteq Y$ denotes an output-burst, see for example Figure 2.9.

Burst-mode specifications extend finite automata to include concurrency by asserting that all actions in a burst may happen in any order. If a burst-mode specification $M = \langle X, Y, Z, z, \delta\rangle$ is in state $s \in Z$ with $(s, (x, y), s') \in \delta$, then after receiving input-burst x, M will perform output actions $a \in y$ in an unspecified order and proceed to state s'. Burst-mode specifications cannot accommodate non-determinism outside of a burst, and no input-burst can therefore be the subset of another input-burst leading from the same state.

Extended burst-mode specifications augment the burst-mode transition relation δ to permit the use of boolean guards on any transition. Both burst-mode and extended burst-mode specifications have received significant attention in the literature due to an algorithm for their efficient hazard-free synthesis into low latency fundamental mode circuits [79, 114, 115]. An implementation of this algorithm is available in the synthesis tool 3D.

Trace Structures

A trace structure is a triple $T = \langle A, B, X\rangle$ where $\mathbf{i}T \overset{\text{def}}{=} A$ is an alphabet of input symbols, $\mathbf{o}T \overset{\text{def}}{=} B$ is an alphabet of output symbols and $\mathbf{t}T \overset{\text{def}}{=} X$ is a set of sequences or *traces* over symbols in $\mathbf{a}T \overset{\text{def}}{=} A \cup B$. The purpose of a trace structure is to combine a set of traces and the symbols over which those traces are defined into a single entity. Trace structures can be applied to circuit design by equating symbols in $\mathbf{a}T$ to events on the wires of a circuit.

Trace-expressions are an extension of regular-expressions to apply to trace structures. Trace-expressions differ from regular-expressions in that they introduce an explicit concurrency construct $\|$ known as *parallel composition* [31] or *weave* [34]. The purpose of

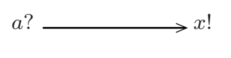
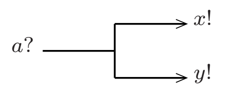
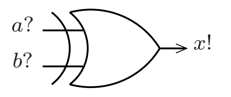
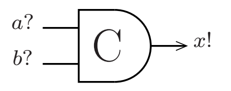
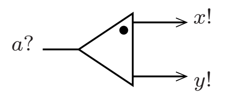
Name	Symbol	Trace-expression
Wire		**pref** $[a?; x!]$
Fork		**pref** $[a?; (x! \parallel y!)]$
Merge		**pref** $[(a?; x!) \mid (b?; x!)]$
Join		**pref** $[(a?; x!) \parallel (b?; x!)]$
Toggle		**pref** $[a?; x!; a?; y!]$

Figure 2.10: Example delay insensitive components and their trace-expressions.

parallel composition is to take two trace-expressions S, T and construct a trace structure that contains all possible interleavings of traces in both S and T. If t is a sequence of symbols from alphabet Σ and $P \subseteq \Sigma$, then $t \upharpoonright P$ denotes the sequence remaining after all symbols not in P have been removed from t. A trace $t \in \mathbf{t}(S \parallel T)$ if and only if $t \upharpoonright \mathbf{a}S \in \mathbf{t}S$ and $t \upharpoonright \mathbf{a}T \in \mathbf{t}T$. A generic trace-expression grammar is as follows:

$$T \ ::= \ \epsilon \ \big|\ x? \ \big|\ x! \ \big|\ [T] \ \big|\ T_1; T_2 \ \big|\ T_1 | T_2 \ \big|\ T_1 \parallel T_2 \ \big|\ \mathbf{pref}\, T$$

$x?$ denotes an input action, $x!$ denotes an output action, $[T]$ denotes arbitrary repetition of traces in $\mathbf{t}T$, and $\mathbf{pref}\, T$ denotes prefix-closure on traces in $\mathbf{t}T$. Trace structures have been used extensively in the design of delay insensitive circuits [36, 80, 85, 93], and delay insensitivity of a trace structure has been formalised by both Ebergen [34] and Dill [31]. Some examples of delay insensitive components and their associated trace-expressions are shown in Figure 2.10. Automated verification procedures based on trace structures have also been developed by both Dill [31] and Ebergen [33].

Communicating Processes

Communicating processes are an abstract model of concurrency built on the concept of processes interacting through channels or actions. Notations for communicating processes have been proposed independently by both Hoare and Milner.

According to Milner a Calculus of Communicating Systems (CCS) is a formal model defined in terms of labelled transition systems [64]. A CCS expression P is considered to be an atomic entity capable of performing sequences of actions over a definite alphabet, denoted αP. The performance of any action $a \in \alpha P$ is regarded as a labelled transition $P \xrightarrow{a} Q$ that transforms a CCS expression P into a new CCS expression Q.

CCS differs from trace structures in that each observable action $a \in \Sigma$ has two forms: positive, denoted a, and negative, denoted $\overline{a}$. Performance of an action a equates to synchronisation of two CCS expressions $a.P$ and $\overline{a}.Q$, denoted by the transformation $a.P \,[\!]\, \overline{a}.Q \xrightarrow{a} P \,[\!]\, Q$. A CCS action a therefore models a two way handshake between processes P and Q rather than the blind sending of an event a along a wire in a circuit. CCS actions may also embed the transmission of value, denoted $a(v_1, v_2, \ldots, v_n).P$. Embedding of value in actions permits CCS to directly model the transfer of data along channels, an ability with immediate application to asynchronous design. CCS also makes a distinction between hidden and observable actions by introducing the notion of a hidden transition, denoted by the special action τ. A cut-down grammar for CCS expressions is as follows:

$$P \;::=\; \mathrm{NIL} \;\big|\; P + P \;\big|\; a(v_1 \ldots v_n).P \;\big|\; \overline{a}(v_1 \ldots v_n).P \;\big|\; \tau.P \;\big|\; P \,[\!]\, P \;\big|\; P \setminus a$$

NIL denotes the terminal expression that can't do anything, $a.P$ denotes an expression capable of performing the positive action a, $\overline{a}.P$ denotes an expression capable of performing the negative action $\overline{a}$, and $\tau.P$ denotes an expression capable of performing a hidden action τ. $P + Q$ denotes an expression that may behave either as P or Q, and $P \,[\!]\, Q$ denotes the concurrent execution of P and Q. For example, $x.P + y.Q$ differs from $x.P \,[\!]\, y.Q$ in that $x.P + y.Q \xrightarrow{x} P$ whereas $x.P \,[\!]\, y.Q \xrightarrow{x} P \,[\!]\, y.Q$. $P \setminus a$ makes $a, \overline{a}$ actions hidden in expression P. For example, $(a.P \,[\!]\, \overline{a}.Q) \setminus \{a\} \xrightarrow{\tau} P \,[\!]\, Q$. An interactive environment for visualising CCS expressions and for deciding equivalences between them can be found in the Edinburgh Concurrency Workbench [67].

The π-calculus is a successor to CCS that introduces a notion of *mobile* connectivity between processes [66]. The π-calculus and its derivatives are an active area of research. However it should be noted that an integrated circuit never dynamically changes its structure and so the π-calculus has so far offered little advantage over CCS in the field of circuit design.

Communicating Sequential Processes (CSP) are an alternative notation for communicating processes due to Hoare [11, 46]. Language constructs in CSP have much in common with CCS, and every CSP expression P is also defined over a definite alphabet αP. However, CSP differs formally from CCS in that the meaning of a CSP expression is set-theoretic and is not based on labelled transition systems: in CSP, each expression P is in one-to-one correspondence with a set $F = \mathit{failures}(P)$ of pairs $\langle s, X\rangle$ such that the following conditions apply:

1. s is any sequence over αP, and $X \subseteq \alpha P$.
2. $\langle \varepsilon, \emptyset\rangle \in F$.
3. If $\langle st, \emptyset\rangle \in F$ then $\langle s, \emptyset\rangle \in F$.
4. If $\langle s, Y\rangle \in F$ and $X \subseteq Y$ then $\langle s, X\rangle \in F$.
5. If $a \in \alpha P$ and $\langle s, X\rangle \in F$ and $\langle sa, \emptyset\rangle \notin F$ then $\langle s, X \cup \{a\}\rangle \in F$.

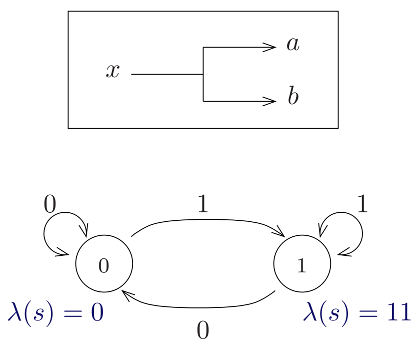
The purpose of $\langle s, X\rangle \in \mathit{failures}(P)$ is to assert that after performing the sequence of actions denoted by s, P will *reject* any of the actions contained in X. If $\langle s, \emptyset\rangle \in \mathit{failures}(P)$ then s is is a valid *trace* of P. The purpose of Condition 2 is to assert that the empty sequence is always a valid trace, and the purpose of Condition 3 is to assert that valid traces are prefix-closed. The purpose of Condition 4 is to assert that any subset of rejected actions is also rejected, and the purpose of Condition 5 is to assert that impossible events are always rejected.

Further to CSP, CCS, and the π-calculus, a communicating process notation called Communicating Hardware Processes (CHP) has also been defined by Martin. CHP has much in common with CSP however it has no underlying formal semantics, and is defined qualitatively in terms of translation into quasi-delay insensitive circuits [60].

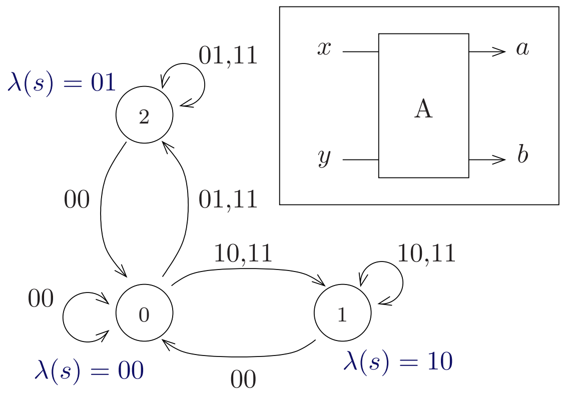
CSP and CCS also serve as a generic platform on which other process-oriented notations can be evolved. For example, the Rainbow environment [4] is a multi-lingual system whose underlying semantics is defined in terms of a process notation similar to CCS and CSP. Rainbow is aimed specifically at the design of asynchronous circuits, and each of its front-end languages targets different aspects of asynchronous design. For example, the Green language offers a data-flow oriented notation, whereas the Yellow language offers a control-flow oriented language.

Gate Networks

A gate network is a representation of concurrent behaviours as a set of independent state graphs. Each vertex in each state graph uniquely determines a vector of binary output



$\delta(s,x)$		x		$\lambda(s) = ab$
		0	1	
s	0	{0}	{1}	00
	1	{0}	{1}	11



$\delta(s,xy)$		xy				$\lambda(s) = ab$
		00	01	10	11	
	0	{0}	{2}	{1}	{1,2}	00
s	1	{0}	{0}	{1}	{1}	10
	2	{0}	{2}	{0}	{2}	01

(a) Isochronic Fork **(b) Arbitration Element**

Figure 2.11: Example gate network module definitions.

values, and each edge of each state graph is labelled with a vector of binary input values. State graphs are coupled together by virtue of the fact that an output value of one state graph may be an input value to another state graph. The purpose of a gate network is as a generic model of a circuit in which each primitive component or gate is modelled by an independent state graph, and connectivity between components is determined by binary input and output values on wires.

According to Brzozowski and Zhang [15], a gate network is a pair $\langle N, P \rangle$ where $N = \{M_1, \ldots, M_n\}$ is a set of independent modules and K is a connectivity function. Each module M_i is a five-tuple $\langle S_i, X_i, Z_i, \lambda_i, \delta_i \rangle$, where S_i is a unique set of internal states, X_i is a unique set of binary input variables, Z_i is a unique set of binary output variables, $\lambda_i \in S_i \to \{0,1\}^{|Z_i|}$ is an output value function, and $\delta_i \in (S_i \times \{0,1\}^{|X_i|}) \to (2^{S_i} - \{\emptyset\})$ is an excitation function such that for every $(s, z) \in (S_i \times \{0,1\}^{|X_i|})$ either $\delta_i(s, v) = \{s\}$ or $s \notin \delta_i(s, v)$. A connectivity function is a bijection $K \in (\bigcup_i X_i \to \bigcup_i Z_i)$ connecting each module output to a unique module input.

Each module $M_i \in N$ can be likened to a state graph where vertices are states in S_i and where edges are labelled with binary value assignments to each input variable in X_i. For each state-input pair $(s, v) \in S_i \times \{0,1\}^{|X_i|}$, the excitation function δ_i either asserts stability, $\delta_i(s, v) = \{s\}$, or asserts excitation, $s \notin \delta_i(s, v)$. Some example module definitions are shown in Figure 2.11.

The state of a gate network is defined by a vector $s \in (S_1 \times \cdots \times S_n)$ of state assignments to each of its internal modules. Transition from one gate network state to the next is determined by an *execution model* relation $R \subseteq (S_1 \times \cdots \times S_n) \times (S_1 \times \cdots \times S_n)$. An execution model may be either Single or Multiple-Winner [16]. A Single-Winner execution model asserts that at most one module can change state per gate network transition, whereas a Multiple-Winner execution model asserts that any number of modules can change state per gate network transition.

The Extended Multiple-Winner execution model is an extension of the Multiple-Winner execution model to apply to gate networks where the values on inputs and outputs can be either 0, Uncertain (Φ), or 1. Analysis of gate networks under an Extended Multiple-Winner execution model can be efficiently implemented using a technique called ternary simulation [16].

Process Spaces

Process spaces are a model of concurrency built on an abstract notion of execution that makes no explicit reference any notion of sequence [76]. In process space theory, a process P over $\mathcal{E}$ is a pair (X, Y) where $\mathcal{E}$ is an abstract set of executions and $X \cup Y = \mathcal{E}$. A process $P = (X, Y)$ represents a contract between a device and its environment. This contract asserts that the device only accesses executions in $\mathbf{as}P \overset{\text{def}}{=} X$, and that the environment only accepts executions in $\mathbf{at}P \overset{\text{def}}{=} Y$. For each process $P = (X, Y)$ the sets $\mathbf{g}P \overset{\text{def}}{=} X \cap Y$, $\mathbf{r}P \overset{\text{def}}{=} \overline{Y}$, and $\mathbf{e}P \overset{\text{def}}{=} \overline{X}$ denote a partitioning of $\mathcal{E}$ into three parts: $\mathbf{g}P$ denotes *goals*, which are valid executions conforming to the device-environment contract; $\mathbf{r}P$ denotes *rejects*, which are executions that must be avoided by the environment; and $\mathbf{e}P$ denotes *errors*, which are executions that must be avoided by the device.

If P and Q are processes over $\mathcal{E}$ then the concurrent composition of P and Q can be expressed in one of two ways:

1. $P \times Q \overset{\text{def}}{=} ((\mathbf{as}P \cap \mathbf{as}Q),\ (\mathbf{at}P \cap \mathbf{at}Q) \cup (\mathbf{e}P \cap \mathbf{r}Q) \cup\ (\mathbf{r}P \cap \mathbf{e}Q))$

2. $P \oplus Q \overset{\text{def}}{=} ((\mathbf{as}P \cap \mathbf{as}Q) \cup (\mathbf{e}P \cap \mathbf{r}Q) \cup\ (\mathbf{r}P \cap \mathbf{e}Q),\ (\mathbf{at}P \cap \mathbf{at}Q))$

Both $P \times Q$ and $P \oplus Q$ model concurrency between two contracts by asserting that executions acceptable to both processes remain acceptable, and that executions accessible to both processes remain accessible. $P \times Q$ and $P \oplus Q$ differ in their treatment of executions that are errors to one process but rejects to the other. $P \times Q$ asserts that such executions become errors whereas $P \oplus Q$ asserts that such executions become rejects. This difference equates to an assertion regarding whether it is the devices or the environments that are interacting: in the case of $P \times Q$ it is the devices that are interacting and an error for either device is an error for the composition. In the case of $P \oplus Q$ it is the environments that are interacting and a reject by either environment is a reject by the composition.

Process spaces are significant because they present a purely set-theoretic model of processes and concurrency that is void of sequence, connectivity, wires, events, levels, inputs or outputs. Process spaces are discussed in detail by Negulescu [76, 77] where they are used as the foundation for an automated verification program called Firemaps.

DI-Algebra

The DI-algebra [49] is a notation for describing concurrent processes that can *only* communicate with each other across a delay insensitive interface. A delay insensitive interface is an interface that may delay any signal for an arbitrary amount of time. DI-algebra has an underlying failure-set semantics similar to the trace structures of Dill [31] and the Communicating Sequential Processes of Hoare [46].

DI-algebra begins with the assertion of a finite set $A = ((I \times \{?\}) \cup (O \times \{!\}))$ of symbols where $a? \in A$ denotes an input symbol and $a! \in O$ denotes and output symbol. A DI-specification is a set of possibly recursive concurrent process equations $X_i = E_i$ where X_i are *process variables* and E_i are *process expressions* formed from one or more applications of the following grammar:

$$\begin{array}{rcl} E & ::= & \perp \;\big|\; X_i \;\big|\; E \sqcap E \;\big|\; [\,\mathit{choice}\,] \\ \mathit{choice} & ::= & (\mathit{guard} \rightarrow E) \;\big|\; (\mathit{guard} \rightarrow E) \;\Box\; \mathit{choice} \\ \mathit{guard} & ::= & \mathit{skip} \;\big|\; a? \;\big|\; a! \end{array}$$

$\perp$ denotes the degenerate or *chaos* process that can do anything. $E \sqcap F$ denotes a process that can non-deterministically choose to behave either as E or as F. $[\mathit{choice}]$ denotes the guarded-choice between a set of guards of the form $(\mathit{guard} \rightarrow F)$. The guard $(a? \rightarrow F)$ matches an input on a and then behaves as F, the guard $(a! \rightarrow F)$ outputs an a and then behaves as F, and the guard $(\mathit{skip} \rightarrow F)$ can always be matched. A process $[\mathit{choice}]$ that is able to match one of its guards must eventually do so, however if more than one guard can be matched then that process can non-deterministically select any one of its

Interpretation	Explanation
$\top$	Reject
∇	Process must output something
$\square$	Quiescent
Δ	Process must receive something
$\bot$	Error

Table 2.3: State interpretations in the XDI model.

matching guards. Two common shorthands for guarded-choice are $a?; E \overset{\text{def}}{=} [a? \rightarrow E]$ and $a!; E \overset{\text{def}}{=} [a! \rightarrow E]$.

DI-specifications assert a set of laws that must be adhered to by their underlying semantics. For example, $a?; b?; E = b?; a?; E$ since a delay insensitive interface implies that the order in which inputs arrive cannot be determined. DI-algebra also defines a partial *refinement* ordering $\sqsupseteq$ between processes such that $P \sqsupseteq Q$ if and only if $P \sqcap Q = Q$. If $P \sqsupseteq Q$ then P can not be distinguished from Q by any environment, and may therefore replace Q in any implementation. An important law of refinement is $E \sqsupseteq \bot$ which asserts that any process E is a refinement of the chaotic process $\bot$. In this sense $\bot$ is a least element of $\sqsupseteq$ and can be used to solve the recursive $X_i = E_i$ equations that form a DI-specification by computing least fixed-points with respect to $\sqsupseteq$.

A DI-specification differs from other concurrent notations in that concurrency is *implicit* in the underlying algebra. For example, consider the join element shown in Figure 2.10 with trace-expression $M = \textbf{pref}\,[(a?; x!) \parallel (b?; x!)]$. An equivalent specification in DI-algebra need only describe an "example" execution sequence for the join, such as $M = a?; b?; x!; M$, with the underlying algebraic laws performing closure on any concurrency that must ensue when the interface to M is delay-insensitive.

The Extended Delay Insensitive (XDI) Model is an alternative model of delay insensitive processes in which the underlying semantics are based on state graphs [106, 107]. Although it is also possible to translate DI-specifications into finite automata [57], the XDI Model extends DI-algebra further by augmenting each process state with one of the interpretations shown in Table 2.3. The result of this extension is an ability to treat process and environment with a symmetry not possible using a failure-set semantics of DI-algebra. This symmetry permits the XDI model to express progress constraints for both the environment and the process where failure-sets can only express progress constraints for the process.

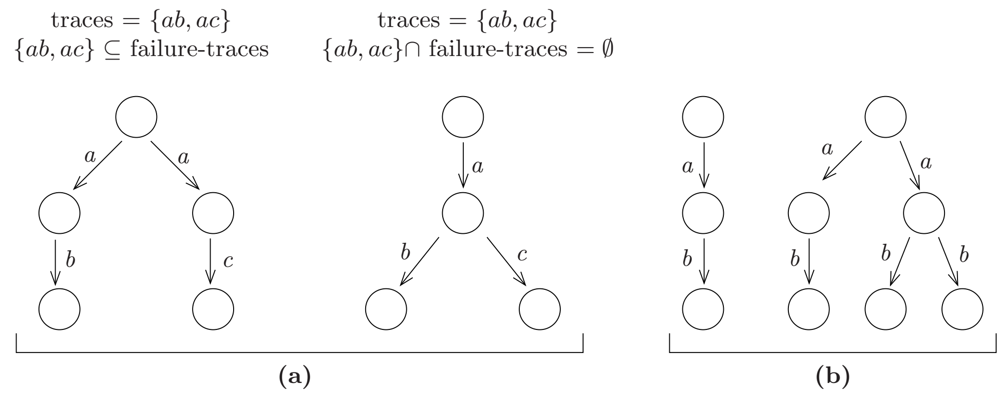

traces = $\{ab, ac\}$
$\{ab, ac\} \subseteq$ failure-traces
traces = $\{ab, ac\}$
$\{ab, ac\} \cap$ failure-traces = $\emptyset$
a
b
c
(a)
(b)

Figure 2.12: (a) Non-deterministic choice (b) Structural inequality.

2.3.5 Equivalences

An equivalence is a relationship between two entities. In Section 2.3.1 equivalence between model and specification was outlined as a type of satisfaction criterion. Equivalences are also necessary for state minimisation [1] since two states of a model can only be combined into one if they are known to be equivalent. The purpose of this section is to present a brief outline of those equivalences most relevant to the formal methods in hardware design. Further information on equivalence, and a formal definition of the equivalence relations that follow can be found in Shiple [92].

Trace equivalence

Trace equivalence asserts that two processes are equivalent if their corresponding sets of *valid* executions are identical. Trace equivalence is applicable to all the models discussed here, but it is also the weakest in that it asserts the largest number of processes in each equivalence class.

Failure-trace equivalence

Failure-trace equivalence asserts that two processes are equivalent if their corresponding sets of both *valid* and *invalid* executions are identical. Failure-trace equivalence extends trace-equivalence to distinguish between the two different types of non-deterministic choice shown in Figure 2.12(a).

Observational equivalence

Observational equivalence asserts that two processes are equivalent if neither can be distinguished from the other by any number of *experiments* at their external interface. Observational equivalence enables hierarchical abstraction of circuit models by permitting certain actions or symbols to be considered hidden or internal.

Bisimulation equivalence
Bisimulation equivalence asserts that two processes are equivalent if each process can *simulate* the other action for action. Strong bisimulation requires that hidden actions are included in these simulations, whereas weak bisimulation requires only that simulations apply to visible actions [65].

Structural equivalence
Structural equivalence or graph isomorphism is the strongest form of equivalence which asserts that two models are the same except for a renaming of variables or terms. Structural equivalence is in most cases too strong to be of any practical use, see for example Figure 2.12(b).

2.3.6 Efficient Implementation using Binary Decision Diagrams

A Binary Decision Diagram (BDD) over a finite set of variables V is a binary tree B formed by one or more applications of the following grammar [13, 14]:

$$\begin{array}{rcl} B & ::= & \mathsf{T} \;\big|\; \mathsf{F} \;\big|\; \mathsf{node}(v, B, B) \\ v & ::= & \text{any element of } V \end{array}$$

If $B = \mathsf{node}(v, P, Q)$ then $\mathsf{var}(B) \stackrel{\text{def}}{=} v$, $\mathsf{then}(B) \stackrel{\text{def}}{=} P$, and $\mathsf{else}(B) \stackrel{\text{def}}{=} Q$. The purpose of a BDD B is to represent a boolean expression over the variables in V. This representation equates boolean value assignments to V with paths from root to leaf in B. If a particular value assignment results in a path to T then B evaluates to T and if a particular value assignment results in a path to F then B evaluates to F. A BDD B can be converted to a boolean expression $[\![B]\!]$ recursively as follows:

$$\begin{array}{rcl} [\![\mathsf{T}]\!] & \stackrel{\text{def}}{=} & \mathsf{T} \\ [\![\mathsf{F}]\!] & \stackrel{\text{def}}{=} & \mathsf{F} \\ [\![\mathsf{node}(v, P, Q)]\!] & \stackrel{\text{def}}{=} & (v \wedge [\![P]\!]) \vee (\neg v \wedge [\![Q]\!]) \end{array}$$

If the variables in V are ordered by $<$, and for every BDD $B = \mathsf{node}(v, P, Q)$ the following two conditions apply then B is said to be an Ordered-BDD (OBDD):

1. If $P \notin \{\mathsf{T}, \mathsf{F}\}$ then $\mathsf{var}(B) < \mathsf{var}(P)$.

2. If $Q \notin \{\mathsf{T}, \mathsf{F}\}$ then $\mathsf{var}(B) < \mathsf{var}(Q)$.

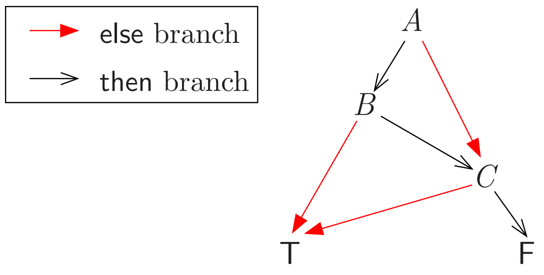


Figure 2.13: A Reduced-OBDD for $A(B\overline{C} \vee \overline{B}) \vee \overline{A}\,\overline{C}$.

If B is an BDD and $v \in V$ then $B{\downarrow_v}$ denotes the BDD for B with the value T substituted for v. If B is a BDD and $v \in V$ then $B{\downarrow_{\overline{v}}}$ denotes the BDD for B with the value F substituted for v. If $B = \mathsf{node}(w, P, Q)$ is an OBDD and $v < w$ then $B{\downarrow_v} = B{\downarrow_{\overline{v}}} = B$. If $B = \mathsf{node}(w, P, Q)$ is an OBDD and $v = w$ then $B{\downarrow_v} = P$, $B{\downarrow_{\overline{v}}} = Q$. Hence if B is an OBDD and $v \leq w$ then $B{\downarrow_v}$ and $B{\downarrow_{\overline{v}}}$ can be computed in constant time. Using this assertion it is possible to implement all boolean operators on OBDDs efficiently using a single generic if-then-else algorithm, $\mathsf{ite}(F, G, H)$, to compute the OBDD for $(F \wedge G) \vee (\neg F \wedge H)$ recursively on the structure of F, G, H as follows:

$$\begin{aligned}
\mathsf{ite}(\mathsf{T}, G, H) &= G \\
\mathsf{ite}(\mathsf{F}, G, H) &= H \\
\mathsf{ite}(\mathsf{node}(v, P, Q), \mathsf{F}, \mathsf{F}) &= \mathsf{F} \\
\mathsf{ite}(\mathsf{node}(v, P, Q), \mathsf{F}, \mathsf{T}) &= \mathsf{node}(v, Q, P) \\
\mathsf{ite}(\mathsf{node}(v, P, Q), \mathsf{T}, \mathsf{F}) &= \mathsf{node}(v, P, Q) \\
\mathsf{ite}(\mathsf{node}(v, P, Q), \mathsf{T}, \mathsf{T}) &= \mathsf{T} \\
\mathsf{ite}(\mathsf{node}(v, P, Q), \mathsf{node}(w, R, S), \mathsf{node}(x, T, U)) &= \\
&\qquad \textsc{let}\ a = \min(v, w, x)\ \textsc{in}\ \mathsf{node}(a, \mathsf{ite}(P{\downarrow_a}, R{\downarrow_a}, T{\downarrow_a}), \mathsf{ite}(Q{\downarrow_{\overline{a}}}, S{\downarrow_{\overline{a}}}, U{\downarrow_{\overline{a}}}))
\end{aligned}$$

A Reduced-OBDD (ROBDD) is an OBDD in which every node is distinct, and therefore any two branches to identical sub-trees share the same pointer, see Figure 2.13. ROBDD function libraries are both memory efficient and fast [83] and have been extensively applied to formal verification for the purpose of reachability analysis on state graphs [50].

2.4 Summary

The purpose of this chapter was to present a brief introduction to asynchronous design and its associated formal methods. Two different classifications of asynchronous design style were presented, and formal methods in asynchronous design were classified according to two simple criteria: sequence and concurrency. An important foundation behind this classification was a desire to overview differences between the semantics of each formal notation in such away that similarities between their underlying symbols and constructs remained clear. In particular, the reader's attention is now drawn to the observation that encoding of events or actions as elements from a finite set of symbols is ubiquitous. This observation was an important inspiration behind the proposition-oriented methods evolved in the following chapters.

Chapter 3
Extending Events to Propositions

3.1 Introduction

If C is a circuit with n wires then the digital value on all of these wires can be represented by a bit-vector $s = \langle b_1, b_2, \ldots, b_n \rangle$ of length n. Any sequence $S = s_0 s_1 s_2 \ldots$ of such bit-vectors can be used to represent an execution of C. Any S of this form is said to be a *relative-time* execution since it does not contain any absolute timing information. Any pair (s_i, s_{i+1}) of consecutive states in S identifies a set of level changes or *events* on a certain subset e_i of the wires in C. For every s_i and e_i it is possible to deduce s_{i+1}, and therefore by induction, provided the initial state s_0 is known, any sequence of event-sets $e_1 e_2 e_3 \ldots$ may also be used to denote an execution of C. A sequence of bit-vectors denotes a *level-oriented* execution whereas a sequence of event-sets denotes an *event-oriented* execution. An *edge-oriented* execution is an event-oriented execution where every e_i is partitioned into two parts: one part containing rising events and the other part containing falling events.

Any relative-time execution may be restricted by asserting that no two events can occur simultaneously. This restriction is referred to as a *Single-Winner* model of behaviour. In the context of a Single-Winner event-oriented execution every e_i contains exactly one element, and in the context of a Single-Winner level-oriented execution every pair (s_i, s_{i+1}) of consecutive states differ in exactly one bit position. A Single-Winner restriction is justifiable for relative-time executions since any two consecutive relative-time events may be arbitrarily close together in absolute-time. Conversely any two apparently simultaneous events can always be ordered with sufficient absolute-time resolution. A relative-time execution which is not Single-Winner is referred to as a *Multiple-Winner* execution. Single-Winner event-oriented executions are often denoted as a sequence of wire names. Single-Winner edge-oriented executions are often denoted as a sequence of wire names in which each wire name w is also identified as either a rising or a falling event: for example as w^+ or w^-.

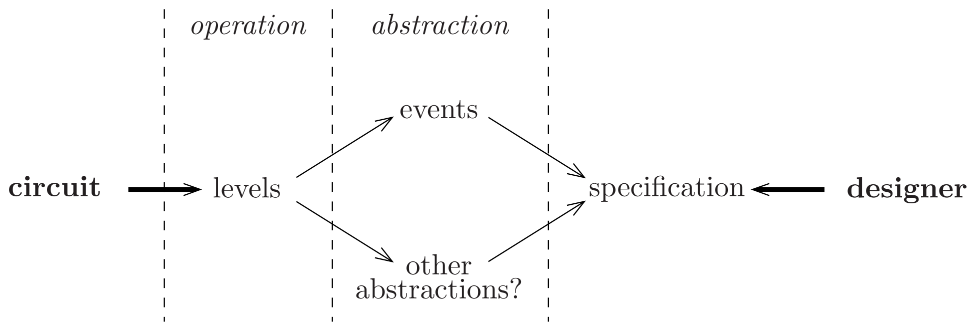


Figure 3.1: Abstractions between circuit and specification.

In the context of circuit verification, the underlying model of execution limits both the circuit behaviours that can be modelled and the type of specifications that can be made: something that cannot be modelled cannot be verified. Furthermore, since a MOS transistor is a voltage controlled device, the *operation* of a digital circuit is dependent on levels not events. Event-based models of behaviour are necessarily *abstractions* that infer events from levels for the purpose of designer convenience, see Figure 3.1. If a circuit is described directly using an event-oriented model of behaviour then the abstraction still exists, it is just that the translation from level-oriented behaviour to event-oriented behaviour has been performed by hand. The wide-spread success of notations such as Petri-nets, trace-expressions, burst-mode machines, and DI-algebra is a strong indication that event-oriented abstractions are effective for asynchronous circuit design, however this thesis aims to demonstrate that an alternative abstraction may be more appropriate.

The alternative abstraction is referred to as *proposition-oriented* behaviour. Its origin lies in the observation that verbal building blocks for describing circuit behaviours include any statement which identifies meaningful *instants* or *intervals* in time. For example, the verbal statements “dual-rail wires a_0, a_1 assert a data-value” and “a rises when b and c are low” both identify meaningful instants in time. Alternatively, “a is low” or “b is low and c is high” are also meaningful verbal statements that identify well-defined intervals in time. A proposition-oriented execution is any sequence $\phi_0\phi_1\phi_2\ldots$ of boolean propositions in which each ϕ_i matches a definite set of level-oriented (s_i, s_{i+1}) pairs, see Figure 3.2. Proposition-oriented executions include both event-oriented and edge-oriented executions as special cases in which only certain types of proposition may be used.

The purpose of this chapter is to document a formal foundation for proposition-oriented behaviours that is suitable for asynchronous circuit verification. Section 3.2 describes some mathematical preliminaries. Section 3.3 defines an underlying level-oriented model

$$\begin{tikzcd}
\text{level-oriented sequence:} & s_0 \arrow[dr] & & s_1 \arrow[dl] \arrow[dr] & & s_2 \arrow[dl] \arrow[dr] & & s_3 \arrow[dl] \arrow[dr] & & s_4 \arrow[dl] \\
\text{transitions:} & & (s_0, s_1) \arrow[d, leftrightarrow] & & (s_1, s_2) \arrow[d, leftrightarrow] & & (s_2, s_3) \arrow[d, leftrightarrow] & & (s_3, s_4) \arrow[d, leftrightarrow] & \\
\text{propositions:} & & \phi_0 & & \phi_1 & & \phi_2 & & \phi_3 &
\end{tikzcd}$$

Figure 3.2: Proposition-oriented behaviour.

of behaviour on which two proposition-oriented abstractions are then built in Sections 3.5 and 3.6: Section 3.5 describes a proposition-oriented form of regular-expression, and Section 3.6 describes a proposition-oriented form of trace-expression. The application of both proposition-oriented regular-expressions and proposition-oriented trace-expressions to asynchronous circuit design are discussed in detail in Chapter 6. Section 3.4 shows how some basic circuit building blocks such as AND-gates, C-elements, and arbiters can be encoded using the level-oriented model of behaviour defined in Section 3.3.

3.2 Mathematical Preliminary

3.2.1 Set Notation

If X is a set then $x \in X$ means that x is an element of X, and $x \notin X$ means that x is not an element of X. $|X|$ denotes the number of elements in X. If X and Y are sets then $X \subseteq Y$ means that X is a subset of Y. If $X \subseteq Y$ and $X \neq Y$ then $X \subset Y$ meaning that X is a proper subset of Y. If $|X| = 0$ then $X = \emptyset$ is the empty set. $2^X = \{Y \mid Y \subseteq X\}$ denotes the set of all subsets of X, also called the power set of X. $\forall x \in X$ denotes universal quantification over all elements x in X, and $\exists x \in X$ denotes existential quantification over some element x in X.

If X and Y are sets then $X \times Y = \{(x, y) \mid x \in X \text{ and } y \in Y\}$ denotes the product set of X and Y. $X \to Y$ denotes the set of all total functions from X to Y and $X \rightharpoondown Y$ denotes the set of all partial functions from X to Y. If $f \in X \to Y$ then $f(x) = y$ is shorthand for $(x, y) \in f$. $f \in X \to Y$ and $f \in X \rightharpoondown Y$ are different from $f \subseteq X \times Y$ in that if $x \in X$ then any $f \in X \to Y$ must have $|\{y \mid f(x) = y\}| = 1$ and any $f \in X \rightharpoondown Y$ must have $|\{y \mid f(x) = y\}| \leq 1$. If X and Y are sets then $X \cup Y$ denotes set union, $X \cap Y$ denotes set intersection, and $X \uplus Y$ denotes disjoint union. $X - Y = \{x \mid x \in X$ and $x \notin Y\}$ denotes set subtraction. If $X = \{x_1, x_2, \ldots, x_n\}$ then $\bigcap_{x \in X} f(x)$ is shorthand for

$f(x_1) \cap f(x_2) \cap \cdots \cap f(x_n)$ and $\bigcup_{x \in X} f(x)$ is shorthand for $f(x_1) \cup f(x_2) \cup \cdots \cup f(x_n)$. Define $\bigcup_{x \in \emptyset} f(x) \overset{\text{def}}{=} \emptyset$ and define $\bigcap_{x \in \emptyset} f(x) \overset{\text{def}}{=} \Omega$ where Ω is the universe of discourse appropriate for $f(x)$.

3.2.2 Boolean Expressions

If V is a finite set of variable names then the set *Bexp* of all boolean expressions over V is defined recursively as follows:

$$\begin{array}{rcl} \mathit{Bexp} & ::= & \mathsf{T} \mid \mathsf{F} \mid v \mid \neg \mathit{Bexp} \mid \mathit{Bexp} \wedge \mathit{Bexp} \mid \mathit{Bexp} \vee \mathit{Bexp} \mid \mathit{Bexp} \oplus \mathit{Bexp} \mid \\ & & \mathit{Bexp} \Rightarrow \mathit{Bexp} \mid \mathit{Bexp} \Leftrightarrow \mathit{Bexp} \mid v = v \mid v \neq v \\ v & ::= & \text{Any element of } V \end{array}$$

The meaning $[\![B]\!]_V \subseteq 2^V$ of a boolean expression B over is defined inductively on the structure of B as follows:

- $[\![\mathsf{T}]\!]_V \overset{\text{def}}{=} 2^V$. Denotes the boolean constant TRUE.
- $[\![\mathsf{F}]\!]_V \overset{\text{def}}{=} \emptyset$. Denotes the boolean constant FALSE.
- $[\![v]\!]_V \overset{\text{def}}{=} \{X \subseteq V \mid v \in X\}$. Denotes the boolean variable v.
- $[\![\neg B]\!]_V \overset{\text{def}}{=} 2^V - [\![B]\!]_V$. Denotes boolean negation.
- $[\![B_1 \vee B_2]\!]_V \overset{\text{def}}{=} [\![B_1]\!]_V \cup [\![B_2]\!]_V$. Denotes logical OR.
- $[\![B_1 \wedge B_2]\!]_V \overset{\text{def}}{=} [\![B_1]\!]_V \cap [\![B_2]\!]_V$. Denotes logical AND.
- $[\![B_1 \oplus B_2]\!]_V \overset{\text{def}}{=} [\![(B_1 \wedge \neg B_2) \vee (\neg B_1 \wedge B_2)]\!]_V$. Denotes logical EXCLUSIVE-OR.
- $[\![B_1 \Rightarrow B_2]\!]_V \overset{\text{def}}{=} [\![\neg B_1 \vee B_2]\!]_V$. Denotes boolean implication.
- $[\![B_1 \Leftrightarrow B_2]\!]_V \overset{\text{def}}{=} [\![(B_1 \Rightarrow B_2) \wedge (B_2 \Rightarrow B_1)]\!]_V$. Denotes boolean double-implication.
- $[\![B_1 \neq B_2]\!]_V \overset{\text{def}}{=} [\![B_1 \oplus B_2]\!]_V$. Denotes expression inequality.
- $[\![B_1 = B_2]\!]_V \overset{\text{def}}{=} [\![\neg(B_1 \neq B_2)]\!]_V$. Denotes expression equality.

Every $s \in [\![B]\!]_V$ denotes the *characteristic set* for a particular binary value assignment $A \in V \to \{0, 1\}$ to the variables in $v \in V$: if $v \in s$ then $A(v) = 1$ else $A(v) = 0$. Define $\mathsf{bool}(V) \overset{\text{def}}{=} 2^V$ so that $s \in 2^V$ can be re-written as $s \in \mathsf{bool}(V)$ whenever s is to be interpreted as the characteristic set for a boolean variable value assignment. Define

$\mathsf{bexp}(V)$ to be the set of all boolean expressions over the variables in V. If B_1 and B_2 are boolean expressions over the variables in V then define $B_1 \equiv B_2 \overset{\text{def}}{=} [\![B_1]\!]_V = [\![B_2]\!]_V$.

If B is a boolean expression over V and $V \subseteq W$ then define $[\![B]\!]_W \overset{\text{def}}{=} \{b \subseteq W \mid b \cap V \in [\![B]\!]_V\}$. If B_1 is a boolean expression over V_1 and B_2 is a boolean expression over V_2 then any boolean expression $B_1 \mathit{op}\ B_2$ constructed from B_1 and B_2 is defined over the variables in $V_1 \cup V_2$.

If B is a boolean expression over V_1, q is a boolean expression over V_2, and $x \in V_1$, then define $B[q/x]$ to denote the boolean expression over $V_1 \cup V_2$ obtained by replacing all instances of x in B by q. If $Q \in V_1 \rightharpoonup \mathsf{bexp}(V_2)$ and $V_1 \cap V_2 = \emptyset$ then define $B[Q]$ to denote the iterative application of $B[q/x]$ for each $(x, q) \in Q$. Note that since $V_1 \cap V_2 = \emptyset$ the order of these applications does not matter. If B does not depend on x then $B[q/x] \equiv B$ and if $B = x$ then $B[q/x] \equiv q$.

If B is a boolean expression over V and $x \in V$ then define $\exists x.B \overset{\text{def}}{=} B[\mathsf{T}/x] \vee B[\mathsf{F}/x]$ and define $\forall x.B \overset{\text{def}}{=} B[\mathsf{T}/x] \wedge B[\mathsf{F}/x]$. If $X = \{x_1, \ldots, x_n\} \subseteq V$ then $\exists X.B$ is shorthand for $\exists x_1. \cdots .\exists x_n.B$ and $\forall X.B$ is shorthand for $\forall x_1. \cdots .\forall x_n.B$. If $B = \{b_1, b_2, \ldots, b_n\} \subseteq \mathsf{bexp}(V)$ then $\bigwedge_{b \in B} b$ is shorthand for $b_1 \wedge b_2 \wedge \cdots \wedge b_n$ and $\bigvee_{b \in B} b$ is shorthand for $b_1 \vee b_2 \vee \cdots \vee b_n$. Define $\bigvee_{b \in \emptyset} b \overset{\text{def}}{=} \mathsf{F}$ and define $\bigwedge_{b \in \emptyset} b \overset{\text{def}}{=} \mathsf{T}$.

3.2.3 Finite Sequences

If Σ is a finite set of symbols and $n \in \mathbb{N}$, then any function $\sigma \in \{0 \ldots (n-1)\} \to \Sigma$ is a *finite sequence* of length n over Σ. Define $|\sigma| = n$. The sequence of length zero is the empty sequence and is denoted by the symbol ε. If $|\sigma| > 0$ then define $\mathsf{last}(\sigma) \overset{\text{def}}{=} \sigma(|\sigma| - 1)$. The concatenation of two sequences σ_1 and σ_2 over Σ is a function $\sigma_1\sigma_2 \in \{0 \ldots (|\sigma_1| + |\sigma_2| - 1)\} \to \Sigma$ such that $\sigma_1\sigma_2(i) = \sigma_1(i)$ if $i < |\sigma_1|$ and $\sigma_1\sigma_2(i) = \sigma_2(i - |\sigma_1|)$ if $i \geq |\sigma_1|$.

For every $s \in \Sigma$, $\langle s \rangle_\Sigma$ denotes the singleton sequence of length 1 with $\langle s \rangle_\Sigma(0) = s$. If σ is a finite sequence over Σ and $A \subseteq \Sigma$ then $\sigma \restriction A$ denotes the finite sequence formed by removing all symbols not in A from σ. If σ is a finite sequence over Σ then the *pairwise extension* of σ is the finite sequence σ^2 over $\Sigma \times \Sigma$ defined as follows: if $|\sigma| < 2$ then $\sigma^2 \overset{\text{def}}{=} \varepsilon$ else $|\sigma^2| = |\sigma| - 1$ and $\sigma^2(n) \overset{\text{def}}{=} (\sigma(n), \sigma(n+1))$.

A *set of finite sequences* over Σ is any set X such that every $x \in X$ is a finite sequence over Σ. If X is a set of finite sequences over Σ and $n \in \mathbb{N}$ then X^n is defined recursively as follows: $X^0 \overset{\text{def}}{=} \{\varepsilon\}$ and $X^{n+1} \overset{\text{def}}{=} \{\sigma_1\sigma_2 \mid \sigma_1 \in X \text{ and } \sigma_2 \in X^n\}$. Define $X^* \overset{\text{def}}{=} \bigcup_{n \in \mathbb{N}} X^n$.

If X is a set of finite sequences over Σ then $\textbf{pref}\, X \overset{\text{def}}{=} \{\sigma_1 \mid \exists\sigma_2.\ \sigma_1\sigma_2 \in X\}$ denotes the prefix-closure of X, and $\textbf{stpref}\, X \overset{\text{def}}{=} (\textbf{pref}\, X) - X$ denotes strict prefix-closure of X.

3.3 Component Network Model of Behaviour

In Section 3.1 a relative-time level-oriented execution of circuit C was defined as a sequence $s_0 s_1 s_2 \ldots$ of bit-vectors. Each $s_i = \langle b_0 b_1 b_2 \ldots\rangle$ determined a particular binary value assignment to the level of every wire in C. Gate network models [15] are a representation of the set of all relative-time level-oriented executions for C as sets of state graphs, see Section 2.3.4. The purpose of this section is to introduce a new gate network model, referred to as a *component network model*, from which proposition-oriented behaviours can then be formalised.

3.3.1 Primitive Component

A primitive component C is a state graph $\langle ins, outs, S, T, I\rangle$ where:

- ins is a finite set of input wire names.
- $outs$ is a finite set of output wire names.
- $S \subseteq \mathsf{bool}(outs)$ is a finite set of component states such that each state $s \in S$ identifies a unique binary value assignment to each output wire $w \in outs$. If $s \in S$ and $w \in outs$ then define $\lambda_w(s) \overset{\text{def}}{=} 1$ if $w \in s$, and define $\lambda_w(s) \overset{\text{def}}{=} 0$ if $w \notin s$.
- $T \subseteq S \times \mathsf{bool}(ins) \times S$ is a labelled transition relation over value assignments to the wires in ins. If $s_1 \in S$, $s_2 \in S$, and e is a boolean expression over the variables in ins then define $s_1 \xrightarrow{e} s_2 \overset{\text{def}}{=} \{(s_1, l, s_2) \mid l \in [\![e]\!]_{ins}\}$.
- $I \subseteq \mathsf{bool}(ins) \times S$ is a set of possible initial configurations for C. I is not defined as $I \subseteq S$ since in general the initial state of C might depend on the initial values of its inputs. Defining $I \subseteq \mathsf{bool}(ins) \times S$ allows initial states of C to be bound to certain initial input values and thus enables this dependency to be expressed.

Since T is a relation and not a function, the behaviour of C is non-deterministic. Non-determinism is necessary if components involving arbitration are to be successfully modelled by C. C is also a wire-state only model [16] in that its state can always be determined from the binary values on its outputs. A wire-state only model is chosen since it prevents

C from hiding internal state at its outputs, a property which is discussed further in Section 4.1.

If C is a primitive component then $C.S$ denotes the states of C, $C.outs$ denotes the output wire names for C, $C.\lambda_x$ denotes the output valuation function for output $x \in C.outs$, and so forth.

3.3.2 Component Network

For any finite set $N = \{c_1, c_2, \ldots, c_n\}$ of primitive components define:

- $\mathsf{driven}(N) \overset{\text{def}}{=} \bigcup_{c \in N}(c.outs)$.
- $\mathsf{wires}(N) \overset{\text{def}}{=} \bigcup_{c \in N}(c.outs \cup c.ins)$.

If $\mathsf{driven}(N) = \mathsf{wires}(N)$ and $\forall c \in N.\ \mathsf{driven}(N - c)\ \cap\ c.outs = \emptyset$ then N is called a *component network*. If $\mathsf{driven}(N) \subset \mathsf{wires}(N)$ then there is at least one wire without any output valuation function. If $\mathsf{driven}(N - c)\ \cap\ c.outs \neq \emptyset$ for some $c \in N$ then there is at least one wire with more than one output valuation function.

A component network N can be constructed from any digital circuit provided every primitive gate in that circuit has an analogous primitive component definition. The translation from circuit to component network requires a unique $c \in N$ for every primitive gate instance, and a unique identifier $w \in \mathsf{wires}(N)$ for every wire. If $w \in c_i.outs$ then wire w is driven by c_i and has value $c_i.\lambda_w(s)$ when c_i is in state s. If w is also in $c_j.ins$ then component c_j has wire w as one of its inputs. Requiring $\mathsf{driven}(N) = \mathsf{wires}(N)$ ensures that no wires are left floating and requiring that $\forall c \in N.\ \mathsf{driven}(N - c) \cap c.outs = \emptyset$ ensures that no wire has multiple drivers.

Component Network Model

If $N = \{c_1, c_2, \ldots, c_n\}$ is a component network then the *component network model* for N is an automaton $\langle \mathsf{states}(N), \mathsf{trans}(N), \mathsf{init}(N) \rangle$ where:

- $\mathsf{states}(N) \overset{\text{def}}{=} c_1.S \times c_2.S \times \cdots \times c_n.S$ is the product state-set formed from the individual primitive component state-sets in N.
- $\mathsf{proj}(s \in S, c \in N) \in c.S$ denotes the element of s representing a state in $c.S$.
- $\mathsf{parent}(x \in \mathsf{wires}(N)) \in N$ denotes the unique primitive component in N for which $x \in c.outs$.

- $\mathsf{val}(x \in \mathsf{wires}(N), s \in S) \overset{\text{def}}{=} \mathsf{parent}(x).\lambda_x(\mathsf{proj}(s, \mathsf{parent}(x)))$ is the value of wire x in network model state s.
- $\mathsf{inset}(c \in N, s \in S) \overset{\text{def}}{=} \{x \in c.ins \mid \mathsf{val}(x, s) = 1\}$ denotes the set of inputs to primitive component c that are at level 1 in network model state s.
- $\mathsf{trans}(N) \overset{\text{def}}{=} \{(s_1, s_2) \in S \times S \mid \forall c \in N.\ (\mathsf{proj}(s_1, c), \mathsf{inset}(c, s_1), \mathsf{proj}(s_2, c)) \in c.T\}$ is a transition relation including all pairs of states (s, s') such that each of the c_i makes a valid transition.
- $\mathsf{alltrans}(N) \overset{\text{def}}{=} \mathsf{states}(N) \times \mathsf{states}(N)$ is the maximal possible set for $\mathsf{trans}(N)$.
- $\mathsf{init}(N) \overset{\text{def}}{=} \{s \in S \mid \forall c \in N.\ (\mathsf{inset}(c, s), \mathsf{proj}(s, c)) \in c.I\}$ is a set of possible initial states.

The component network model for N is a form of product state graph which models the concurrent execution of each $C \in N$ but requires that any coupling between inputs and outputs is adhered to. The behaviour of a component network model is identical to that of a Multiple-Winner execution model since more than one primitive component may change state per component network transition and therefore more than one wire may change level per component network transition.

3.3.3 Network Executions

If N is a component network then a *network execution* of N is any finite sequence σ over $\mathsf{states}(N)$ where if $|\sigma| > 0$ then $\sigma(0) \in \mathsf{init}(N)$ and if $|\sigma| > 1$ then $\forall n \in \{0 \ldots (|\sigma| - 2)\}.\ (\sigma(n), \sigma(n+1)) \in \mathsf{trans}(N)$. A network execution of N represents any sequence of states beginning with an initial state of N and continuing in such a way that every consecutive pair of states is a valid transition of N. Define the *language* of N to be the set $L(N) \overset{\text{def}}{=} \{\sigma \text{ over } \mathsf{states}(N) \mid \sigma \text{ is a network execution of } N\}$.

Level-Oriented Network Executions

If N is a component network then define the *level-oriented execution* of N to be the finite sequence σ_l over $\mathsf{bool}(\mathsf{wires}(N))$ where $\sigma_l(n) \overset{\text{def}}{=} \bigcup_{c \in N} \mathsf{proj}(\sigma(n), c)$. Each $\sigma_l(n)$ denotes the set of all component network wires whose level is 1 in component network state $\sigma(n)$, and since component networks are a wire-state only model, σ_l and σ are in one-to-one correspondence. Define the *level-language* of N to be the set
$L_l(N) \overset{\text{def}}{=} \{\sigma_l \text{ over } \mathsf{bool}(\mathsf{wires}(N)) \mid \sigma \in L(N)\}$.

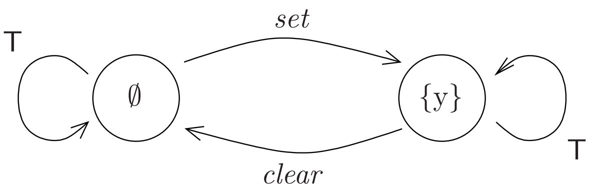


Figure 3.3: Transition relation for C_{single}.

3.4 Primitive Components for Asynchronous Design

This section documents some basic primitive components sufficient to model most asynchronous circuit designs. Each of these components adopts an *inertial* model of delay [16]: any output y of circuit element E that is enabled to change its value may wait an arbitrary number of component network transitions before doing so. If, while waiting, the inputs to E continue to change so as to disable y, then y may no longer change its value. In the context of a primitive component C, an output y is *enabled* in configuration $(s_1, l) \in C.S \times \mathsf{bool}(C.ins)$ if there exists another state $s_2 \neq s_1$ with $(s_1, l, s_2) \in C.T$ and $C.\lambda_y(s_1) \neq C.\lambda_y(s_2)$.

3.4.1 Generic Single Output Gate

Conventional boolean logic gates have single outputs. Many basic gates in asynchronous design also have single outputs. Buffers, inverters, complex gates, C-elements, generalised C-elements, and threshold gates [94] are all examples of single output devices. The transition relation for a generic primitive component $C_{single} = \langle ins, \{y\}, \mathsf{bool}(\{y\}), T, I \rangle$ with a single output y is shown in Figure 3.3. C_{single} has two states $\emptyset$ and $\{y\}$. If C_{single} is in state $\emptyset$ then $\lambda_y(\emptyset) = 0$ and if C_{single} is in state $\{y\}$ then $\lambda_y(\{y\}) = 1$. The behaviour of y is inertial and therefore s0 $\xrightarrow{\mathsf{T}}$ s0 and s1 $\xrightarrow{\mathsf{T}}$ s1 since y may always retain its previous value. The condition under which y can fall from level 1 to 0 is identified by the predicate *clear*, and the condition under which y can rise from level 0 to 1 is identified by the predicate *set*.

If E is the circuit element being modelled by C_{single} then *set* represents a pull-up transistor stack for y in E and *clear* represents a pull-down transistor stack for y in E. If $set \wedge clear \not\equiv \mathsf{F}$ then it is possible to enable both the pull-up and pull-down stacks simultaneously and consequently create a short from VDD to ground. All practical instances of C_{single} must therefore have $set \wedge clear \equiv \mathsf{F}$. For a conventional boolean gate it is also true that $set \vee clear \equiv \mathsf{T}$, however this does not apply to certain asynchronous components with memory, such as a C-element.

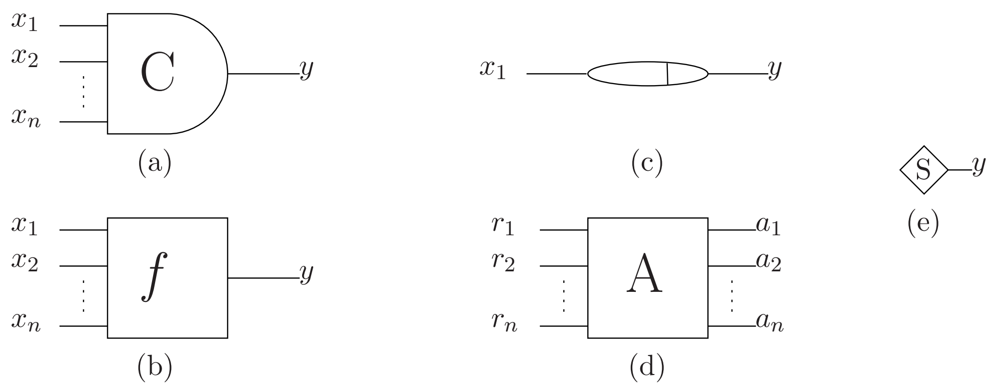


Figure 3.4: Primitive Component Icons.

C_{single} can start up in any state except for initial configurations where either *set* or *clear* are true in which case the initial state of C_{single} is determined by the transistor stack which is active during circuit initialisation. A formal definition of I is as follows:

$$I \overset{\text{def}}{=} (\mathsf{bool}(ins) \times \mathsf{bool}(\{y\})) - \{(l, \emptyset) \mid l \in [\![set]\!]_{ins}\} - \{(l, \{y\}) \mid l \in [\![clear]\!]_{ins}\}$$

3.4.2 C-element

Let C_{celem} be the primitive component model for an n-input symmetric C-element. Let $C_{celem}.ins = \{x_1, x_2, \ldots, x_n\}$ as shown in Figure 3.4(a). C_{celem} can now be constructed from C_{single} by defining $set = x_1 \wedge x_2 \wedge \cdots \wedge x_n$ and $clear = \neg x_1 \wedge \neg x_2 \wedge \cdots \wedge \neg x_n$. An input x_i can be made asymmetric [39] either by removing x_i from *set* or by removing $\neg x_i$ from *clear*. If x_i is removed from *set* then x_i is a *negative*$(-)$ asymmetric input and if $\neg x_i$ is removed from *clear* then x_i is a *positive*$(+)$ asymmetric input.

3.4.3 Boolean Functions as Complex Gates

Let C_f be the primitive component model for a single complex gate implementing the boolean function f of arity n. Let $C_f.ins = \{x_1, x_2, \ldots, x_n\}$ as shown in Figure 3.4(b). C_f can now be constructed from C_{single} by defining $set = f(x_1, x_2, \ldots, x_n)$ and $clear = \neg f(x_1, x_2, \ldots, x_n)$.

The primitive component model for an inertial delay element C_{delay}, see Figure 3.4(c), is identical to that of a complex gate C_{id} where id is the identity function $id(x) = x$. The implementation of an inertial delay element as a complex gate may also be referred to as a buffer.

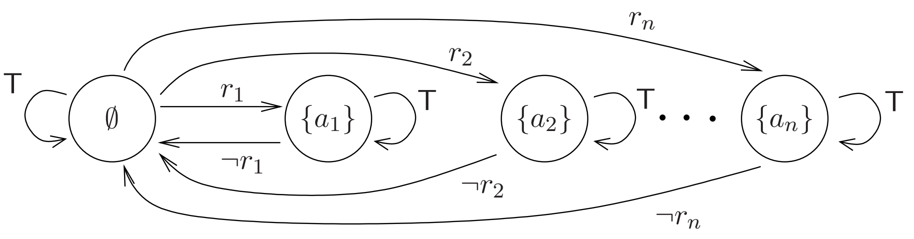


Figure 3.5: Transition relation for $C_{arbiter(n)}$.

3.4.4 Arbiter

Let $C_{arbiter(n)} = \langle\{r_1, r_2, \ldots, r_n\}, \{a_1, a_2, \ldots, a_n\}, \{\emptyset, \{a_1\}, \{a_2\}, \ldots, \{a_n\}\}, T, I\rangle$ be the primitive component model for an n-input arbitration element with n-outputs. The transition relation for $C_{arbiter(n)}$ is shown in Figure 3.5, and its associated circuit icon is shown in Figure 3.4(d). $C_{arbiter(n)}$ has $n + 1$ states. If $C_{arbiter(n)}$ is in state $\emptyset$ then no output is granted and every a_i is at level 0. If $C_{arbiter(n)}$ is in state $\{a_i\}$ then the i^{th} output is granted and a_i only is at level 1. To grant output i input r_i must be at level 1, and to revoke a grant on output i input r_i must be at level 0. If $i \neq j$ then $C_{arbiter(n)}$ cannot make a transition from state $\{a_i\}$ to state $\{a_j\}$ without entering state $\emptyset$ in between. $C_{arbiter(n)}$ always starts up in state $\emptyset$ even if some of the r_i are at level 1 during circuit initialisation. A formal definition for I is $I \overset{\text{def}}{=} \{(l, \emptyset) \mid l \in \mathsf{bool}(ins)\}$.

3.4.5 Source

A source component $C_{source} \overset{\text{def}}{=} \langle\emptyset, \{y\}, \mathsf{bool}(\{y\}), T, I\rangle$ is a special form of primitive component with no inputs and a single output y that can always change its value:
$I = \mathsf{bool}(\emptyset) \times \mathsf{bool}(\{y\})$ and $T = \mathsf{bool}(\{y\}) \times \mathsf{bool}(\emptyset) \times \mathsf{bool}(\{y\})$.

C_{source} does not model any physical device, and is intended merely as the model for a wire y whose behaviour is unknown or unconstrained. A circuit icon for C_{source} is shown in Figure 3.4(e).

3.5 Proposition-Oriented Regular-Expressions

The purpose of this section is to evolve a basic sequential notation over proposition-oriented behaviours in the context of a component network model. In order to do this a special type of proposition called a *network proposition* is first defined. A simple regular-expression-like specification language is then formalised from which proposition-oriented behaviours can be related to component network executions.

3.5.1 Network Proposition

For any component network N define:

- $\mathsf{wires}'(N) \stackrel{\text{def}}{=} \{w' \mid w \in \mathsf{wires}(N)\}$ to denote a duplicate set of wire names w' for each wire $w \in \mathsf{wires}(N)$.
- $\mathsf{dprime}(x \subseteq \mathsf{wires}(N)) \stackrel{\text{def}}{=} x \cup \{w' \mid w \in x\}$ is the set of wire names from x in both primed and unprimed form. For example, if $x = \{w_1, w_2\}$ then $\mathsf{dprime}(x) = \{w_1, w_2, w_1', w_2'\}$.
- $\mathsf{dwires}(N) \stackrel{\text{def}}{=} \mathsf{dprime}(\mathsf{wires}(N))$ is the set of all component network wire names in both primed and unprimed form.
- $\mathsf{curr}(x \subseteq \mathsf{dwires}(N)) \stackrel{\text{def}}{=} x \cap \mathsf{wires}(N)$ is the set of all unprimed variables in x.
- $\mathsf{next}'(x \subseteq \mathsf{dwires}(N)) \stackrel{\text{def}}{=} x \cap \mathsf{wires}'(N)$ is the set of all primed variables in x.
- $\mathsf{next}(x) \stackrel{\text{def}}{=} \{w \mid w' \in \mathsf{next}'(x)\}$ is the set of all primed variables in x renamed back to unprimed form.
- $\mathsf{oneset}(s \in \mathsf{states}(N)) \stackrel{\text{def}}{=} \{w \in \mathsf{wires}(N) \mid \mathsf{val}(w, s) = 1\}$ is the set of all wires whose value is 1 in component network state s.
- $\mathsf{smap}(x \in \mathsf{wires}(N)) \stackrel{\text{def}}{=} \{s \in \mathsf{states}(N) \mid \mathsf{oneset}(s) = x\}$ is the set of all component network states s with $\mathsf{oneset}(s) = x$.
- $\mathsf{tmap}(x \subseteq \mathsf{dwires}(N)) \stackrel{\text{def}}{=} \{(s_1, s_2) \mid s_1 \in \mathsf{smap}(\mathsf{curr}(x))$ and $s_2 \in \mathsf{smap}(\mathsf{next}(x))\}$ is the set of component network transitions from states matching $\mathsf{curr}(x)$ to states matching $\mathsf{next}(x)$.

Every wire name $w \in \mathsf{wires}(N)$ occurs twice in $\mathsf{dwires}(N)$, once in primed form and once in unprimed form. Any subset x of $\mathsf{dwires}(N)$ can therefore be used to denote two sets of value assignments to the wires in $\mathsf{wires}(N)$. The unprimed variables in $\mathsf{curr}(x)$ denote one of these assignments and the primed variables in $\mathsf{next}'(x)$ denote the other assignment. Every $\mathsf{curr}(x)$ identifies a specific set $\mathsf{smap}(\mathsf{curr}(x))$ of component network states which result in the same value assignment to the wires in $\mathsf{wires}(N)$ as $\mathsf{curr}(x)$. A similar set $\mathsf{smap}(\mathsf{next}(x))$ is identified by $\mathsf{next}(x)$. It is entirely possible for $\mathsf{smap}(\mathsf{curr}(x))$ or $\mathsf{smap}(\mathsf{next}(x))$ to be empty. $\mathsf{tmap}(x)$ denotes the complete set of network transitions from states in $\mathsf{smap}(\mathsf{curr}(x))$ to states in $\mathsf{smap}(\mathsf{next}(x))$. $\mathsf{tmap}(x)$ may be empty, as may $\mathsf{tmap}(x) \cap \mathsf{trans}(N)$.

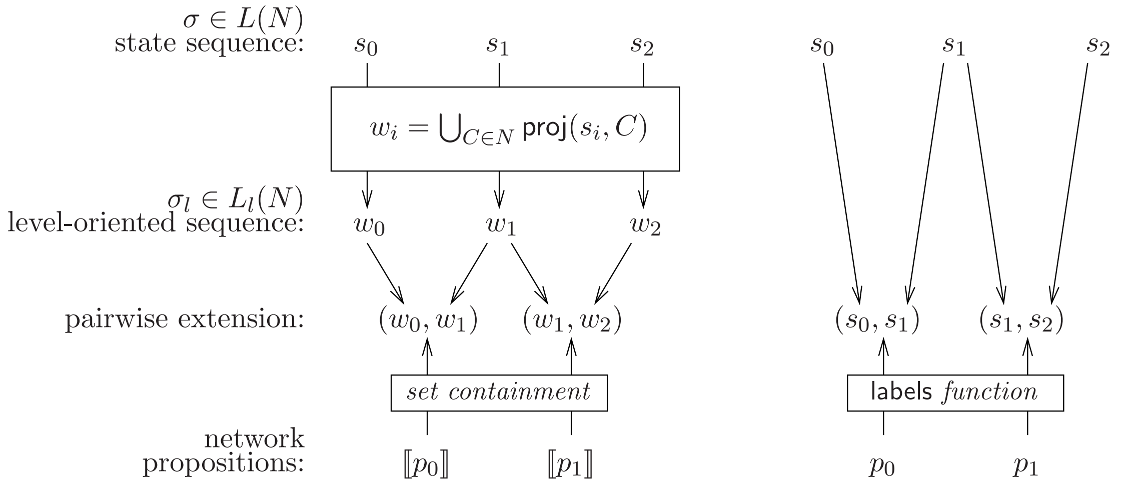


Figure 3.6: Relationship between network propositions and network executions.

A *network proposition* for N is any boolean expression over the variables in $\mathsf{dwires}(N)$. The set of all possible network propositions for N is denoted by $\mathsf{allprops}(N)$. Network propositions are closed under the boolean expression operators defined in Section 3.2.2: if p_1 and p_2 are network propositions then so are $\neg p_1$, $p_1 \wedge p_2$, $p_1 \vee p_2$, and so forth. A network proposition p is *safe* if $p \not\equiv \mathsf{F}$. Every network proposition p identifies a unique set of network transitions $\mathsf{labels}(p) \subseteq \mathsf{alltrans}(N)$ where:

$$\mathsf{labels}(p) \overset{\text{def}}{=} \bigcup_{x \in [\![p]\!]_{\mathsf{dwires}(N)}} (\mathsf{tmap}(x))$$

The relationship between network propositions and network executions is summarised in Figure 3.6. It is important to note that network propositions reason with component network *transitions* not states, and therefore the pairwise extension σ^2 of a network execution σ must always be constructed before σ can be related to a sequence of network propositions. If $a \in \mathsf{wires}(N)$ and $b \in \mathsf{wires}(N)$ then define $a^+ \overset{\text{def}}{=} a = 0 \;\wedge\; a' = 1$, $a^- \overset{\text{def}}{=} a{=}1 \;\wedge\; a'{=}0$, and $a^* \overset{\text{def}}{=} a \neq a'$.

3.5.2 Regular-Expressions over Network Propositions

If N is a component network then the set *Rexp* of all regular-expressions over network propositions for N is defined recursively as follows:

$$\begin{aligned} \mathit{Rexp} \;&::=\; p \;\big|\; \mathit{Rexp}\,;\mathit{Rexp} \;\big|\; \mathit{Rexp} \mid \mathit{Rexp} \;\big|\; [\mathit{Rexp}] \;\big|\; \mathit{Rexp}\,\&\,\mathit{Rexp} \\ p \;&::=\; \text{any safe network proposition.} \end{aligned}$$

For every $E \in \mathit{Rexp}$ the language $L_{re}(E)$ of E is defined recursively on the structure of E as follows:

- $L_{re}(p) \overset{\text{def}}{=} \{\langle l\rangle_{\mathsf{alltrans}(N)} \mid l \in \mathsf{labels}(p)\}$ is the set of singleton sequences for each network transition labelled by p.
- $L_{re}(E_1\,; E_2) \overset{\text{def}}{=} \{\sigma_1\sigma_2 \mid \sigma_1 \in L_{re}(E_1) \text{ and } \sigma_2 \in L_{re}(E_2)\}$ denotes expression concatenation.
- $L_{re}(E_1 \mid E_2) \overset{\text{def}}{=} L_{re}(E_1) \cup L_{re}(E_2)$ denotes expression alternation. σ is in $L_{re}(E_1 \mid E_2)$ if σ is in $L_{re}(E_1)$ or σ is in $L_{re}(E_2)$.
- $L_{re}([E\,]) \overset{\text{def}}{=} (L_{re}(E))^*$ denotes arbitrary repetition of the sequences in $L_{re}(E)$.
- $L_{re}(E_1 \;\&\; E_2) \overset{\text{def}}{=} L_{re}(E_1) \cap L_{re}(E_2)$ denotes expression intersection. σ is in $L_{re}(E_1 \;\&\; E_2)$ if σ is in $L_{re}(E_1)$ and σ is in $L_{re}(E_2)$.

This definition of $L_{re}(E)$ is identical to a conventional definition of regular-expression languages [48] in all cases except for $L_{re}(p)$: a conventional definition of $L_{re}(p)$ would have been as $L_{re}(p) \overset{\text{def}}{=} \langle p\rangle_{\mathsf{allprops}(N)}$ denoting the singleton sequence for p over $\mathsf{allprops}(N)$. However, in the context of network-propositions, every $\sigma \in L_{re}(E)$ is intended to denote the pairwise extension of some network execution and not a sequence of network-propositions. $L_{re}(p)$ is therefore defined instead to match the set of singleton sequences $\langle l\rangle_{\mathsf{alltrans}(N)}$ for each $l \in \mathsf{labels}(p)$, and the *alphabet* of every $E \in \mathit{Rexp}$ is $\mathsf{alltrans}(N)$ not $\mathsf{allprops}(N)$ as might otherwise be expected.

If N is a component network then a network execution $\sigma \in L(N)$ is said to *match* $E \in \mathit{Rexp}$ if $\sigma^2 \in \mathbf{pref}\, L_{re}(E)$. Example applications of proposition-oriented regular-expressions to the verification of asynchronous circuits are given in Chapter 6.

3.6 Proposition-Oriented Trace-Expressions

A proposition-oriented regular-expression can be built using any symbol from an expressive alphabet: boolean expressions over (current,next)-state wire value pairs. For example $[a = a' \wedge b = b']$ can be used to match any network execution where wires a and b never change value or $[\neg a^*]; a^*; [\neg b^*]$ can be used to match any network execution where wire b never changes value after a has changed value. A proposition-oriented regular-expression does however suffer from a limited ability to express the concurrency inherent to most

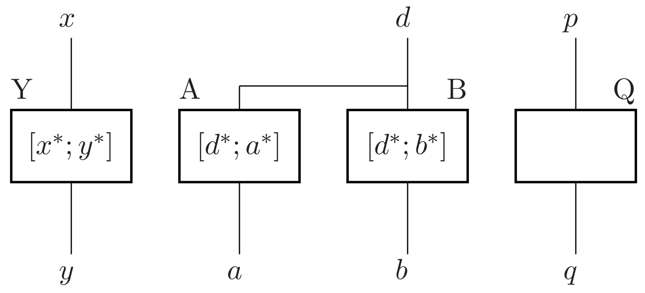


Figure 3.7: Hardware concurrency and regular-expressions.

asynchronous hardware design. Trace-expressions extend regular-expressions in such a way as to counteract this limitation as follows:

Firstly, every trace-expression T is assigned a sort Σ_T denoting a particular set of wire names. If wire $x \in \Sigma_T$ then T is identified as "observing" events on wire x. In a conventional definition of trace-expressions the underlying model of behaviour is Single-Winner event-oriented and the symbols from which T is built are wire names not network propositions: symbol a denotes an event on wire a. In this case the sort of T is just the set of all symbols occurring in T. For example $\Sigma_{[x;y]} = \{x, y\}$. If σ is a Single-Winner event-oriented execution then the trace-expression language of T differs from the regular-expression language of T in that all symbols $\sigma(n) \notin \Sigma_T$ are *thrown away* from σ before it is matched against T.

Secondly, trace-expressions extend the grammar for regular-expressions to include a special *parallel composition* operator "$\|$" that can be used to combine multiple specifications for connected components: suppose T_1 and T_2 are trace-expressions for two connected modules. To match Single-Winner event-oriented execution σ against the parallel composition $T_1 \| T_2$, all symbols $\sigma(n) \notin \Sigma_{T_1}$ are thrown away from σ before it is matched against T_1, and all symbols $\sigma(n) \notin \Sigma_{T_2}$ are thrown away from σ before it is matched against T_2. $T_1 \| T_2$ matches σ provided both cut-down sequences match their respective trace-expressions.

As an example of the benefit of trace-expressions over regular-expressions consider the component network outlined in Figure 3.7. There are four components, Y, A, B, and Q. Components Y, A, and B are intended to behave in such a way that their input and output events alternate. For example, component A has input wire d and output wire a, and its intended behaviour is described by the regular expression $[d^*; a^*]$. Nothing is required of component Q. Components A and B share the same input d and their operation is therefore related whereas the operation of components Y and Q is independent of either A or B. An attempt to merge the regular-expressions for all four components into a single regular-expression denoting the intended behaviour for the complete component network raises the following two concerns:

- $[d^*; a^*]$ only matches network executions in which either d or a change value on every transition. What about transitions where any of x, y, b, c, p, or q are changing value instead? What about transitions where no wire changes value? A similar concern applies to $[d^*; b^*]$ and $[x^*; y^*]$.

- Every valid network execution must match both $[d^*; a^*]$ and $[d^*; b^*]$. Every d^* must therefore be followed by both an a^* and a b^* before the next d^* can occur, but the order in which a^* and b^* happen is not important. The number of ways to permutate n items is $n!$ and $n!$ grows exponentially with n. An enumeration of all $n!$ possible sequences as a regular-expression therefore also grows exponentially with n. Can this be avoided?

If all of the regular-expressions are viewed as a trace-expression then both concerns disappear: the first concern is resolved by the way in which symbols are thrown away from circuit executions based on Σ and the second concern is resolved by using the parallel composition operator. As a result, $[d^*; a^*] \parallel [d^*; b^*] \parallel [x^*; y^*]$ denotes the correct combined trace-expression for the complete component network.

A migration of trace-expressions to reason over network propositions requires a meaningful redefinition of the sort Σ_T of trace-expression T in a proposition-oriented context. Since the purpose of Σ_T is to identify those wires which should be observed by T, redefinition of Σ_T in a proposition-oriented context requires that the concept of "observability" be extended first to network propositions themselves. If the network proposition is a^*, a^+, or a^- then the extraction of wire name a is not difficult, however extracting the appropriate wire names from $b^* \oplus c^*$ or $b^* \wedge (c = 0)$ requires further thought: consider the trace-expression $T = [a^*; (b^* \ \wedge \ c{=}0)]$ where the intention is for events on a and b to alternate but for c to be at level 0 every time an event on b occurs. T does not assert anything regarding *when* or *how* c should change level, it merely asserts that $c = 0$ when b^*. Events on c must therefore be ignored by T even though c appears somewhere in the definition of T. Conversely if $T = [a^*; (b^* \ \oplus \ c^*)]$ where the intention is for events on a to alternate with events on either b or c, then T must not ignore events on c. This distinction can be related closely to the addition of *read-arcs* into an STG [69], where the value of certain signals can be passively bound to the enabling of a transition firing.

In the context of proposition-oriented behaviours, the determination of the sort Σ_p for a network proposition p relates directly to those events that *must* happen for p to be satisfied. For example $b^* \wedge (c=0)$ necessitates an event on b whereas $b^* \oplus c^*$ necessitates an event on either b or c. If p is satisfiable when no wires change value, such as $\neg a^*$, then p is passive and cannot meaningfully observe any wire including a. *Active propositions* are a special subset of network propositions, each of which can be meaningfully argued as observing at least one wire. An active proposition can be viewed as a formal definition of the generic notion of an event: every active proposition identifies a well-defined set of instants in time. For example, $a^* \wedge b = 0$ is an active proposition whereas $\neg a^*$ is not. Furthermore, $a^* \wedge b^*$ cannot be an active proposition since it attempts to package two events into one, an assertion that can always be violated if enough absolute-time resolution is available. If $a^* \wedge b^*$ cannot be active then $a^* \vee b^*$ cannot be active either since it includes, as a subset, the same impossible instants as $a^* \wedge b^*$. $a^* \oplus b^*$ is however a valid active proposition since it excludes those instants where both a^* and b^* can occur.

Active propositions are defined in Section 3.6.1 below, and proposition-oriented trace-expressions over active propositions are then defined in Section 3.6.2.

3.6.1 Active Propositions

If N is a component network and p is a network proposition for N then define:

- $\mathsf{stable}(W \subseteq \mathsf{wires}(N)) \overset{\text{def}}{=} \bigwedge_{w \in W}(w = w')$ is a network proposition requiring that all wires $w \in W$ don't change their value.
- $\mathsf{allstable}(N) \overset{\text{def}}{=} \mathsf{stable}(\mathsf{wires}(N))$.
- $\mathsf{changeone}(w \in \mathsf{wires}(N)) \overset{\text{def}}{=} \bigwedge_{x \in \mathsf{wires}(N)-\{w\}}(x = x') \;\wedge\; w^*$ is a network proposition matching any network transition where only wire w changes value.
- $\mathsf{activeset}(p) \overset{\text{def}}{=} \{w \in \mathsf{wires}(N) \mid p \wedge \mathsf{changeone}(w) \not\equiv \mathsf{F}\}$ is the set of all wires w where a value change on w alone might satisfy p. If $w \in \mathsf{activeset}(p)$ then p is said to be *active* on w.
- $\mathsf{twoevents}(W \subseteq \mathsf{wires}(N)) \overset{\text{def}}{=} \bigvee_{w_1 \in W} \bigvee_{w_2 \in W-\{w_1\}}(w_1^* \;\wedge\; w_2^*)$ is a network proposition that is satisfied whenever at least two wires in W change value simultaneously.

Proposition	activeset(p)	Condition				Active?
		live	**notempty**	**complete**	**mutex**	
$\neg a^*$	–	No	–	–	–	No
$a^* \wedge b^*$	$\emptyset$	Yes	No	–	–	No
$a^* \vee (b^* \wedge c^*)$	$\{a\}$	Yes	Yes	No	–	No
$a^* \vee b^*$	$\{a, b\}$	Yes	Yes	Yes	No	No
a^*	$\{a\}$	Yes	Yes	Yes	Yes	Yes
$a^* \oplus b^*$	$\{a, b\}$	Yes	Yes	Yes	Yes	Yes
$a^* \wedge b{=}0$	$\{a\}$	Yes	Yes	Yes	Yes	Yes

Table 3.1: Comparison of active and inactive propositions.

A network proposition p is an *active proposition* if the following conditions are met:

- **live**: $p \wedge \mathsf{allstable}(N) \equiv \mathsf{F}$. At least one wire in $\mathsf{wires}(N)$ must change value if p is to be satisfied.
- **notempty**: $\mathsf{activeset}(p) \neq \emptyset$. p must be active on at least one wire.
- **complete**: $\mathsf{stable}(\mathsf{activeset}(p)) \wedge p \equiv \mathsf{F}$. p cannot be satisfied if all wires on which it is active remain stable.
- **mutex**: $\mathsf{twoevents}(\mathsf{activeset}(p)) \wedge p \equiv \mathsf{F}$. Any pair of events on two different active wires for p must be mutually exclusive. If $|\mathsf{activeset}(p)| = 1$ then this condition is vacuously true.

Conditions **live** and **notempty** encapsulate the generic concept of an event: something which cannot be satisfied when no wires change value, and can be satisfied when only one wire changes value. Conditions **complete** and **mutex** assert practical constraints to ensure that every active proposition is meaningful in a relative-time execution model: for example $a^* \wedge b^*$ cannot be active since simultaneity is impossible in a relative-time execution model. A demonstration of each of the four conditions being applied to distinguish active and inactive propositions is shown in Table 3.1.

3.6.2 Trace-Expressions over Active Propositions

If N is a component network then the set *Texp* of all trace-expressions over active propositions for N is defined recursively as follows:

$$\begin{aligned} \mathit{Texp} \;&::=\; p \;\big|\; \mathit{Texp}\,;\mathit{Texp} \;\big|\; \mathit{Texp} \,|\, \mathit{Texp} \;\big|\; [\mathit{Texp}] \;\big|\; \mathit{Texp}\;\&\;\mathit{Texp} \;\big|\; \mathit{Texp} \parallel \mathit{Texp} \;\big| \\ &\qquad \mathit{Texp} \not< \mathit{Texp} \\ p \;&::=\; \text{any active network proposition.} \end{aligned}$$

If $T \in \mathit{Texp}$ then define the sort Σ_T of T recursively on the structure of T as follows:

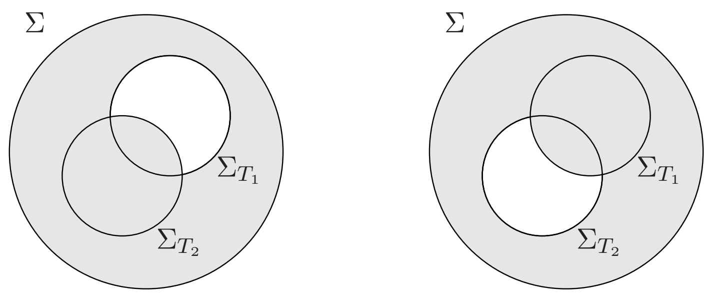
- $\Sigma_p \overset{\text{def}}{=} \mathsf{activeset}(p)$.
- $\Sigma_{T_1;T_2} = \Sigma_{T_1|T_2} = \Sigma_{T_1\&T_2} = \Sigma_{T_1\|T_2} = \Sigma_{T_1 \not< T_2} \overset{\text{def}}{=} \Sigma_{T_1} \cup \Sigma_{T_2}$.
- $\Sigma_{[T]} \overset{\text{def}}{=} \Sigma_T$.

If p is an active proposition for component network N and $\Sigma \subseteq \mathsf{wires}(N)$ is a set of wires in N, then define $\mathsf{tother}(\Sigma, p) \overset{\text{def}}{=} \mathsf{stable}(\Sigma - \mathsf{activeset}(p))$ to be a network proposition labelling any network transition where every wire $w \in \Sigma - \mathsf{activeset}(p)$ remains stable. If N is a component network and $W \subseteq \mathsf{wires}(N)$ is a set of wires in N then define $\mathsf{tlabels}(W \subseteq \mathsf{wires}(N)) \overset{\text{def}}{=} \mathsf{labels}(\neg\mathsf{stable}(W))$ to denote the set of network transitions in which at least one wire $w \in W$ changes value. Define the *language* of a trace-expression $T \in \mathit{Texp}$ as $L_{te}(T) \overset{\text{def}}{=} \mathsf{tlang}(\Sigma_T, T)$ where $\mathsf{tlang}(\Sigma, T)$ is defined recursively on the structure of T as follows:

- $\mathsf{tlang}(\Sigma, p) \overset{\text{def}}{=} \bigcup_{l \in \mathsf{labels}(p \wedge \mathsf{tother}(\Sigma,p))} \{\sigma \mid (\sigma \upharpoonright \mathsf{tlabels}(\Sigma) = \langle l \rangle_{\mathsf{alltrans}(N)}\}$ denotes the set of all finite sequences over $\mathsf{alltrans}(N)$ which when restricted to transitions in $\mathsf{tlabels}(\Sigma)$ leave the singleton sequence $\langle l \rangle_{\mathsf{alltrans}(N)}$ for some transition in $l \in \mathsf{labels}(p \wedge \mathsf{tother}(\Sigma, p))$.
- $\mathsf{tlang}(\Sigma, T_1 \,;T_2) \overset{\text{def}}{=} \{\sigma_1\sigma_2 \mid \sigma_1 \in \mathsf{tlang}(\Sigma, T_1) \wedge \sigma_2 \in \mathsf{tlang}(\Sigma, T_2)\}$ denotes expression concatenation.
- $\mathsf{tlang}(\Sigma, T_1 \mid T_2) \overset{\text{def}}{=} \mathsf{tlang}(\Sigma, T_1) \cup \mathsf{tlang}(\Sigma, T_2)$ denotes expression alternation.
- $\mathsf{tlang}(\Sigma, [T]) \overset{\text{def}}{=} (\mathsf{tlang}(\Sigma, T))^*$ denotes arbitrary repetition.
- $\mathsf{tlang}(\Sigma, T_1 \ \&\ T_2) \overset{\text{def}}{=} \mathsf{tlang}(\Sigma, T_2) \cap \mathsf{tlang}(\Sigma, T_1)$ denotes expression intersection.
- $\mathsf{tlang}(\Sigma, T_1 \ \|\ T_2) \overset{\text{def}}{=} \mathsf{tlang}(\Sigma - (\Sigma_{T_2} - \Sigma_{T_1}), T_1) \cap \mathsf{tlang}(\Sigma - (\Sigma_{T_1} - \Sigma_{T_2}), T_2)$ denotes parallel composition.
- $\mathsf{tlang}(\Sigma, T_1 \not< T_2) \overset{\text{def}}{=} (\mathbf{stpref}\, \mathsf{tlang}(\Sigma - (\Sigma_{T_2} - \Sigma_{T_1}), T_1)) \cap \mathsf{tlang}(\Sigma - (\Sigma_{T_1} - \Sigma_{T_2}), T_2)$ denotes biased composition.

If N is a component network then a network execution $\sigma \in L(N)$ is said to *match* $T \in \mathit{Texp}$ if $\sigma^2 \in \mathbf{pref}\, L_{te}(T)$.

This definition of $\mathsf{tlang}(\Sigma, T)$ relates closely to conventional trace-expression semantics that have been presented in the literature [31, 34]. $\mathsf{tlang}(\Sigma, T)$ differs from convention



▨ $= \Sigma - (\Sigma_{T_1} - \Sigma_{T_2})$ ▨ $= \Sigma - (\Sigma_{T_2} - \Sigma_{T_1})$

Figure 3.8: Sub-alphabets for $\mathsf{tlang}_{\Sigma}(T_1 \nless T_2)$ and $\mathsf{tlang}_{\Sigma}(T_1 \parallel T_2)$.

in that $L_{te}(T)$ is defined over sequences of network transitions not sequences of wire names. The purpose of $\mathsf{tlabels}(\Sigma)$ in case $\mathsf{tlang}(\Sigma, p)$ is to permit the singleton sequence $\langle l \rangle_{\mathsf{alltrans}(N)}$ to be surrounded by an arbitrary number of network transitions in which no wire $w \in \Sigma$ changes value. The augmentation of $l \in \mathsf{labels}(p)$ to $l \in \mathsf{labels}(p \wedge \mathsf{tother}(\Sigma, p))$ in case $\mathsf{tlang}(\Sigma, p)$ is to ensure that any network transition l belongs to both $\mathsf{labels}(p)$ *and* $\mathsf{labels}(\mathsf{tother}(\Sigma, p))$. The augmentation is necessary since component network executions may incur multiple wire value changes per transition: if $\sigma \in L_{te}(T)$ and $\sigma(i) \in \mathsf{alltrans}(N)$ identifies a change on more than one wire, then at most one of the wires whose value is changing can belong to the sort Σ_T of T. For example, if $T = [a^*; b^*; c^*]$ then when proposition a^* is being matched wires b and c must remain stable. Since $\Sigma_{[a^*;b^*;c^*]} = \{a, b, c\}$, $\Sigma_T - \mathsf{activeset}(a^*) = \{b, c\}$ and therefore $\mathsf{tother}(\Sigma_T, a^*) \equiv \mathsf{stable}(\{b, c\})$ as required.

In the case of $L_{te}(T_1 \parallel T_2)$ the parallel composition of T_1 and T_2 must be performed. If wire $w \in \Sigma_{T_1} - \Sigma_{T_2}$ then w is connected to T_1 but not connected to T_2 and consequently w must be removed from the sort Σ passed to $\mathsf{tlang}(\Sigma, T_2)$. If w were included in $\mathsf{tlang}(\Sigma, T_2)$ then T_2 would erroneously assert that wire w never change value. Conversely if wire $w \in \Sigma_{T_2} - \Sigma_{T_1}$ then w is connected to T_2 but not connected to T_1 and must not be placed in the sort Σ passed to $\mathsf{tlang}(\Sigma, T_2)$. Figure 3.8 shows two Venn diagrams that visualise the resulting sort modifications performed by $\mathsf{tlang}(T_1 \parallel T_2)$.

Case $\mathsf{tlang}(T_1 \nless T_2)$ denotes a new operator called biased composition. The purpose of $T_1 \nless T_2$ is to denote a special form of parallel composition that only matches network executions σ where the restricted version of σ^2 must match T_1 *before* it matches T_2. Biased composition is discussed in more detail in Chapter 6 where is it used to describe a generic form of relative timing assumption that is similar to, but more expressive than, the chain-constraints of Negulescu and Peeters [78].

Example applications of proposition-oriented trace-expressions to the verification of asynchronous circuits are given in Chapter 6.

3.7 Summary

The purpose of this chapter was to introduce proposition-oriented behaviours and to explain their foundation. This introduction was given in the context of a gate network circuit model called a component network. A set-theoretic semantics for two different proposition-oriented notations was given. The first of these notations extended regular-expressions to reason over proposition-oriented behaviours, and the second notation extended trace-expressions to reason over proposition-oriented behaviours. The extension of trace-expressions to reason over proposition-oriented behaviours led to the introduction of active propositions, an abstract notion of event more generic than x^*, x^+ or x^-, but specific enough to retain the validity of a conventional trace-expression semantics.

Chapter 4
Proposition Automata

4.1 Introduction

A fundamental assumption behind classical automata-theory is the notion of a finite alphabet of input symbols Σ. Quantification over $a \in \Sigma$ is used extensively in state-minimisation algorithms, and table-based representations of transition relations are often indexed with (current state, input symbol) pairs [1]. Whenever such quantification or indexing occurs it is ubiquitously assumed that no two different input symbols may be matched simultaneously. In the context of proposition-oriented specifications the "alphabet" of symbols available to a designer are network propositions, and it is entirely possible for an arbitrary pair p, q of network propositions to be matched simultaneously, even if $p \not\equiv q$. For example, T, x^*, x^+ are all matched by any component network transition in which $x = 0$ and $x' = 1$.

The loss of exclusivity between pairs of input symbols occurs because a network proposition p is merely a convenient shorthand for a set $\mathsf{labels}(p)$ of component network transitions. If N is a component network then the underlying alphabet of symbols available to a designer writing a proposition-oriented specification is in fact the set of all component network transitions, $\mathsf{alltrans}(N)$. For example, if $[x^*; y^*]$ is a proposition-oriented regular expression, then x^* is semantically equivalent to the alternation $(t_0|t_1|\cdots|t_n)$, where each of the t_i are symbols from the alphabet $\mathsf{alltrans}(N)$, and $\{t_0, t_1, \ldots, t_n\} = \mathsf{labels}(x^*)$. The expressivity of any proposition-oriented notation is therefore bounded by the expressivity of any symbol-based notation in which the alphabet of symbols is $\mathsf{alltrans}(N)$. The primitive component definitiion in Section 3.3.1 was purposefully defined using a wire-state only model [16] so that any subset l of $\mathsf{alltrans}(N)$ can be identified by at least one network proposition p such that $\mathsf{labels}(p) = l$. In this sense proposition-oriented behaviours over component networks are therefore *equally* expressive to any symbol-based notation over $\mathsf{alltrans}(N)$.

If component network N has $n = |\mathsf{wires}(N)|$ wires then there are 2^n possible value assignments to those n wires. In the case of a network proposition p the size of $\mathsf{labels}(p)$ therefore grows exponentially with n and an explicit enumeration of $\mathsf{labels}(p)$ in the encoding of p must therefore also grow exponentially with n. Conversely, a direct encoding of p as a boolean-expression is of constant size with respect to n, and in practice an encoding of p as $\mathsf{labels}(p)$ would therefore be extremely inefficient.

The purpose of this chapter is to introduce a special type of finite automaton, called a *proposition automaton*, whose transitions are labelled with boolean expressions, and therefore whose exclusivity between input symbols cannot be assumed. Proposition automata are important because they formalise an efficient representation with which to encode both proposition-oriented regular-expressions and proposition-oriented trace-expressions.

In the sections that follow, Section 4.2 defines proposition automata. Section 4.3 describes the translation from proposition-oriented regular-expressions to proposition automata, and Section 4.4 describes the translation from proposition-oriented trace-expressions to proposition automata. Section 4.5 combines these translations by introducing a new notation, referred to as proposition-expressions, in which proposition-oriented trace-expressions are embedded into proposition-oriented regular-expressions using an explicit **trace** construct. Section 4.6 explains how to convert a non-deterministic proposition automaton into a deterministic proposition automaton, and Section 4.7 describes a minimisation algorithm for deterministic proposition automata. The purpose of deterministic proposition automata is discussed further in Chapter 5, where deterministic proposition automata and component networks form the foundation of a proposition-oriented verification procedure based on Binary Decision Diagrams.

4.2 Proposition Automaton

If N is a component network then a proposition automaton M over N is a tuple $\langle S,\ I,\ A,\ T\rangle$ where:

- $M.\mathsf{states} \overset{\text{def}}{=} S,\ M.\mathsf{init} \overset{\text{def}}{=} I,\ M.\mathsf{acc} \overset{\text{def}}{=} A,\ M.\mathsf{trans} \overset{\text{def}}{=} T.$
- S is a finite set of states.
- $I \subseteq S$ is a set of initial states and $A \subseteq S$ is a set of accepting states.
- $T \subseteq S \times \mathsf{bexp}(\mathsf{dwires}(N)) \times S$ is a transition relation. Require that for all $(s_1, s_2) \in S \times S$ there exists exactly one p such that $(s_1, p, s_2) \in T$. Every pair

(s_1, s_2) of states in $S \times S$ therefore identifies a unique boolean expression $p \in \mathsf{bexp}(\mathsf{dwires}(N))$ such that $(s_1, p, s_2) \in T$.

A *path* in M is any sequence σ over S such that if $|\sigma| > 0$ then $\sigma(0) \in I$ and if $|\sigma| > 1$ then for all $i \in \{0 \ldots |\sigma| - 2\}$ it is true that $(\sigma(i), \mathsf{F}, \sigma(i+1)) \notin T$. If $\mathsf{last}(\sigma) \in A$ then σ is said to be an *accepting path* in M.

Define any sequence α over $\mathsf{alltrans}(N)$ to be an *input sequence* for M. If $\alpha \neq \varepsilon$ is an input sequence for M, and σ is a path in M, then σ is said to *support* α in M if $|\sigma| = |\alpha| + 1$ and for all $i \in \{0, \ldots, |\alpha| - 1\}$ with $(\sigma(i), p, \sigma(i+1)) \in T$, it is true that $\alpha(i) \in \mathsf{labels}(p)$. Given any input sequence $\alpha \neq \varepsilon$ over $\mathsf{alltrans}(N)$, M is said to *accept* α if there exists a supporting path for α in M that is also accepting. If $A \cap I \neq \emptyset$ then M is said to accept the empty sequence ε. Given any $(s_1, p, s_2) \in T$, if $p \equiv \mathsf{F}$ then M may never transition from state s_1 to state s_2. Conversely, if $p \equiv \mathsf{T}$ then M can always transition from state s_1 to state s_2. Define the language $L(M)$ of M as

$$L(M) \overset{\text{def}}{=} \{\alpha \text{ over } \mathsf{alltrans}(N) \mid M \text{ accepts } \alpha\}.$$

If any two $(s_1, p, s_2) \in T$ and $(s_1, q, s_3) \in T$ with $s_2 \neq s_3$ have $p \wedge q \not\equiv \mathsf{F}$ then M is said to be *non-deterministic* since any $l \in \mathsf{labels}(p \wedge q)$ may result in a transition in M from state s_1 to state s_2 or from state s_1 to state s_3. M is *deterministic* if it is not non-deterministic and $|M.\mathsf{init}| = 1$. If M is deterministic then every input sequence α has at most one accepting path in M.

4.2.1 Normalising Proposition Automata

If T does not contain a unique (s_1, p, s_2) for each (s_1, s_2) then M is said to be *de-normalised.* A de-normalised M can be normalised by computing **normalise**(M), see Algorithm 4.1. **normalise**(M) iteratively reduces any two $(s_1, p, s_2) \in T$ and $(s_1, q, s_2) \in T$ such that $p \neq q$ to $(s_1, (p \vee q), s_2)$. New transitions (s_1, p, s_2) can be added to a normalised M without incurring de-normalisation by calling **addtrans**(M, s_1, p, s_2), see Algorithm 4.2.

In the pseudo-code examples that follow, the expression "$M := \textsc{new}\ \mathsf{propaut}_N(S, I, A, T)$" denotes the creation of a new proposition automaton $M = \langle S,\ I,\ A,\ T_n \rangle$ where $T_n =$ **normalise**(S, T). The unqualified term "$\mathsf{propaut}_N$" is used to denote the set of all possible proposition automata over component network N. If m_1, m_2 are proposition automata and $X \subseteq m_1.\mathsf{states}$ and $Y \subseteq m_2.\mathsf{states}$ then $X \uplus Y$ denotes the disjoint union of X and Y and is defined as $X \uplus Y \overset{\text{def}}{=} \{(s,1) \mid s \in X\} \cup \{(s,2) \mid s \in Y\}$. If $X \subseteq m_1.\mathsf{trans}$ and $Y \subseteq m_2.\mathsf{trans}$ then $X \uplus Y$ also denotes the disjoint union of X and Y, however it is defined instead as $X \uplus Y \overset{\text{def}}{=} \{((s_1,1),p,(s_2,1)) \mid (s_1,p,s_2) \in X\} \cup \{((s_1,2),p,(s_2,2)) \mid (s_1,p,s_2) \in Y\}$.

```
FUNCTION normalise ($S$, REF $M$ : $\mathsf{propaut}_N$)
  FOR ALL $(s_0, s_1) \in M.\mathsf{states} \times M.\mathsf{states}$ DO
    $T := T \cup \{(s_0, \mathsf{F}, s_1)\}$
    WHILE there exists an $(s_0,p,s_1) \in M.\mathsf{trans}$ and $(s_0,q,s_1) \in M.\mathsf{trans}$ with $p \neq q$ DO
      $M.\mathsf{trans} := (M.\mathsf{trans} - \{(s_0,p,s_1),(s_0,q,s_1)\}) \cup \{(s_0,(p \vee q),s_1)\}$
    END WHILE
  END FOR
END FUNCTION
```

Algorithm 4.1: Normalise a proposition-oriented automaton.

```
FUNCTION addtrans (REF $M$ : $\mathsf{propaut}_N$, $s_0$ : $M.\mathsf{states}$, $p$ : $\mathsf{bexp}(\mathsf{dwires}(N))$,
$s_1$ : $M.\mathsf{states}$)
  LET $(s_0,q,s_1) \in M.\mathsf{trans}$
  $M.\mathsf{trans} := (M.\mathsf{trans} - \{(s_o,q,s_1)\}) \cup \{(s_0,(q \vee p),s_1)\}$
END FUNCTION
```

Algorithm 4.2: Add a new transition to a proposition automaton.

```
FUNCTION kill_dead (REF $M$ : $\mathsf{propaut}_N$)
  WHILE $\exists x \in (M.\mathsf{states} - M.\mathsf{acc})$ such that $\forall (x,p,y) \in M.\mathsf{trans}.\ p \equiv \mathsf{F}$ or $x = y$ DO
    $M.\mathsf{states} := M.\mathsf{states} - \{x\}$
    $M.\mathsf{acc} := M.\mathsf{acc} - \{x\}$
    $M.\mathsf{init} := M.\mathsf{init} - \{x\}$
    $M.\mathsf{trans} := M.\mathsf{trans} - \{(x,p,y) \mid \exists y,p.\ (x,p,y) \in M.\mathsf{trans}\}$
  END WHILE
END FUNCTION
```

Algorithm 4.3: Remove dead states from a proposition-oriented automaton.

4.2.2 Dead States in Proposition Automata

Let M be a proposition automaton over component network N. A state $s \in M.\mathsf{states}$ is *dead* if $s \notin M.\mathsf{acc}$ and every $(s,p,d) \in M.\mathsf{trans}$ with $s \neq d$ has $p \equiv \mathsf{F}$. Any state $s \in M$ that is dead may be removed from M without affecting $L(M)$. Since the removal of a dead state may subsequently render states that were previously alive as dead, a procedure to remove *all* dead states from M must therefore have the capacity to iterate over several cycles, see procedure **kill_dead**(M) in Algorithm 4.3.

```
FUNCTION complete (REF $M$ : $\mathsf{propaut}_N$)
  kill_dead($M$)
  FOR ALL $s \in M.\mathsf{states}$ DO
    addtrans($M$, $s$, $\neg\mathsf{exits}(s)$, $\mathsf{err}$)
  END FOR
  addtrans($M$, $\mathsf{err}$, $\mathsf{T}$, $\mathsf{err}$)
  $M.\mathsf{states} := M.\mathsf{states} \cup \{\mathsf{err}\}$
END FUNCTION
```

Algorithm 4.4: Completion of a proposition automaton.

4.2.3 Complete Proposition Automata

Let M be a proposition automaton over component network N. If $\alpha \in L(N)$ is a network execution such that $\alpha^2 \in L(M)$ then M must have at least one accepting supporting path σ in M. Conversely, if $\alpha \notin L(M)$ then there can be no accepting supporting paths σ for α^2 in M. However, if $\alpha \notin L(M)$ then there may or may not be *non*-accepting supporting paths for α^2 in M. If *any* sequence α over $\mathsf{alltrans}(N)$ has some supporting path in M then M is said to be *complete*.

For each state $s \in M.\mathsf{states}$ let $\mathsf{exits}(s) \overset{\text{def}}{=} \bigvee_{(s,p,x) \in M.\mathsf{trans}}(p)$ be a network proposition identifying all component network transitions supported by M from state s. A procedure **complete**(M) for making M complete without affecting $L(M)$ is shown in Algorithm 4.4. Execution of **complete**(M) inserts a transition from each state $s \in M.\mathsf{states}$ to a special dead state err under any input symbol not already matched by an existing transition from s. State err is then made terminal through the addition of transition $(\mathsf{err}, \mathsf{T}, \mathsf{err})$. Since **complete**$(M)$ invokes **kill_dead**(M) prior to the insertion of state err, on completion of **complete**(M) state err identifies the only dead state in M. As a result, *any* input sequence $\alpha^2 \notin \mathbf{pref}\, L(M)$ can *only* be supported by paths in M that terminate in state err.

4.2.4 Support Sets for Proposition Automata

Let M be a proposition automaton over component network N. For any sequence α over $\mathsf{alltrans}(N)$ define $\mathsf{support}(M, \alpha) \overset{\text{def}}{=} \{\sigma \mid \sigma \text{ is a supporting path for } \alpha^2 \text{ in } M\}$. If M is deterministic then every α over $\mathsf{alltrans}(N)$ has $|\mathsf{support}(M, \alpha)| \leq 1$. If M is complete then every α over $\mathsf{alltrans}(N)$ has $|\mathit{support}(M, \alpha)| \geq 1$. If M is both complete and deterministic then every α over $\mathsf{alltrans}(N)$ has $|\mathsf{support}(M, \alpha)| = 1$.

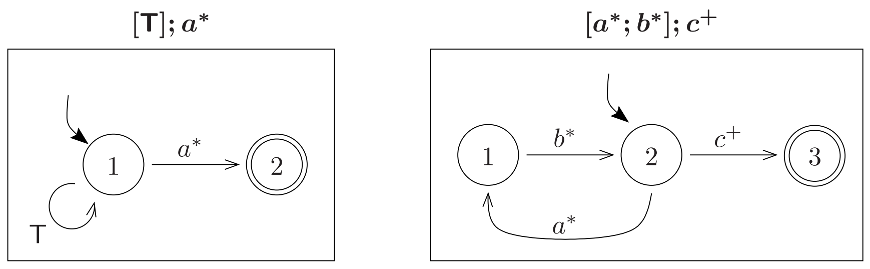


Figure 4.1: Proposition-oriented regular-expressions as proposition automata.

4.3 Regular-Expressions as Proposition Automata

This section explains how proposition-oriented regular-expressions can be translated into non-deterministic proposition automata with the same language. Let E be a proposition-oriented regular expression over component network N. A procedure **mk_regexp**(E) for constructing a proposition automaton M such that $L_{re}(E) = L(M)$ is shown in Algorithm 4.7. **mk_regexp**(E) is recursive on the structure of E and its operation parallels the definition of $L_{re}(E)$ as discussed in Section 3.5.

In the case of **mk_regexp**$(n_1 \,\&\, n_2)$ a proposition automaton M must be constructed that accepts the intersection of $L(m_1)$ and $L(m_2)$. This intersection can be computed as the *product automaton* **mk_product**(m_1, m_2) of m_1 and m_2, see Algorithm 4.5: A product automaton m_{12} = **mk_product**(m_1, m_2) for proposition automata m_1 and m_2 denotes the concurrent execution of both m_1 and m_2 and is accepting if and only if both m_1 and m_2 are accepting.

A proposition automaton has no notion of an empty transition, and as a result both cases **mk_regexp**$([n])$ and **mk_regexp**$(n_1; n_2)$ employ the use of a special procedure **addempty**, see Algorithm 4.6. The purpose of **addempty**(M, X, Y) is to mimic the insertion of an empty transition in M from every state $x \in X$ to every state $y \in Y$: Each transition $(s, p, x) \in M.\mathsf{trans}$ with $x \in X$ is duplicated as a transition (s, p, y) to all states $y \in Y$. Conversely, each transition $(y, p, s) \in M.\mathsf{trans}$ with $y \in Y$ is duplicated as a transition (x, p, s) from all states $x \in X$.

Two examples of proposition-oriented regular-expressions and their equivalent proposition automata are shown in Figure 4.1.

```
FUNCTION mk_product  ($M_1$ : $\mathsf{propaut}_N$, $M_2$ : $\mathsf{propaut}_N$) : $\mathsf{propaut}_N$
  $M$ := NEW $\mathsf{propaut}_N(M_1.\mathsf{states} \times M_2.\mathsf{states},\ M_1.\mathsf{init} \times M_2.\mathsf{init},\ M_1.\mathsf{acc} \times M_2.\mathsf{acc},\ \emptyset)$
  FOR ALL $((x_1, p, y_1), (x_2, q, y_2)) \in (M_1.\mathsf{trans} \times M_2.\mathsf{trans})$ DO
    addtrans$(M, (x_1, x_2), p \wedge q, (y_1, y_2))$
  END FOR
  RETURN $M$
END FUNCTION
```

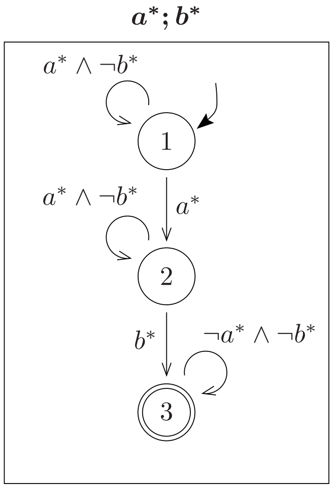
Algorithm 4.5: Construct a product automaton.

```
FUNCTION addempty  (REF $M$ : $\mathsf{propaut}_N$, $X \subseteq M.\mathsf{states}$, $Y \subseteq M.\mathsf{states}$)
  FOR ALL $(f, p, t) \in M.\mathsf{trans}$ such that $t \in X$ DO
    FOR ALL $y \in Y$ DO addtrans$(M,\ f,\ p,\ y)$ END FOR
  END FOR
  FOR ALL $(f, p, t) \in M.\mathsf{trans}$ such that $f \in Y$ DO
    FOR ALL $x \in X$ DO addtrans$(M,\ x,\ p,\ t)$ END FOR
  END FOR
END FUNCTION
```

Algorithm 4.6: Add empty transitions to a proposition automaton.

```
FUNCTION mk_regexp  ($D$ : node) : $\mathsf{propaut}_N$
  SWITCH $D$
    CASE $n_1; n_2$
      LET $m_1$ = mk_regexp$(n_1)$  AND  $m_2$ = mk_regexp$(n_2)$
      LET $S = (m_1.\mathsf{states} \uplus m_2.\mathsf{states})$  AND  $T = (m_1.\mathsf{trans} \uplus m_2.\mathsf{trans})$
      $M$ := NEW $\mathsf{propaut}_N(S,\ (m_1.\mathsf{init} \uplus \emptyset),\ (\emptyset \uplus m_2.\mathsf{acc}),\ T)$
    CASE $n_1 \,|\, n_2$
      LET $m_1$ = mk_regexp$(n_1)$  AND  $m_2$ = mk_regexp$(n_2)$
      LET $S = (m_1.\mathsf{states} \uplus m_2.\mathsf{states})$  AND  $T = (m_1.\mathsf{trans} \uplus m_2.\mathsf{trans})$
      $M$ := NEW $\mathsf{propaut}_N(S,\ (m_1.\mathsf{init} \uplus m_2.\mathsf{init}),\ (m_1.\mathsf{acc} \uplus m_2.\mathsf{acc}),\ T)$
    CASE $n_1 \,\&\, n_2$
      LET $m_1$ = mk_regexp$(n_1)$  AND  $m_2$ = mk_regexp$(n_2)$
      $M$ := mk_product$(m_1,\ m_2)$
    CASE $[n]$
      LET $m$ = mk_regexp$(n)$
      $M$ := NEW $\mathsf{propaut}_N(m.\mathsf{states},\ m.\mathsf{init},\ (m.\mathsf{init} \cup m.\mathsf{acc}),\ m.\mathsf{trans})$
      addempty$(M,\ m.\mathsf{acc},\ m.\mathsf{init})$
    CASE $p$
      $M$ := NEW $\mathsf{propaut}_N(\{0, 1\},\ \{0\},\ \{1\},\ \{(0, p, 1)\})$
  END SWITCH
  RETURN $M$
END FUNCTION
```

Algorithm 4.7: Make a proposition automaton from a regular-expression.



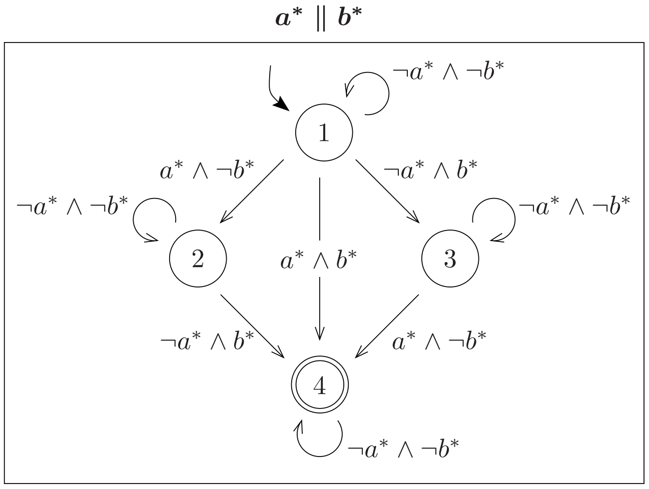


Figure 4.2: Proposition-oriented trace-expressions as proposition automata.

4.4 Trace-Expressions as Proposition Automata

This section explains how proposition-oriented trace-expressions can be translated into non-deterministic proposition automata with the same language. Let E be a proposition-oriented trace expression over component network N. A procedure **mk_trexp**(E, Σ_E) for constructing a proposition automaton M such that $L_{te}(E) = L(M)$ is shown in Algorithm 4.8. **mk_trexp**(E, Σ) is recursive on both E and Σ_E and its operation parallels the definition of $\mathsf{tlang}(E,\ \Sigma)$ as discussed in Section 3.6.2.

If $E \neq n_1 \parallel n_2$, $E \neq n_1 \not< n_2$, and $E \neq p$ then execution of **mk_trexp**(E, Σ) can be likened to execution of **mk_regexp**(E). If $E = p$ then $M =$ **mk_trexp**(p, Σ) differs from **mk_regexp**(p) in that M must also restrict all input sequences σ to $\sigma \upharpoonright \mathsf{tlabels}(\Sigma)$. In the context of M this restriction can be encoded through the addition of transitions $(0, \mathsf{stable}(\Sigma), 0)$ and $(1, \mathsf{stable}(\Sigma), 1)$ to $M.\mathsf{trans}$: M must *ignore* any input symbol $i \in \mathsf{alltrans}(N)$ such that $i \in \mathsf{stable}(\Sigma)$.

If $E = n_1 \parallel n_2$ or $E = n_1 \not< n_2$ then the each recursive call to **mk_trexp** must be given a restricted alphabet $\Sigma_E - (\Sigma_{n_2} - \Sigma_{n_1})$ or $\Sigma_E - (\Sigma_{n_2} - \Sigma_{n_1})$ as discussed in Section 3.6.2. In order to perform the appropriate restriction the alphabets Σ_{n_1} and Σ_{n_2} must be determined. If E is a proposition-oriented trace-expression then Σ_E can be trivially computed as shown by procedure **sigma**(E) in Algorithm 4.9.

If $E = n_1 \not< n_2$ then M must accept the intersection of **stpref** $L(m_1)$ and $L(m_2)$. This intersection can be computed using procedure **mk_product** provided a proposition automaton sp_m_1 can be computed such that $L(sp\_m_1) =$ **stpref** $L(m_1)$. If m_1 has no dead states then computation of sp_m_1 can be achieved merely by swapping accepting and non-accepting states in m_1: Since m_1 has no dead states, for every path σ_1 in m_1 that is not accepting there must exist another sequence $\sigma_2 \neq \varepsilon$ over $m_1.\mathsf{states}$ such that $\sigma_1\sigma_2$ is an accepting path in m_1.

Two examples of proposition-oriented trace-expressions and their equivalent proposition automata are shown in Figure 4.1.

```
FUNCTION mk_trexp  ($D$ : node, $\Sigma$ : wire set) : $\mathsf{propaut}_N$
  SWITCH $D$
    CASE $n_1 ; n_2$
      LET $m_1 = \mathbf{mk\_trexp}(n_1,\ \Sigma)$ AND $m_2 = \mathbf{mk\_trexp}(n_2,\ \Sigma)$
      LET $S = (m_1.\mathsf{states} \uplus m_2.\mathsf{states})$ AND $T = (m_1.\mathsf{trans} \uplus m_2.\mathsf{trans})$
      $M$ := NEW $\mathsf{propaut}_N(S,\ (m_1.\mathsf{init} \uplus \emptyset),\ (\emptyset \uplus m_2.\mathsf{acc}),\ T)$
    CASE $n_1 \mid n_2$
      LET $m_1 = \mathbf{mk\_trexp}(n_1,\ \Sigma)$ AND $m_2 = \mathbf{mk\_trexp}(n_2,\ \Sigma)$
      LET $S = (m_1.\mathsf{states} \uplus m_2.\mathsf{states})$ AND $T = (m_1.\mathsf{trans} \uplus m_2.\mathsf{trans})$
      $M$ := NEW $\mathsf{propaut}_N(S,\ (m_1.\mathsf{init} \uplus m_2.\mathsf{init}),\ (m_1.\mathsf{acc} \uplus m_2.\mathsf{acc}),\ T)$
    CASE $n_1 \,\&\, n_2$
      LET $m_1 = \mathbf{mk\_trexp}(n_1,\ \Sigma)$ AND $m_2 = \mathbf{mk\_trexp}(n_2,\ \Sigma)$
      $M$ := $\mathbf{mk\_product}(m_1,\ m_2)$
    CASE $n_1 \parallel n_2$
      LET $\Sigma_1 = \mathbf{sigma}(n_1)$ AND $\Sigma_2 = \mathbf{sigma}(n_2)$
      LET $m_1 = \mathbf{mk\_trexp}(n_1,\ \Sigma-(\Sigma_2-\Sigma_1))$ AND $m_2 = \mathbf{mk\_trexp}(n_2,\ \Sigma-(\Sigma_1-\Sigma_2))$
      $M$ := $\mathbf{mk\_product}(m_1,\ m_2)$
    CASE $n_1 \not< n_2$
      LET $\Sigma_1 = \mathbf{sigma}(n_1)$ AND $\Sigma_2 = \mathbf{sigma}(n_2)$
      LET $m_1 = \mathbf{mk\_trexp}(n_1,\ \Sigma-(\Sigma_2-\Sigma_1))$ AND $m_2 = \mathbf{mk\_trexp}(n_2,\ \Sigma-(\Sigma_1-\Sigma_2))$
      $\mathbf{kill\_dead}(m_1)$
      $sp\_m_1$ := NEW $\mathsf{propaut}_N(m_1.\mathsf{states},\ m_1.\mathsf{init},\ (m_1.\mathsf{states} - m_1.\mathsf{acc}),\ m_1.\mathsf{trans})$
      $M$ := $\mathbf{mk\_product}(sp\_m_1,\ m_2)$
    CASE $[n]$
      LET $m = \mathbf{mk\_trexp}(n)$
      $M$ := NEW $\mathsf{propaut}_N(m.\mathsf{states},\ m.\mathsf{init},\ (m.\mathsf{init} \cup m.\mathsf{acc}),\ m.\mathsf{trans})$
      $\mathbf{addempty}(M,\ m.\mathsf{acc},\ m.\mathsf{init})$
    CASE $p$
      ensure that $p$ is active
      LET $T = \{(0, p \wedge \mathsf{tother}(\Sigma, p), 1), (0, \mathsf{stable}(\Sigma), 0), (1, \mathsf{stable}(\Sigma), 1)\}$
      $M$ := NEW $\mathsf{propaut}_N(\{0,1\},\ \{0\},\ \{1\},\ T)$
  END SWITCH
  RETURN $M$
END FUNCTION
```

Algorithm 4.8: Make a proposition automaton from a trace-expression.

```
FUNCTION sigma  ($D$ : node) : wire set
  SWITCH $D$
    CASE $n_1; n_2 \;\big|\; n_1 \,|\, n_2 \;\big|\; n_1 \,\&\, n_2 \;\big|\; n_1 \parallel n_2 \;\big|\; n_1 \not< n_2$
      RETURN $\mathbf{sigma}(n_1) \cup \mathbf{sigma}(n_2)$
    CASE $[n]$
      RETURN $\mathbf{sigma}(n)$
    CASE $p$
      RETURN $\mathsf{activeset}(p)$
  END SWITCH
END FUNCTION
```

Algorithm 4.9: Computation of Σ for a trace-expression.

4.5 Proposition-Expressions

Sections 4.3 and 4.4 combined demonstrate that both proposition-oriented regular expressions and proposition-oriented trace-expressions can be translated into proposition automata. In this sense both types of expression are indistinguishable once translated into proposition automata. In particular, every proposition-oriented trace-expression has an equivalent proposition-oriented regular-expression. For example, the proposition oriented trace-expression $r^+; a^-$ is equivalent to the proposition-oriented regular-expression $[\neg r^* \wedge \neg a^*]; r^+; [\neg r^* \wedge \neg a^*]; a^+; [\neg r^* \wedge \neg a^*]$. Proposition-oriented trace-expressions can be embedded into proposition-oriented regular-expressions by providing an explicit **trace** construct to identify when to assert a trace-expression semantics as follows:

$$\begin{array}{rcl} \mathit{Pexp} & ::= & p \;\big|\; \mathit{Pexp}\,;\mathit{Pexp} \;\big|\; \mathit{Pexp}\,|\,\mathit{Pexp} \;\big|\; [\mathit{Pexp}] \;\big|\; \mathit{Pexp}\,\&\,\mathit{Pexp} \;\big|\; \mathbf{trace}(\mathit{Texp}) \\ p & ::= & \text{any safe network proposition.} \\ & & \text{(for a definition of } \mathit{Texp} \text{ see Section 3.6.2)} \end{array}$$

If expression $E \in \mathit{Pexp}$ then E is said to be a *proposition-expression*, and may be translated into a proposition automaton using procedure **mk_pexp**, see Algorithm 4.10. **mk_pexp** is identical to procedure **mk_regexp** except that an extra case statement has been added in Line 20 to pass all occurrences of **trace**(E) to procedure **mk_trexp**(E, **sigma**(E)). If E is a proposition-expression then **trace**(E) differs from E in that it implies E be treated as a proposition-oriented trace-expression rather than a proposition-oriented regular-expression.

An embedding of trace-expressions within regular-expressions results in a flexible notation where a trace-expression semantics can be asserted locally within sub-expressions as desired. In the context of proposition-oriented behaviours, where alphabet symbols are network propositions, this flexibility enables a variety of behaviours to be specified that have no equivalents outside of a proposition-oriented domain. The benefits of proposition-expressions are discussed further in Chapter 6.

```
FUNCTION mk_pexp  ($D$ : node) : $\mathsf{propaut}_N$
  SWITCH $D$
    CASE $n_1; n_2$
      LET $m_1 = \mathbf{mk\_pexp}(n_1)$  AND  $m_2 = \mathbf{mk\_pexp}(n_2)$
      LET $S = (m_1.\mathsf{states} \uplus m_2.\mathsf{states})$  AND  $T = (m_1.\mathsf{trans} \uplus m_2.\mathsf{trans})$
      $M$ := NEW $\mathsf{propaut}_N(S,\ (m_1.\mathsf{init} \uplus \emptyset),\ (\emptyset \uplus m_2.\mathsf{acc}),\ T)$
    CASE $n_1 \,|\, n_2$
      LET $m_1 = \mathbf{mk\_pexp}(n_1)$  AND  $m_2 = \mathbf{mk\_pexp}(n_2)$
      LET $S = (m_1.\mathsf{states} \uplus m_2.\mathsf{states})$  AND  $T = (m_1.\mathsf{trans} \uplus m_2.\mathsf{trans})$
      $M$ := NEW $\mathsf{propaut}_N(S,\ (m_1.\mathsf{init} \uplus m_2.\mathsf{init}),\ (m_1.\mathsf{acc} \uplus m_2.\mathsf{acc}),\ T)$
    CASE $n_1 \,\&\, n_2$
      LET $m_1 = \mathbf{mk\_pexp}(n_1)$  AND  $m_2 = \mathbf{mk\_pexp}(n_2)$
      $M$ := $\mathbf{mk\_product}(m_1,\ m_2)$
    CASE $[n]$
      LET $m = \mathbf{mk\_pexp}(n)$
      $M$ := NEW $\mathsf{propaut}_N(m.\mathsf{states},\ m.\mathsf{init},\ (m.\mathsf{init} \cup m.\mathsf{acc}),\ m.\mathsf{trans})$
      $\mathbf{addempty}(M,\ m.\mathsf{acc},\ m.\mathsf{init})$
    CASE $p$
      $M$ := NEW $\mathsf{propaut}_N(\{0,1\},\ \{0\},\ \{1\},\ \{(0,p,1)\})$
20: CASE $\mathbf{trace}(n)$
      $M$ := $\mathbf{mk\_trexp}(n, \mathbf{sigma}(E))$
  END SWITCH
  RETURN $M$
END FUNCTION
```

Algorithm 4.10: Make a proposition automaton from a proposition-expression.

4.6 Deterministic Proposition Automata

The purpose of this section is to explain how a non-deterministic proposition automaton can be translated into a deterministic proposition automaton with the same language. If N is a component network and M is a proposition automaton over N then conversion of M to a deterministic proposition automaton parallels conversion of conventional finite automata into deterministic finite automata using a powerset construction [48]. Conversion using proposition automata differs from conversion using conventional finite automata in that input symbols to proposition automata are not mutually-exclusive. For example, let $\mathsf{wires}(N) = \{a, b\}$ and suppose M is as shown in Figure 4.3(a). Each of the two transi-

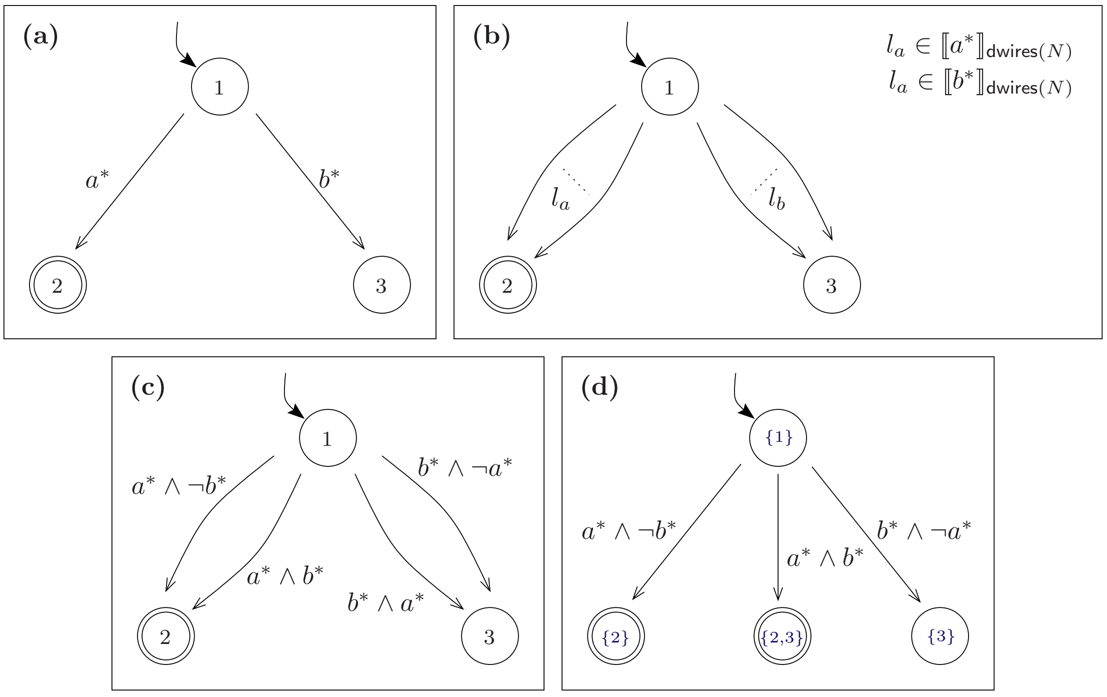


Figure 4.3: Proposition automaton label exclusivity.

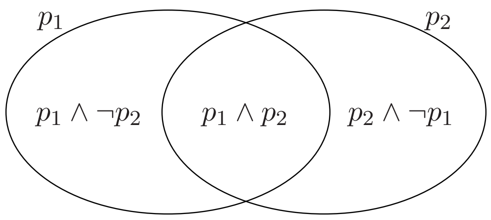
tion labels a^* and b^* identifies a set of boolean value assignments to the wire names in $\mathsf{dwires}(N)$, see Figure 4.3(b). Since certain value assignments $l \in [\![a^* \wedge b^*]\!]_{\mathsf{dwires}(N)}$ are identified by both transition labels, M is therefore non-deterministic. To convert M to a deterministic proposition automaton using a powerset construction, $[\![a^*]\!]_{\mathsf{dwires}(N)}$ and $[\![b^*]\!]_{\mathsf{dwires}(N)}$ must first be partitioned into mutually-exclusive subsets, each of which can be identified by a new boolean expression, see Figure 4.3(c). A powerset construction can then be applied to Figure 4.3(c) resulting in the deterministic automaton shown in Figure 4.3(d).

Let P and Q be sets of boolean expressions over $\mathsf{dwires}(N)$. Define Q to be a *cover* of P if properties **mutex**, **sufficient**, and **complete** hold as follows:

- **mutex**: $\forall q_1 \in Q, q_2 \in Q - \{q_1\}.\ q_1 \wedge q_2 \equiv \mathsf{F}$
  No two different q_1, q_2 share a common value assignment.

- **sufficient**: $\forall p \in P.\ \exists X \subseteq Q.\ p \equiv \bigvee_{x \in X}(x)$
  Every $p \in P$ can be formed as the disjunction of a subset of Q.

- **complete**: $\bigvee_{p \in P}(p) \equiv \bigvee_{q \in Q}(q)$
  P and Q identify the same set of value assignments.



sufficient

$p_1 \equiv (p_1 \wedge \neg p_2) \vee (p_1 \wedge p_2)$

$p_2 \equiv (p_2 \wedge \neg p_1) \vee (p_1 \wedge p_2)$

complete

$(p_1 \wedge \neg p_2) \vee (p_1 \wedge p_2) \vee (p_2 \wedge \neg p_1) \equiv p_1 \vee p_2$

(a) **(b)**

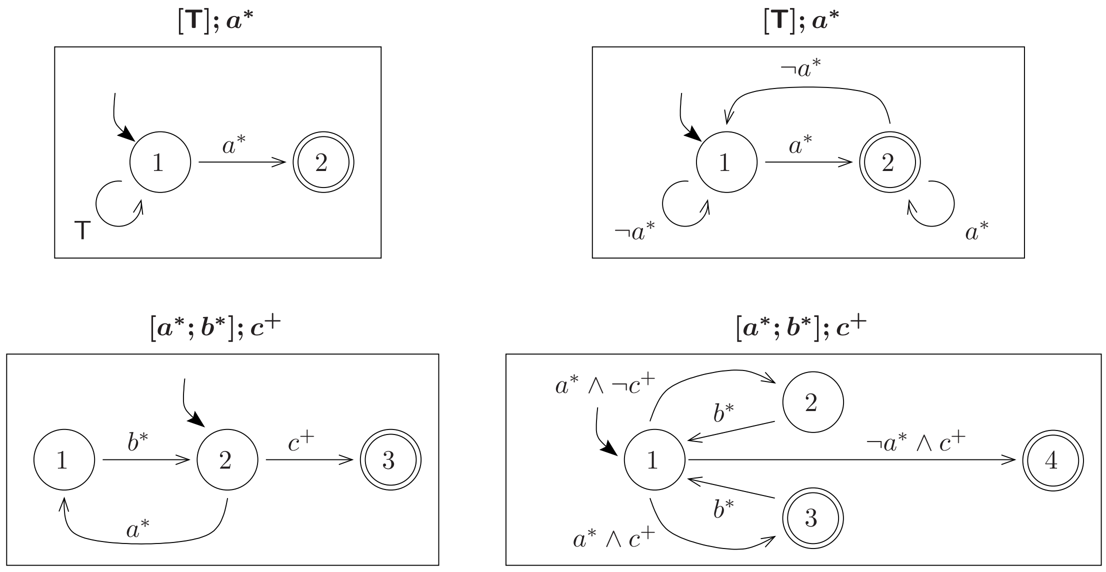
Figure 4.4: Boolean expression exclusivity.

If $Q = P$ then properties **sufficient** and **complete** can be trivially satisfied. Hence, if property **mutex** holds for P then P is a cover of P. If property **mutex** does not hold for P then procedure **cover** can be called to construct a cover, **cover**(P), of P as shown in Algorithm 4.11. Execution of **cover**(P) equates to the repeated refinement of any pair $\{p_1, p_2\} \subseteq P$ such that $p_1 \wedge p_2 \not\equiv \mathsf{F}$ to a triple $\{(p_1 \wedge \neg p_2),\ (p_1 \wedge p_2),\ (p_2 \wedge \neg p_1)\}$, see Figure 4.4(a). Each refinement removes at least one violation of property **mutex** and is invariant on properties **sufficient** and **complete**, see Figure 4.4(b). On completion **cover**(P) satisfies all three properties **mutex**, **sufficient** and **complete**, and therefore $Q =$ **cover**(P) denotes a valid cover of P as required.

```
FUNCTION cover ($P$ : bexp(dwires($N$)) set) : bexp (dwires (N)) set
  WHILE there exists a $\{p_1, p_2\} \subseteq P$ such that $p_1 \wedge p_2 \not\equiv \mathsf{F}$ DO
    $P := (P - \{p_1, p_2\}) \cup \{(p_1 \wedge \neg p_2),\ (p_1 \wedge p_2),\ (p_2 \wedge \neg p_1)\}$
  END WHILE
  RETURN $P - \{\mathsf{F}\}$
END FUNCTION
```

Algorithm 4.11: Construct a mutually-exclusive cover set.

If $P = \{p \mid \exists s_1, s_2.\ (s_1, p, s_2) \in M.\mathsf{trans}\}$ denotes the set of all transition labels in M then **cover**(P) identifies a minimal set of "conventional" input symbols from which every $p \in P$ can be constructed as an alternation. In the worst case every $p \in$ **cover**(P) has $|[\![p]\!]_{\mathsf{dwires}(N)}| = 1$, and **cover**($P$) is in bijection with the set $[\![\bigvee_{p \in P}(p)]\!]_{\mathsf{dwires}(N)} \subseteq \mathsf{bool}(\mathsf{dwires}(N))$, denoting all boolean value assignments to variables in $\mathsf{dwires}(N)$ that are matched by some transition label in M. In the context of Figure 4.3, Figure 4.3(b) equates to such a worst case partitioning whereas Figure 4.3(c) equates to a minimal partitioning such as that returned by **cover**($\{a^*, b^*\}$).



(a) Non-Deterministic **(b) Deterministic**

Figure 4.5: Deterministic and non-deterministic proposition automata.

A procedure **mk_dfa**(M) for constructing a deterministic proposition automaton from a non-deterministic proposition automaton M is shown in Algorithm 4.12. The operation of **mk_dfa**(M) equates to that of a powerset construction in which a set *todo* $\subseteq 2^{M.\mathsf{states}}$ is used to identify reachable powerset states that have yet to be processed. The initial value of *todo* is set to $\{M.\mathsf{init}\}$. For every $S_1 \in todo$ the set P of all transition labels leading from each $s_1 \in S_1$ is constructed. For every $q \in \mathbf{cover}(P)$ the set $S_2 \subseteq M.\mathsf{states}$ of possible destination states for each $s_1 \in S_1$ is determined, and if $S_2 \neq \emptyset$ then transition (S_1, q, S_2) is added to the powerset transition relation trans. Predicate $p \wedge q \equiv q$ is used in Line 7 to determine if $q \in \mathbf{cover}(P)$ belongs to the subset $X \subseteq \mathbf{cover}(P)$ for which $\bigvee_{x \in X}(x) \equiv p$.

Two examples of non-deterministic proposition automata and their deterministic counterparts are shown in Figure 4.5.

4.7 Deterministic Proposition Automata Minimisation

The purpose of this section is to demonstrate the application of cover sets to the problem of deterministic proposition automaton minimisation. Cover sets are important since they permit an algorithm operating on proposition automata to enumerate over a set

```
FUNCTION mk_dfa ($M$ : $\mathsf{propaut}_N$) : $\mathsf{propaut}_N$
  $todo$ := $\mathsf{states}$ := $\mathsf{init}$ := $\{M.\mathsf{init}\}$
  WHILE $todo \neq \emptyset$ DO
    pick an $S_1 \in todo$
    $P := \{p : \mathsf{bexp}\ (\mathsf{dwires}\ (\mathrm{N})) \mid \exists s_1 \in S_1, s_2 \in M.\mathsf{states}.\ (s_1, p, s_2) \in M.\mathsf{trans}\}$
    FOR ALL $q \in \mathbf{cover}(P)$ DO
7:    $S_2 := \{s_2 \in M.\mathsf{states} \mid \exists s_1 \in S_1, p \in P.\ (s_1, p, s_2) \in M.\mathsf{trans}$ and $p \wedge q \equiv q\}$
      IF $S_2 \neq \emptyset$ THEN
        $\mathsf{trans}$:= $\mathsf{trans}$ $\cup\{(S_1, q, S_2)\}$
        IF $S_2 \notin \mathsf{states}$ THEN
          $todo$ := $todo \cup S_2$
          $\mathsf{states}$ := $\mathsf{states} \cup S_2$
        END IF
      END IF
    END FOR
    $todo$ := $todo - S_1$
  END WHILE
  $\mathsf{acc} := \{S \in \mathsf{states} \mid \exists s \in S.\ s \in M.\mathsf{acc}\}$
  RETURN NEW $\mathsf{propaut}_N(\mathsf{states}, \mathsf{init}, \mathsf{acc}, \mathsf{trans})$
END FUNCTION
```

Algorithm 4.12: Conversion to a deterministic proposition automaton.

of mutually-exclusive input symbols. In the context of classical automata minimisation, input symbol enumeration is ubiquitous, since it is over sequences of input symbols that the equivalence of two states is defined: Let M be a deterministic proposition automaton over component network N. A pair s_0, s_1 of states in M are *equivalent* if $L(M_0) = L(M_1)$ where $M_i \overset{\text{def}}{=} \mathsf{propaut}_N(M.\mathsf{states}, \{s_i\}, M.\mathsf{acc}, M.\mathsf{trans}))$ is a proposition automaton accepting input sequences in M that start from state s_i. A recursive procedure $\mathbf{equiv}(M, s_0, s_1, \emptyset)$ for determining the equivalence of two states s_0, s_1 in M is shown in Algorithm 4.13. The operation of **equiv** parallels that of its classical finite automaton counterpart and can be explained as follows:

If $s_0 \in M.\mathsf{acc}$ and $s_1 \notin M.\mathsf{acc}$ then s_0, s_1 are not equivalent since $\varepsilon \in L(M_0)$ whereas $\varepsilon \notin L(M_1)$. If s_0, s_1 are both accepting or both non-accepting then equivalence of s_0 and s_1 requires recursion on pairs d_0, d_1 of states reachable from s_0, s_1 under every possible input symbol $l \in \mathsf{bool}(\mathsf{dwires}(N))$. Since M is deterministic the set $P_0 = \{p \mid \exists y \in M.\mathsf{states}.\ (s_0, p, y) \in M.\mathsf{trans}\}$ and the set $P_1 = \{p \mid \exists y \in M.\mathsf{states}.\ (s_1, p, y) \in M.\mathsf{trans}\}$ are both valid covers of themselves, however this does not imply that $P_0 \cup P_1$ denotes a valid cover of itself. Consequently, the set

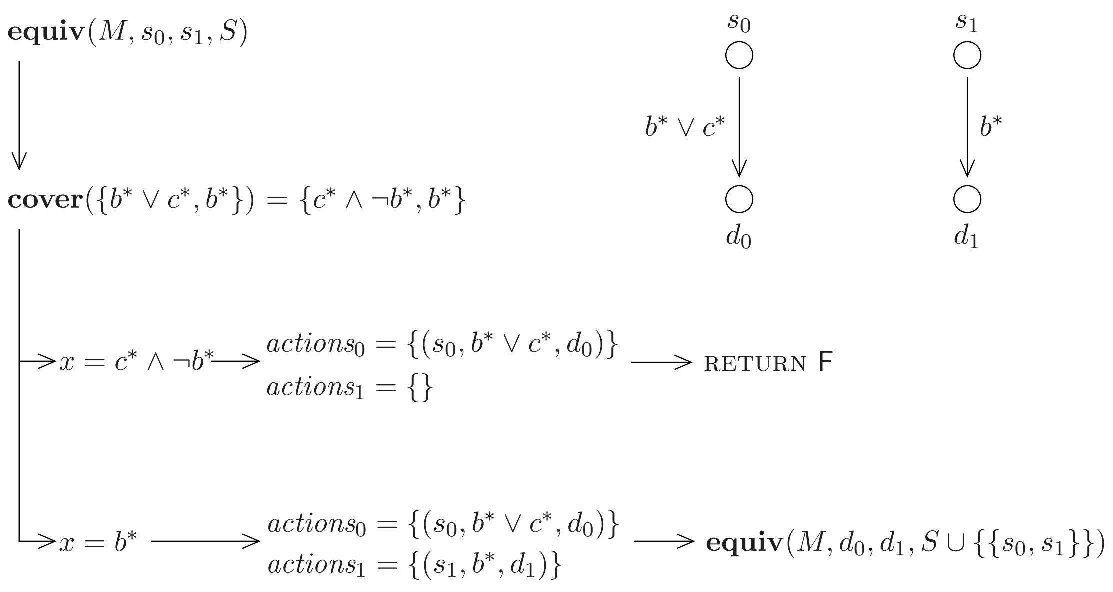

$\mathbf{equiv}(M, s_0, s_1, S)$
s_0
s_1
$b^* \vee c^*$
b^*
d_0
d_1
$\mathbf{cover}(\{b^* \vee c^*, b^*\}) = \{c^* \wedge \neg b^*, b^*\}$
$x = c^* \wedge \neg b^*$
$actions_0 = \{(s_0, b^* \vee c^*, d_0)\}$
$actions_1 = \{\}$
RETURN F
$x = b^*$
$actions_0 = \{(s_0, b^* \vee c^*, d_0)\}$
$actions_1 = \{(s_1, b^*, d_1)\}$
$\mathbf{equiv}(M, d_0, d_1, S \cup \{\{s_0, s_1\}\})$

Figure 4.6: An example of recursion in procedure equiv.

$Q = \mathbf{cover}(P_0 \cup P_1) = \mathbf{cover}(\{p \mid \exists x \in \{s_0, s_1\}, y \in M.\mathsf{states}.\ (x, p, y) \in M.\mathsf{trans}\})$ is used to denote a valid cover of all transition labels possible from state s_0 or state s_1. For each $x \in Q$, $actions_0$ denotes the set of transitions possible from s_0 under input symbol x, and $actions_1$ denotes the set of transitions possible from s_1 under the same input symbol x. Since M is deterministic $|actions_0| \leq 1$ and $|actions_1| \leq 1$, and since $Q = \mathbf{cover}(P_0 \cup P_1)$ either $|actions_0| > 0$ or $|actions_1| > 0$. If $|actions_0| \neq |actions_1|$ then s_0 cannot be equivalent to s_1 since only one of s_0 and s_1 can transition on input symbol x. If $|actions_0| = |actions_1| = 1$ then s_0 is not equivalent to s_1 only if d_0 is not equivalent to d_1. An example of recursion in procedure **equiv** is shown in Figure 4.6. The purpose of variable S in Line 4 is to detect any loops within the recursion of **equiv**. A return value of T on detection of such a loop equates to the assertion that any pair s_0, s_1 of states in M are equivalent until such time as it can be shown otherwise.

Procedure **equiv** provides a means to minimise the number of states in a deterministic proposition by iterative refinement on an equivalence relation between states, see procedure **mk_min_dfa** in Algorithm 4.14. If M is a deterministic proposition automaton over component network N then $M_{min} = \mathbf{mk\_min\_dfa}(M)$ denotes a deterministic proposition automaton with the minimum number of states possible such that $L(M_{min}) = L(M)$. The operation of **mk_min_dfa** equates to the iterative refinement of state equivalence from both above and below: Refinement from above is denoted by a relation E identifying states

```
FUNCTION equiv  ($M$ : $\mathsf{propaut}_N$, $s_0$ : $M.\mathsf{states}$, $s_1$ : $M.\mathsf{states}$, $S \subseteq M.\mathsf{states} \times M.\mathsf{states}$)
: $\{\mathsf{T}, \mathsf{F}\}$
   IF $s_0 \in M.\mathsf{acc}$ and $s_1 \notin M.\mathsf{acc}$ THEN RETURN $\mathsf{F}$ END IF
   IF $s_1 \in M.\mathsf{acc}$ and $s_0 \notin M.\mathsf{acc}$ THEN RETURN $\mathsf{F}$ END IF
4: IF $\{s_0, s_1\} \in S$ THEN RETURN $\mathsf{T}$ END IF
   LET $Q = \mathbf{cover}(\{p \mid \exists x \in \{s_0, s_1\}, y \in M.\mathsf{states}.\ (x, p, y) \in M.\mathsf{trans}\})$
   FOR ALL $x \in Q$ DO
      LET $actions_0 = \{(s_0, p, d_0) \in M.\mathsf{trans} \mid p \wedge x \equiv x\}$
      LET $actions_1 = \{(s_1, q, d_1) \in M.\mathsf{trans} \mid q \wedge x \equiv x\}$
      IF $|actions_0| = |actions_1| = 1$ THEN
         LET $\{(s_0, p, d_0)\} = actions_0$ AND $\{(s_1, q, d_1)\} = actions_1$
         IF $\neg\mathbf{equiv}(M,\ d_0,\ d_1,\ S \cup \{\{s_0, s_1\}\})$ THEN RETURN $\mathsf{F}$ END IF
      ELSE
         RETURN $\mathsf{F}$
      END IF
   END FOR
   RETURN $\mathsf{T}$
END FUNCTION
```

Algorithm 4.13: Determine if two proposition automaton states are equivalent.

```
FUNCTION mk_min_dfa  ($M$ : $\mathsf{propaut}_N$) : $\mathsf{propaut}_N$
   $D := \{\{s_0, s_1\} \mid s_0 \in (M.\mathsf{states} - M.\mathsf{acc})$ and $s_1 \in M.\mathsf{acc}\}$.
   $E := \{\{s\} \mid s \in M.\mathsf{states}\}$
   FOR ALL $(s_0, s_1) \in M.\mathsf{states} \times M.\mathsf{states}$ such that $s_0 \neq s_1$ DO
      IF $\{s_0, s_1\} \notin D$ and $s_0 \notin [s_1]_E$ THEN
         IF $\mathbf{equiv}(M,\ s_0,\ s_1,\ \emptyset)$ THEN
            $E := (E - \{[s_0]_E, [s_1]_E\}) \cup \{[s_0]_E \cup [s_1]_E\}$.
         ELSE
            $D := D \cup \{\{s_0, s_1\}\}$
         END IF
      END IF
   END FOR
   $M_{min}$ := NEW $\mathsf{propaut}_N(E,\ \{[s]_E \mid s \in M.\mathsf{init}\},\ \{[s]_E \mid s \in M.\mathsf{acc}\},\ \emptyset)$
   FOR ALL $(e_1, e_2) \in E \times E$ DO
      FOR ALL $(s_1, s_2) \in e_1 \times e_2$ such that $(s_1, p, s_2) \in M.\mathsf{trans}$ DO
         $\mathbf{addtrans}(M_{min},\ e_1,\ p,\ e_2)$
      END FOR
   END FOR
   RETURN $M_{min}$
END FUNCTION
```

Algorithm 4.14: Minimise a deterministic proposition automaton.

that are known to be equivalent. Refinement from below is denoted by a relation D identifying states that are known to be *distinguishable.* Relation E is symmetric, reflexive, and transitive whereas relation D is symmetric and necessarily *not* reflexive. Consequently, **mk_min_dfa** represents E as a partition of the set $M.\mathsf{states}$, and D as a set of unordered state pairs $\{s_0, s_1\} \subseteq M.\mathsf{states}$. The initial value of D is set to distinguish accepting and non-accepting states, and the initial value of E places every $s \in M.\mathsf{states}$ in an equivalence class on its own. After initialising E and D, procedure **mk_min_dfa** then proceeds to determine equivalence or distinguishability for every pair $(s_0, s_1) \in M.\mathsf{states} \times M.\mathsf{states}$ by evaluating $\mathbf{equiv}(M, s_0, s_1, \emptyset)$. Note that if $s \in M.\mathsf{states}$ then the expression $[s]_E$ denotes the unique equivalence class $X \in E$ such that $s \in X$.

On completion of refinement on E and D, **mk_min_dfa** then constructs a minimal proposition automaton M_{min} where each state $s \in M_{min}.\mathsf{states}$ denotes an equivalence class in E. The equivalence class containing the initial state of M is the initial state of M_{min} and the equivalence classes for each accepting state in M are the accepting states of M_{min}.

4.7.1 Alternative Minimisation Algorithms

Cover sets denote a generic mapping from proposition automata to conventional finite automata. In this sense cover sets can be used to extend many different algorithms based on conventional finite automata to operate on proposition automata. The minimisation procedure **mk_min_dfa** is only one of many possible minimisation algorithms, and is presented here merely as an example application of cover sets to deterministic proposition automata minimisation. In the context of the verification program described in Chapter 6, deterministic proposition automata minimisation applies only as an optimisation technique to help reduce the total number of states over which reachability analysis is performed. Since proposition automata minimisation time is in practice small when compared to reachability analysis time, an explanation of more efficient minimisation algorithms is also unnecessary for this thesis.

4.8 Summary

The purpose of this chapter was to introduce proposition automata, a special form of finite automata in which transitions are labelled with boolean expressions and are therefore not mutually-exclusive. Proposition automata were shown to encode both proposition-oriented trace-expressions and proposition-oriented regular-expressions efficiently. These encodings led to the introduction of a new notation, referred to as proposition-expressions, in which proposition-oriented trace-expressions were embedded into proposition-oriented regular-expressions using an explicit **trace** construct. A mapping from proposition-automata to conventional finite automata was defined in terms of cover sets, and this mapping used to compute deterministic proposition automata from non-deterministic proposition automata. Cover sets were also used to describe an algorithm for the minimisation of deterministic proposition automata.

Chapter 5
Proposition-Oriented Verification

5.1 Introduction

Proposition automata realise proposition-oriented behaviours in a form well suited to algorithmic manipulation. The purpose of this chapter is to build on proposition automata and describe a simple proposition-oriented verification procedure, **pcheck**, for component networks and proposition automata that is based on symbolic reachability analysis using Binary Decision Diagrams (BDDs). The objective of **pcheck** is to provide a simple but flexible platform on which the benefits of proposition-oriented behaviours can then be evaluated. **pcheck** does not evolve new techniques for formal verification, and is built using methods already well documented in the literature.

In the sections that follow Section 5.2 introduces the satisfaction criterion on which procedure **pcheck** is based. Section 5.3 shows how satisfaction according to this criterion can be determined using reachability analysis on a special form of product automaton called a *network product*. Section 5.3.1 also introduces the notion of a *deterministic* network product, in which each of the underlying proposition automata are assumed to be deterministic. The relationship between non-deterministic and deterministic network products is then used to explain why reachability analysis using deterministic network products is superior to reachability analysis using non-deterministic network products.

Section 5.4 builds on Section 5.3 to describe the symbolic encoding of deterministic network products using BDDs, and Section 5.5 gives an algorithmic definition of procedure **pcheck** based on this encoding. Section 5.6 overviews those BDD-based optimisations that can be applied to procedure **pcheck**, and Section 5.7 introduces a special type of proposition automata called protocol automata whose purpose is to permit the description of circuit-environment contracts using proposition-oriented trace-expressions.

5.2 Satisfaction Criterion

Let N be a component network and let S be a set of proposition-expressions over N. Assign to each expression $E \in S$ a *sense* and *type* as follows:

5.2.1 Sense Assignment

The *sense* of a proposition-expression E can be either always or never:

- If E is of sense always then E refers to behaviours that are *valid*. Define $\mathsf{always}(E) \overset{\text{def}}{=} \{\sigma \text{ over } \mathsf{states}(N) \mid \sigma^2 \in \textbf{pref}\, L(E)\}$ to be the set of network executions σ in N for which $\sigma^2 \in \textbf{pref}\, L(E)$.

- If E is of sense never then E refers to behaviours that are *invalid*. Define $\mathsf{never}(E) \overset{\text{def}}{=} \{\sigma \text{ over } \mathsf{states}(N) \mid \sigma^2 \notin L(E)\}$ to be the set of network executions σ in N such that $\sigma^2 \notin L(E)$.

An assignment of sense to each proposition-expression $E \in S$ serves as an alternative to the explicit extension of grammar *Pexp* to include constructs for prefix-closure, **pref** (E), and expression negation, $!E$. The foundation of this alternative is the observation that in practice use of **pref** (E) and $!E$ is restricted: **pref** (E) is used to assert that the prefix of a valid execution is also valid, and $!E$ is used when a designer is reasoning about invalid behaviours. Since erroneous executions are not prefix-closed and since valid things are rarely described by negation, **pref** $!E$ and **!pref** (E) are in practice never used. The replacement of **pref** , ! with always, never enforces the restriction that use of **pref** , ! can only be at the top level, and never in combination.

5.2.2 Type Assignment

The *type* of a proposition-expression E can be either verify or cut. If E is of type verify, denoted verify $sense(E)$, then E is a specification for N and the conformance of E to N requires determination of the set-containment problem $L(N) \subseteq sense(E)$. If E is of type cut, denoted cut $sense(E)$, then E is a constraint on N and the application of E to N asserts that any $\sigma \in L(N)$ with $\sigma \notin sense(E)$ be removed from $L(N)$.

If $\bigcap_{\mathsf{cut}}$ denotes intersection over all $E \in S$ of type cut, and $\bigcap_{\mathsf{verify}}$ denotes intersection over all $E \in S$ of type verify, then conformance of N to P equates to evaluation of the satisfaction criterion:

$$\forall \sigma \in L(N).\ \sigma \in \textstyle\bigcap_{\mathsf{cut}} sense(E) \Rightarrow \sigma \in \bigcap_{\mathsf{verify}} sense(E)$$

5.3 Network Product

Let $S = \{E_1, \dots, E_n\}$ be a set of proposition-expressions over a component network N, and assign to each $E_i \in S$ a sense and a type as discussed in Section 5.2. For each $E_i \in S$ compute a complete proposition automaton M_i with a single dead state err such that $L(M_i) = L(E_i)$. Let $P = \{M_1, \dots, M_n\}$, and assign to each M_i the same sense and type as E_i.

For each network execution $\sigma \in L(N)$ computation of the set $\mathsf{support}(M_i, \sigma)$ of supporting paths for σ^2 in M_i is sufficient to determine $\sigma \in \mathit{sense}(M_i)$ as follows:

- $\sigma \in \mathsf{always}(M_i)$ if and only if $\exists \alpha \in \mathsf{support}(M_i, \sigma).\ \mathsf{last}(\alpha) \neq \mathsf{err}$. If $\alpha \in \mathsf{support}(M_i, \sigma)$ and $\mathsf{last}(\alpha) \neq \mathsf{err}$, then α is the prefix of an accepting supporting path in M_i and hence $\sigma^2 \in \mathbf{pref}\, L(M_i)$.

- $\sigma \in \mathsf{never}(M_i)$ if and only if $\forall \alpha \in \mathsf{support}(M_i, \sigma).\ \mathsf{last}(\alpha) \notin M_i.\mathsf{acc}$. If every $\alpha \in \mathsf{support}(M_i, \sigma)$ has $\mathsf{last}(\alpha) \notin M_i.\mathsf{acc}$ then there are no accepting supporting paths for σ^2 in M_i and hence $\sigma^2 \notin L(M_i)$.

Since both $\sigma \in \mathsf{always}(M_i)$ and $\sigma \in \mathsf{never}(M_i)$ depend only on $\mathsf{last}(\alpha)$ determination of $\sigma \in \mathit{sense}(M_i)$ may be simplified further as follows:

- Define $\mathsf{slast}(M_i, \sigma) \overset{\text{def}}{=} \{\mathsf{last}(\alpha) \mid \alpha \in \mathsf{support}(M_i, \sigma)\}$ to denote the set of possible ending states for supporting paths of σ^2 in M_i.

- $\sigma \in \mathsf{always}(M_i)$ if and only if $\exists s \in \mathsf{slast}(M_i, \sigma).\ s \neq \mathsf{err}$.

- $\sigma \in \mathsf{never}(M_i)$ if and only if $\forall s \in \mathsf{slast}(M_i, \sigma).\ s \notin M_i.\mathsf{acc}$.

If determination of $\sigma \in \mathit{sense}(M_i)$ depends only on $\mathsf{slast}(M_i, \sigma)$ then any two $\sigma_1 \in L(N)$, $\sigma_2 \in L(N)$ such that $\mathsf{slast}(M_i, \sigma_1) = \mathsf{slast}(M_i, \sigma_2)$ are indistinguishable by $\mathit{sense}(M_i)$. To assert that N conforms to P it is therefore sufficient to consider every tuple $s \in X$ where $X = \{(\mathsf{slast}(M_1, \sigma), \dots, \mathsf{slast}(M_n, \sigma)) \mid \sigma \in L(N)\}$. Define the *network product* $N{\cdot}P$ of N and P to be the finite automaton formed by augmenting each state $s \in \mathsf{states}(N)$ with a set of possible proposition automaton states for every $M_i \in P$ as follows:

- $\mathsf{states}(N{\cdot}P) \overset{\text{def}}{=} \mathsf{states}(N) \times 2^{M_1.\mathsf{states}} \times \cdots \times 2^{M_n.\mathsf{states}}$.

- If $x = (s_N, s_{M_1}, \dots, s_{M_n}) \in \mathsf{states}(N{\cdot}P)$ then let $x{\downarrow_N}$ denote s_N and for each $M_i \in P$ let $x{\downarrow_{M_i}}$ denote s_{M_i}.

- $\mathsf{alltrans}(N{\cdot}P) \overset{\text{def}}{=} \mathsf{states}(N{\cdot}P) \times \mathsf{states}(N{\cdot}P)$.
- If $t \in \mathsf{alltrans}(N{\cdot}P) = (s_1, s_2)$ then let $t{\downarrow_N}$ denote $(s_1{\downarrow_N}, s_2{\downarrow_N})$ and for each $M_i \in P$ let $t{\downarrow_{M_i}}$ denote $(s_1{\downarrow_{M_i}}, s_2{\downarrow_{M_i}})$.
- $\mathsf{image}(M_i \in P, S \subseteq M_i.\mathsf{states}, l \in \mathsf{alltrans}(N)) \overset{\text{def}}{=} \{s_2 \mid \exists (s_1, p, s_2) \in M_i.\mathsf{trans}.\ s_1 \in S$ and $l \in \mathsf{labels}(p)\}$ denotes the set of states in M_i that can be reached from states in S under input symbol l.
- $\mathsf{trans\_set}(M_i \in P, l \in \mathsf{alltrans}(N)) \overset{\text{def}}{=} \{(s, \mathsf{image}(M_i, s, l)) \mid s \in M_i.\mathsf{states}\}$ denotes the set of all $(s, \mathsf{image}(M_i, s, l))$ pairs in M_i.
- $\mathsf{init}(N{\cdot}P) \overset{\text{def}}{=} \{(s, \mathsf{init}(M_1), \ldots, \mathsf{init}(M_n)) \mid s \in \mathsf{init}(N)\}$ is a set of initial network product states.
- $\mathsf{trans}(N{\cdot}P) \overset{\text{def}}{=} \{t \in \mathsf{alltrans}(N{\cdot}P) \mid t{\downarrow_N} \in \mathsf{trans}(N)$ and $\forall M_i \in P.$ $t{\downarrow_{M_i}} \in \mathsf{trans\_set}(M_i)\}$ is a set of network product transitions.

A *path* in $N{\cdot}P$ is any sequence σ over $\mathsf{states}(N{\cdot}P)$ such that if $|\sigma| > 0$ then $\sigma(0) \in \mathsf{init}(N{\cdot}P)$ and if $|\sigma| > 1$ then $(\sigma(i), \sigma(i+1)) \in \mathsf{trans}(N \cdot P)$ for all $i \in \{0 \ldots |\sigma| - 2\}$. Define $L(N{\cdot}P) \overset{\text{def}}{=} \{\sigma \mid \sigma$ is a path in $N{\cdot}P\}$. If $\sigma \in L(N{\cdot}P)$ then define the *network extraction* of σ to be the unique network execution $\sigma{\downarrow_N} \in L(N)$ such that $|\sigma{\downarrow_N}| = |\sigma|$ and $\sigma{\downarrow_N}(i) = \sigma(i){\downarrow_N}$ for all $i \in \{0 \ldots |\sigma| - 1\}$. If $\sigma \in L(N{\cdot}P)$ then by induction on the length of σ, $\mathsf{last}(\sigma){\downarrow_{M_i}} = \mathsf{slast}(M_i, \sigma{\downarrow_N})$. Consequently,

$$\begin{aligned}\{\mathsf{last}(\sigma) \mid \sigma \in L(N{\cdot}P)\} &= \{(\mathsf{last}(\sigma{\downarrow_N}), \mathsf{slast}(M_1, \sigma{\downarrow_N}), \ldots, \mathsf{slast}(M_n, \sigma{\downarrow_N})) \mid \sigma \in L(N{\cdot}P)\} \\ &= \{(\mathsf{last}(\alpha), \mathsf{slast}(M_1, \alpha), \ldots, \mathsf{slast}(M_n, \alpha)) \mid \alpha \in L(N)\}\end{aligned}$$

and conformance of N to P is decidable by considering every *reachable* network product state $s \in \{\mathsf{last}(\sigma) \mid \sigma \in L(N{\cdot}P)\}$.

5.3.1 Deterministic Network Product

If each of the $M_i \in P$ are both deterministic and complete then for every $\alpha \in L(N)$, $|\mathsf{support}(M_i, \alpha)| = 1$. If $\sigma \in L(N{\cdot}P)$ has $|\mathsf{support}(M_i, \sigma{\downarrow_N})| = 1$ then $|\mathsf{slast}(M_i, \sigma{\downarrow_N})| = 1$. Hence, if *every* $\sigma \in L(N{\cdot}P)$ has $|\mathsf{support}(M_i, \alpha)| = 1$ then every $s \in \{\mathsf{last}(\sigma) \mid \sigma \in L(N{\cdot}P)\}$ has $s{\downarrow_{M_i}} \in \{\,\{s\} \mid s \in M_i.\mathsf{states}\}$ and $N{\cdot}P$ may be simplified as follows:

- $\mathsf{states}(N{\cdot}P) \overset{\text{def}}{=} \mathsf{states}(N) \times M_1.\mathsf{states} \times \cdots \times M_n.\mathsf{states}$.

- $\mathsf{init}(N{\cdot}P) \overset{\text{def}}{=} \{(s, I_1, \ldots, I_n) \mid s \in \mathsf{states}(N) \text{ and } \forall M_i \in P.\ \{I_i\} = \mathsf{init}(M_i)\}$.
- $\mathsf{trans}(N{\cdot}P) \overset{\text{def}}{=} \{(x, y) \in \mathsf{alltrans}(N{\cdot}P) \mid (x, y){\downarrow_N} \in \mathsf{trans}(N) \text{ and } \forall M_i \in P.\ \{y{\downarrow_{M_i}}\} = \mathsf{image}(M_i, \{x{\downarrow_{M_i}}\}, (x, y){\downarrow_N})\}$.

The relationship between $2^{M_1.\mathsf{states}} \times \cdots \times 2^{M_n.\mathsf{states}}$ and $M_1.\mathsf{states} \times \cdots \times M_n.\mathsf{states}$ equates directly to the relationship between non-deterministic automata and their deterministic equivalents based on a classical powerset construction. If each $M_i \in P$ is non-deterministic then $N{\cdot}P$ can still be constructed over $M_1.\mathsf{states} \times \cdots \times M_n.\mathsf{states}$ provided each $M_i \in P$ is first converted into a deterministic proposition automata using a procedure such as **mk_dfa**. Conversion of $M_i \in P$ into deterministic proposition automta prior to the construction of $N{\cdot}P$ is superior since $|\textbf{mk\_dfa}(M_i).\mathsf{states}| < 2^{|M_i.\mathsf{states}|}$ is in practice common and $2^{M_1.\mathsf{states}} \times \cdots \times 2^{M_n.\mathsf{states}}$ is therefore excessive when compared to $\textbf{mk\_dfa}(M_1).\mathsf{states} \times \cdots \times \textbf{mk\_dfa}(M_n).\mathsf{states}$. Furthermore, if each $M_i \in P$ is converted to a deterministic proposition automaton independently from the construction of $N{\cdot}P$, then the determinised M_i may also be minimised using a procedure such as $\textbf{mk\_min\_dfa}(M_i)$. In the sections that follow each $M_i \in P$ is therefore required to be both complete and deterministic, and a deterministic network product therefore always constructed.

5.4 Symbolic Encoding Using Binary Decision Diagrams

This section explains how a deterministic network product $N{\cdot}P$ can be constructed symbolically using Binary Decision Diagrams (BDDs). It is assumed throughout the explanation that some form of BDD function library is available to the programmer [10], and that all primitive boolean-expression operators on BDDs are provided for by this library. A generic type $\mathsf{bdd}(V)$ is used to denote the set of all BDDs over the variables in set V. Conceptually, $\mathsf{bdd}(V)$ may be viewed merely as a practical implementation of the set-theoretic type $\mathsf{bexp}(V)$ from Section 3.2.2.

5.4.1 Symbolic Encoding of a Set

Consider an arbitrary finite set S. In order to encode S using BDDs each element $s \in S$ must be identifiable as a BDD. An encoding of S using BDDs is therefore a function $\mathsf{enc} \in S \to \mathsf{bdd}(V)$ of each element $s \in S$ to a BDD $\mathsf{enc}(s)$ over some set V of boolean valued BDD variables. This encoding can be visualised as a labelling of each state $s \in S$ with some set $[\![\mathsf{enc}(s)]\!]_V$ of value assignments to the variables in V. An encoding enc is *valid* if the following two requirements are met:

$$\begin{array}{rcl} S & = & \{s_1, s_2, s_3, s_4, s_5\} \\ V & = & \{v_0, v_1, v_2\} \end{array}$$

s_i	$\mathsf{enc}(s_i)$	$[\![\mathsf{enc}(s_i)]\!]$	state labels
s_1	$\neg v_0 \wedge \neg v_1 \wedge \neg v_2$	$\{\emptyset\}$	000
s_2	$\neg v_0 \wedge \neg v_1 \wedge v_2$	$\{\{v_2\}\}$	001
s_3	$\neg v_0 \wedge v_1 \wedge \neg v_2$	$\{\{v_1\}\}$	010
s_4	$\neg v_0 \wedge v_1 \wedge v_2$	$\{\{v_1, v_2\}\}$	011
s_5	v_0	$\{\{v_0\}, \{v_0, v_2\}, \{v_0, v_1\}, \{v_0, v_1, v_2\}\}$	100, 101, 110, 111

Figure 5.1: Example symbolic encoding of a set.

- $\forall s \in S.\ \mathsf{enc}(s) \not\equiv \mathsf{F}$. Every state $s \in S$ is labelled by at least one value assignment to the variables in V.
- $\forall s_1 \in S, s_2 \in S - \{s_1\}.\ \mathsf{enc}(s_1) \wedge \mathsf{enc}(s_2) \equiv \mathsf{F}$. No two different states share the same label.

An example symbolic encoding of a set is shown in Figure 5.1. Once S has been encoded by a valid encoding function enc any subset $X \subseteq S$ can be encoded as $\mathsf{bddset}(X, \mathsf{enc}) \stackrel{\text{def}}{=} \bigvee_{x \in X}(\mathsf{enc}(s))$. Note that $\mathsf{bddset}(S, \mathsf{enc}) \not\equiv \mathsf{T}$ is possible since not every label in $\mathsf{bool}(V)$ need be assigned to a state in S. If set S_1 is encoded by $\mathsf{enc}_1 \in S_1 \to V_1$ and set S_2 is encoded by $\mathsf{enc}_2 \in S_2 \to V_2$ with $V_1 \cap V_2 = \emptyset$, then the product set $S_1 \times S_2$ can be constructed by forming the product encoding $\mathsf{enc}_{12}(s_1 \in S_1, s_2 \in S_2) \stackrel{\text{def}}{=} \mathsf{enc}_1(s_1) \wedge \mathsf{enc}_2(s_2)$. Note that $\mathsf{bddset}(S_1 \times S_2, \mathsf{enc}_{12}) = \mathsf{bddset}(S_1, \mathsf{enc}_1) \wedge \mathsf{bddset}(S_2, \mathsf{enc}_2)$ and therefore that "$\wedge$" denotes the BDD encoding equivalent of "$\times$".

Since $\mathsf{enc}(s) = \mathsf{bddset}(\{s\}, \mathsf{enc})$, both the element s and the singleton set $\{s\}$ have the same BDD encoding. If $X \subseteq S$ and $Y \subseteq S$ have symbolic BDD encodings $\mathcal{X}$ and $\mathcal{Y}$ respectively, then $\mathcal{X} \wedge \mathcal{Y}$ encodes $X \cap Y$, $\mathcal{X} \vee \mathcal{Y}$ encodes $X \cup Y$, $\neg \mathcal{X}$ encodes $S - X$, and $\mathcal{X} \subseteq_{\mathsf{bdd}} \mathcal{Y} \stackrel{\text{def}}{=} \mathcal{X} \wedge \mathcal{Y} \equiv \mathcal{X}$ encodes the set-containment problem $X \subseteq Y$. Note also that $\mathsf{enc}(s) \in_{\mathsf{bdd}} \mathcal{X} \stackrel{\text{def}}{=} \mathsf{enc}(s) \wedge \mathcal{X} \equiv \mathsf{enc}(s)$ encodes element-containment $s \in X$ identically to set-containment $\{s\} \subseteq X$.

5.4.2 Simple Automaton Encoding

Let F be an automaton with a finite set S of states. Let $I \subseteq S$ denote a set of initial states of F and $T \subseteq S \times S$ denote a set of unlabelled transitions for F. Define $\mathsf{enc}_F \in S \to \mathsf{vars}(F)$ to be a valid encoding function for S to some set $\mathsf{vars}(F)$ of boolean-valued BDD variables. Define $\mathsf{vars}'(F) \stackrel{\text{def}}{=} \{v' \mid v \in \mathsf{vars}(F)\}$ to be a second set of BDD variable names consisting of all names $v \in \mathsf{vars}(F)$ in primed form. If v is a boolean-valued BDD variable then define $\mathsf{prime}(v) \stackrel{\text{def}}{=} v'$ to denote the boolean-valued BDD variable v' and $\mathsf{unprime}(v') \stackrel{\text{def}}{=} v$ to denote the boolean-valued BDD variable v. The domain of prime is the set of all possible unprimed BDD variables and the domain of $\mathsf{unprime}$ is the set of all possible primed BDD variables. It is assumed that a BDD variable may be primed at most once and therefore that the intersection of these two domains is empty. The BDD encodings $\mathcal{S}$, $\mathcal{I}$ and $\mathcal{T}$ of S, I and T respectively are defined as follows:

- $\mathcal{S} \stackrel{\text{def}}{=} \mathsf{bddset}(S, \mathsf{enc}_F)$.
- $\mathcal{I} \stackrel{\text{def}}{=} \mathsf{bddset}(I, \mathsf{enc}_F)$.
- $\mathcal{T} \stackrel{\text{def}}{=} \bigvee_{(s_1,s_2)\in T}(\mathsf{enc}_F(s_1) \wedge (\mathsf{enc}_F(s_2)[\mathsf{prime}]))$ where $\mathsf{enc}_F(s_2)[\mathsf{prime}]$ denotes the BDD obtained after each $v \in \mathsf{vars}(F)$ is replaced by $v' = \mathsf{prime}(v) \in \mathsf{vars}'(F)$ in $\mathsf{enc}_F(s_2)$.

The purpose of $\mathsf{vars}'(F)$ is to identify a second domain of BDD variables that can be used to store a distinct copy of any BDD over $\mathsf{vars}(F)$. In order to represent transitions of F as pairs $(s_1, s_2) \in S \times S$ two distinct encodings of S are required, one for s_1, denoted $\mathsf{enc}_F(s_1)$, and one for s_2, denoted $\mathsf{enc}_F(s_2)[\mathsf{prime}]$. prime and $\mathsf{unprime}$ serve merely as renaming functions that permit a BDD $\mathcal{X}$ over $\mathsf{vars}(F)$ to be transfered to and from $\mathsf{vars}'(F)$ without otherwise changing the subset $X \subseteq S$ encoded by $\mathcal{X}$. If $l \in [\![\mathcal{S}]\!]_{\mathsf{vars}(F)}$ is a label for some $s \in S$ then define $\mathsf{Prime}(l) \stackrel{\text{def}}{=} \{\mathsf{prime}(v) \mid v \in l\}$ and define $\mathsf{Unprime}(l) \stackrel{\text{def}}{=} \{\mathsf{unprime}(v) \mid v \in l\}$ to denote the casting of l to $\mathsf{vars}(F)$ and $\mathsf{vars}(F')$ respectively.

Simple automata encodings formalise some generic constructs that facilitate the symbolic encoding of a deterministic network product using BDDs. In the remainder of this section, refinements of simple automata encodings are used to encode both component networks and proposition automata.

5.4.3 Primitive Component Encoding

Let N be a component network. Assign to each primitive component $C \in N$ a unique set $\mathsf{vars}(C)$ of boolean valued state-encoding variables. Define $\mathsf{enc}_C \in C.S \rightarrow \mathsf{bdd}(\mathsf{vars}(C))$ to be a valid encoding function for $C.S$. Define $\mathsf{vars}'(C) = \{v' \mid v \in \mathsf{vars}(C)\}$. The complete encoding $\mathcal{C}$ of a primitive component $C \in N$ is a triple $\langle \mathcal{C}.\mathcal{S}, \mathcal{C}.\mathcal{I}, \mathcal{C}.\mathcal{T}\rangle$ as follows:

- $\mathsf{wexp}(w \in \mathsf{wires}(N)) \stackrel{\text{def}}{=} \mathsf{bddset}(\{x \in C.S \mid C.\lambda_w(x) = 1\}, \mathsf{enc}_C)$ where $C = \mathsf{parent}(w)$ is a BDD identifying the set of all value assignments to the variables in $\mathsf{vars}(C)$ for which w is at level 1.

- $\mathsf{wexp}'(w' \in \mathsf{wires}'(N)) \overset{\text{def}}{=} \mathsf{wexp}(w)[\mathsf{prime}]$ is a BDD identical to $\mathsf{wexp}(w)$ except that each variable $v \in \mathsf{vars}(C)$ is replaced by $v' \in \mathsf{vars}'(C)$.

- $\mathsf{invars}(C) \overset{\text{def}}{=} \bigcup_{w \in C.ins}(\mathsf{vars}(\mathsf{parent}(w)))$ denotes the set of state-encoding variables necessary to determine a value for every input wire $w \in C.ins$.

- $\mathsf{lexp}_C(l \subseteq C.ins)) \overset{\text{def}}{=} \bigwedge_{w \in l}(\mathsf{wexp}(w))$ is a BDD identifying all value assignments to the variables in $\mathsf{invars}(C)$ for which only inputs that are in l are at level 1.

- $\mathcal{C}.\mathcal{S} \overset{\text{def}}{=} \mathsf{bddset}(C.S, \mathsf{enc}_C)$ is a BDD over $\mathsf{vars}(C)$ encoding the set of states in C.

- $\mathcal{C}.\mathcal{I} \overset{\text{def}}{=} \bigvee_{(l,s) \in C.I}(\mathsf{lexp}_C(l) \wedge \mathsf{enc}_C(s))$ is a BDD over $\mathsf{vars}(C) \cup \mathsf{invars}(C)$ encoding the set of possible initial configurations $C.I$ of C.

- $\mathcal{C}.\mathcal{T} \overset{\text{def}}{=} \bigvee_{(s_1,l,s_2) \in c.T}(\mathsf{enc}_C(s_1) \wedge \mathsf{lexp}_c(l) \wedge (\mathsf{enc}_C(s_2)[\mathsf{prime}]))$ is a BDD over $\mathsf{vars}(C) \cup \mathsf{invars}(C) \cup \mathsf{vars}'(c)$ encoding the transition relation $C.T$ of C.

5.4.4 Component Network Encoding

Let N be a component network. Define $\mathcal{N} \overset{\text{def}}{=} \{\mathcal{C} \mid C \in N\}$ to be the BDD encoding of N. Let $\mathsf{vars}(N) \overset{\text{def}}{=} \bigcup_{C \in N}(\mathsf{vars}(C))$ and $\mathsf{vars}'(N) \overset{\text{def}}{=} \bigcup_{C \in N}(\mathsf{vars}'(C))$. Define $\mathsf{states}(\mathcal{N})$, $\mathsf{init}(\mathcal{N})$, and $\mathsf{trans}(\mathcal{N})$ as follows:

- $\mathsf{states}(\mathcal{N}) \overset{\text{def}}{=} \bigwedge_{C \in N}(\mathcal{C}.\mathcal{S})$ is a BDD over $\mathsf{vars}(\mathcal{N})$ encoding the set $\mathsf{states}(N)$ of states in N.

- $\mathsf{init}(\mathcal{N}) \overset{\text{def}}{=} \bigwedge_{C \in N}(\mathcal{C}.\mathcal{I})$ is a BDD over $\mathsf{vars}(\mathcal{N})$ encoding the set $\mathsf{init}(N)$ of initial states in N.

- $\mathsf{trans}(\mathcal{N}) \overset{\text{def}}{=} \bigwedge_{C \in N}(\mathcal{C}.\mathcal{T})$ is a BDD over $\mathsf{vars}(\mathcal{N}) \cup \mathsf{vars}'(\mathcal{N})$ encoding the transition relation $\mathsf{trans}(N)$ of N.

5.4.5 Network Proposition Encoding

Let N be a component network, and let p be a network proposition for N. Define the BDD encoding of p to be the boolean expression $p[\mathsf{wexp}][\mathsf{wexp}']$ resulting after each occurrence of $w \in \mathsf{wires}(N)$ in p has been expanded to $\mathsf{wexp}(w)$ and each occurrence of $w' \in \mathsf{wires}'(N)$ in p has been expanded to $\mathsf{wexp}'(w')$.

5.4.6 Proposition Automaton Encoding

Let M be a proposition automaton over component network N. Assign to M a unique set $\mathsf{vars}(M_i)$ of boolean-valued state encoding variables. Define $\mathsf{enc}_M \in M.\mathsf{states} \to \mathsf{bdd}(\mathsf{vars}(M))$ to be a valid encoding function for $\mathsf{vars}(M)$. Define $\mathsf{vars}'(M) \overset{\text{def}}{=} \{v' \mid v \in \mathsf{vars}(M)\}$. The complete BDD encoding $\mathcal{M} = \langle \mathcal{M}.\mathsf{states}, \mathcal{M}.\mathsf{init}, \mathcal{M}.\mathsf{acc}, \mathcal{M}.\mathsf{trans}\rangle$ of M is defined as follows:

- $\mathcal{M}.\mathsf{states} \overset{\text{def}}{=} \mathsf{bddset}(M.\mathsf{states}, \mathsf{enc}_M)$ is a BDD over $\mathsf{vars}(M)$ encoding the set $M.\mathsf{states}$ of states in M.

- $\mathcal{M}.\mathsf{init} \overset{\text{def}}{=} \mathsf{bddset}(M.\mathsf{init}, \mathsf{enc}_M)$ is a BDD over $\mathsf{vars}(M)$ encoding the set $M.\mathsf{init}$ of initial states in M.

- $\mathcal{M}.\mathsf{acc} \overset{\text{def}}{=} \mathsf{bddset}(M.\mathsf{acc}, \mathsf{enc}_M)$ is a BDD over $\mathsf{vars}(M)$ encoding the set $M.\mathsf{acc}$ of accepting states in M.

- $\mathcal{M}.\mathsf{trans} \overset{\text{def}}{=} \bigvee_{(s_1,p,s_2)\in M.\mathsf{trans}}(\mathsf{enc}_M(s_1) \wedge p[\mathsf{wexp}][\mathsf{wexp}'] \wedge (\mathsf{enc}_M(s_2)[\mathsf{prime}]))$ is a BDD over $\mathsf{vars}(M)\ \cup\ (\mathsf{vars}(N) \cup\ \mathsf{vars}'(N))\ \cup\ \mathsf{vars}'(M)$ encoding the transition relation of M.

If $P = \{M_1, \ldots, M_n\}$ is a set of proposition automata over N then define the set $\mathcal{P} \overset{\text{def}}{=} \{\mathcal{M}_1, \ldots, \mathcal{M}_n\}$ of encoded proposition automata over $\mathcal{N}$ as follows:

- $\mathsf{vars}(P) \overset{\text{def}}{=} \mathsf{vars}(M_1) \cup \cdots \cup \mathsf{vars}(M_n)$ is the set of variables over which the encoded M_i are defined.

- $\mathsf{vars}'(P) \overset{\text{def}}{=} \mathsf{vars}'(M_1) \cup \cdots \cup \mathsf{vars}'(M_n)$.

- $\mathsf{states}(\mathcal{P}) \overset{\text{def}}{=} \bigwedge_{M\in P}(\mathcal{M}.\mathsf{states})$ is a BDD over $\mathsf{vars}(P)$ encoding the set $M.\mathsf{states}$ of states for each $M \in P$.

- $\mathsf{init}(\mathcal{P}) \overset{\text{def}}{=} \bigwedge_{M\in P}(\mathcal{M}.\mathsf{init})$ is a BDD over $\mathsf{vars}(P)$ encoding the set $M.\mathsf{init}$ of initial states for each $M \in P$.

- $\mathsf{trans}(\mathcal{P}) \overset{\text{def}}{=} \bigwedge_{M\in P}(\mathcal{M}.\mathsf{trans})$ is a BDD over $\mathsf{vars}(P) \cup \mathsf{vars}'(\mathcal{P})$ encoding the transition relation $M.\mathsf{trans}$ of each $M \in P$.

5.4.7 Deterministic Network Product Encoding

Let N be a component network and $P = \{M_1, \dots, M_n\}$ be a set of proposition automata over N. Require that each $M_i \in P$ be both complete and deterministic. Define the deterministic network product encoding $\mathcal{N}\cdot\mathcal{P}$ of the deterministic network product $N\cdot P$ as follows:

- $\mathsf{vars}(N\cdot P) \overset{\text{def}}{=} \mathsf{vars}(N) \cup \mathsf{vars}(P)$ is the set of BDD variables over which the states of $\mathcal{N}\cdot\mathcal{P}$ are defined.
- $\mathsf{vars}'(N\cdot P) \overset{\text{def}}{=} \mathsf{vars}'(N) \cup \mathsf{vars}'(P)$.
- $\mathsf{states}(\mathcal{N}\cdot\mathcal{P}) \overset{\text{def}}{=} \mathsf{states}(\mathcal{N}) \wedge \mathsf{states}(\mathcal{P})$ is the set of states in $\mathcal{N}\cdot\mathcal{P}$.
- $\mathsf{init}(\mathcal{N}\cdot\mathcal{P}) \overset{\text{def}}{=} \mathsf{init}(\mathcal{N}) \wedge \mathsf{init}(\mathcal{P})$ is a BDD over $\mathsf{vars}(N\cdot P)$ denoting the set of possible initial states for $\mathcal{N}\cdot\mathcal{P}$.
- $\mathsf{trans}(\mathcal{N}\cdot\mathcal{P}) \overset{\text{def}}{=} \mathsf{trans}(\mathcal{N}) \wedge \mathsf{trans}(\mathcal{P})$ is a BDD over $\mathsf{vars}(N\cdot P) \cup \mathsf{vars}'(N\cdot P)$ denoting a set of possible transitions for $\mathcal{N}\cdot\mathcal{P}$.

Each value assignment $l \in [\![\mathsf{states}(\mathcal{N}\cdot\mathcal{P})]\!]_{\mathsf{vars}(N\cdot P)}$ denotes the concatenation of a unique state $l \cap \mathsf{vars}(N) \in [\![\mathsf{states}(N)]\!]_{\mathsf{vars}(N)}$ and a unique state $l \cap \mathsf{vars}(M_i) \in [\![\mathcal{M}_i.\mathsf{states}]\!]_{\mathsf{vars}(M_i)}$ for each $M_i \in P$. A *path* in $\mathcal{N}\cdot\mathcal{P}$ is any sequence σ over $[\![\mathsf{states}(\mathcal{N}\cdot\mathcal{P})]\!]_{\mathsf{vars}(N\cdot P)}$ such that:

- If $|\sigma| > 0$ then $\sigma(0) \in [\![\mathsf{init}(\mathcal{N}\cdot\mathcal{P})]\!]_{\mathsf{vars}(N\cdot P)}$.
- If $|\sigma| > 1$ then $\forall i \in \{0 \dots |\sigma| - 2\}.\ \sigma(i) \cup \mathsf{Prime}(\sigma(i+1)) \in [\![\mathsf{trans}(\mathcal{N}\cdot\mathcal{P})]\!]_{\mathsf{vars}(N\cdot P)}$.

If σ is a path in $\mathcal{N}\cdot\mathcal{P}$ then σ identifies a unique network execution $\sigma\Downarrow_N$ such that $\forall i \in \{0 \dots |\sigma| - 1\}.\ \sigma(i) \cap \mathsf{vars}(N) \in [\![\mathsf{enc}_N(\sigma\Downarrow_N)]\!]_{\mathsf{vars}(N)}$. Furthermore, if σ is a path in $\mathcal{N}\cdot\mathcal{P}$ then for every $M_i \in P$, σ identifies the unique supporting path $\sigma\Downarrow_{M_i}$ for $(\sigma\Downarrow_N)^2$ in M_i such that $\forall j \in \{0 \dots |\sigma| - 1\}.\ \sigma(j) \cap \mathsf{vars}(M_i) \in [\![\mathsf{enc}_{M_i}(\sigma\Downarrow_{M_i}(j))]\!]_{\mathsf{vars}(M_i)}$.

Define the language of $\mathcal{N}\cdot\mathcal{P}$ as $L(\mathcal{N}\cdot\mathcal{P}) \overset{\text{def}}{=} \{\sigma \mid \sigma \text{ is a path in } N\cdot P\}$ and define $L\Downarrow_N(\mathcal{N}\cdot\mathcal{P}) \overset{\text{def}}{=} \{\sigma\Downarrow_N \mid \sigma \in L(\mathcal{N}\cdot\mathcal{P})\}$. Note that $L\Downarrow_N(\mathcal{N}\cdot\mathcal{P}) = L(N)$ is always true since every $M_i \in P$ is complete.

Construction of $\mathcal{N}\cdot\mathcal{P}$ binds every $M_i \in P$ to N such that transitions of N denote input symbols to M and such that paths in $\mathcal{N}\cdot\mathcal{P}$ denote both executions of N and associated supporting paths for each $M_i \in P$. The operator $\sigma\Downarrow_X$ denotes the extraction and decoding of a particular sub-path X from some path σ in $\mathcal{N}\cdot\mathcal{P}$: If $\sigma \in L(\mathcal{N}\cdot\mathcal{P})$ and $M_i \in P$ then $\sigma\Downarrow_N(j) \in \mathsf{states}(N)$ decodes the state of component network N in $\sigma(j)$ and $\sigma\Downarrow_{M_i}(j) \in M_i.\mathsf{states}$ decodes the state of proposition automaton M_i in $\sigma(j)$.

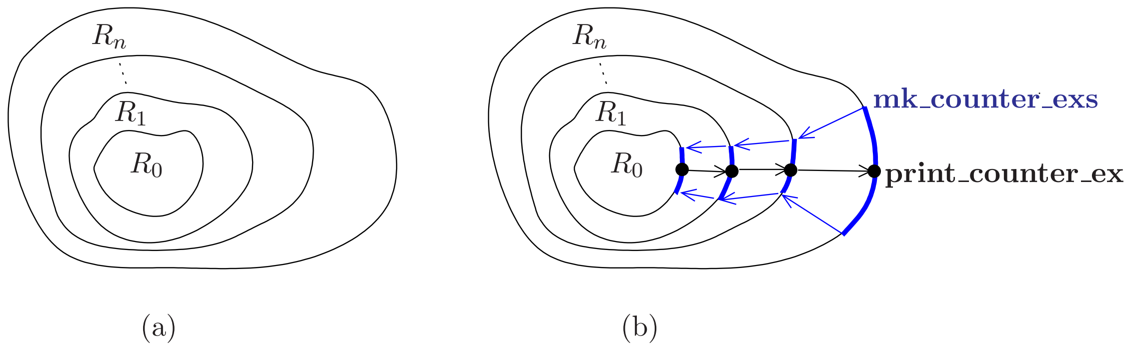


Figure 5.2: Breadth-first reachability analysis.

5.5 Symbolic Verification using Binary Decision Diagrams

Let $S = \{E_1, \ldots, E_n\}$ be a set of proposition-expressions over component network N, and assign to each $E_i \in S$ a sense and a type as discussed in Section 5.2. For each $E_i \in S$ compute a complete and deterministic proposition automaton M_i such that $L(M_i) = L(E_i)$. Let $P = \{M_1, \ldots, M_n\}$ and require that each $M_i \in P$ contain a unique dead state err. Assign to each $M_i \in P$ the same sense and type as E. A procedure **pcheck** for determining conformance of N to S based on the network product encoding $\mathcal{N}\cdot\mathcal{P}$ is shown in Algorithm 5.1. For convenience it is assumed that $\mathsf{vars}(N\cdot P), \mathsf{vars}'(N\cdot P), \mathsf{trans}(\mathcal{N}\cdot\mathcal{P})$, and $\mathsf{init}(\mathcal{N}\cdot\mathcal{P})$ are all available as global variables.

The operation of **pcheck** equates to a symbolic breadth-first search of the set $\{\mathsf{last}(\sigma) \mid \sigma \in L(\mathcal{N}\cdot\mathcal{P})\}$ of states reachable in $\mathcal{N}\cdot\mathcal{P}$. Each R_n encodes a set containing those states reachable within n transitions of an initial component network state, see Figure 5.2(a). R_{n+1} is computed by extending R_n with the image-set of those transitions $R_i \wedge \mathsf{trans}(\mathcal{N}\cdot\mathcal{P})$ possible from each state $s \in [\![R_n]\!]_{\mathsf{vars}(N\cdot P)}$. R_0 is assigned the set of initial states in $\mathcal{N}\cdot\mathcal{P}$ and the repeat loop iterates until $R_n \equiv R_{n-1}$, at which point all reachable states have been visited. Procedure **image_cut**, see Algorithm 5.2, ensures that $\mathcal{N}\cdot\mathcal{P}$ only exhibits network executions that do not violate any of the constraints imposed by each $M_i \in P$ of type cut. Procedure **image_verify**, see Algorithm 5.3, catches any network executions possible in $\mathcal{N}\cdot\mathcal{P}$ that violate a specification imposed by some $M_i \in P$ of type verify. The correctness of **image_cut** and the correctness of **image_verify** is based on the assertion from Section 5.3 that if each $M_i \in P$ is both complete and deterministic then $|\mathsf{slast}(M_i, \sigma)| = 1$, and hence reachable component network states are in one-to-one correspondence with supporting proposition automata states.

```
FUNCTION pcheck
  $n := 0$
  $R := R_0 := \mathsf{init}(\mathcal{N}\cdot\mathcal{P})$
  REPEAT
    $Im := (\exists \mathsf{vars}(N\cdot P).\ R_n \wedge \mathsf{trans}(\mathcal{N}\cdot\mathcal{P}))[\mathsf{unprime}]$
    image_cut($Im$)
    $err\_states$ := image_verify($Im$)
    IF $err\_states \not\equiv \mathsf{F}$ THEN
      print_counter_ex($err\_states,\ R_n$)
    END IF
    $n := n + 1$
    $R_n := Im \vee R_{n-1}$
  UNTIL $R_n \equiv R_{n-1}$
END FUNCTION
```

Algorithm 5.1: Symbolic procedure for determining conformance of N to P.

```
FUNCTION image_cut  (REF $Im$ : $\mathsf{bdd}(\mathsf{vars}(N\cdot P))$)
  FOR ALL $M_i \in P$ DO
    IF cut never($M_i$) THEN
      $Im := Im \wedge \neg\mathcal{M}_i.\mathsf{acc}$
    END IF
    IF cut always($M_i$) THEN
      $Im := Im \wedge \neg\mathsf{enc}_{M_i}(\mathsf{err})$
    END IF
  END FOR
END FUNCTION
```

Algorithm 5.2: Symbolic removal of erroneous reachable states in $\mathcal{N}\cdot\mathcal{P}$.

```
FUNCTION image_verify  ($Im$ : $\mathsf{bdd}(\mathsf{vars}(N\cdot P))$) : $\mathsf{bdd}(\mathsf{vars}(N\cdot P))$)
  FOR ALL $M_i \in P$ DO
    $err\_states := \mathsf{F}$
    IF verify never($M_i$) THEN
      $err\_states := Im \wedge \mathcal{M}_i.\mathsf{acc}$
    END IF
    IF verify always($M_i$) THEN
      $err\_states := Im \wedge \mathsf{enc}_{M_i}(\mathsf{err})$
    END IF
    IF $err\_states \not\equiv \mathsf{F}$ THEN
      Violation of specification $M$.
      RETURN $err\_states$
    END IF
  END FOR
  RETURN $\mathsf{F}$
END FUNCTION
```

Algorithm 5.3: Symbolic detection of erroneous network executions in $\mathcal{N}\cdot\mathcal{P}$.

If a specification verify $sense(M_i)$ is violated by N then a counter example to the specification can be generated by procedure **print_counter_ex**(err_states, R_n), see Algorithm 5.4. Execution of **print_counter_ex** begins with a call to **mk_counter_exs** which performs a backwards breadth-first search starting from erroneous states in *err_states*. The sequence σ constructed by **mk_counter_exs** identifies the set of all erroneous network executions leading to states in *err_states*. Once σ is computed, procedure **print_counter_ex** walks forwards over σ picking an example minimum length erroneous network execution, see Figure 5.2(b).

```
FUNCTION print_counter_ex ($Err$ : $\mathsf{bdd}(\mathsf{vars}(N\cdot P))$, $R$ : $\mathsf{bdd}(\mathsf{vars}(N\cdot P))$)
  $\sigma := \langle Err \rangle$
  mk_counter_exs($R$, $\sigma$)
  FOR $i := 0$ to $|\sigma| - 2$ step 1 DO
5:  pick any state $s \in [\![\sigma(i)]\!]_{\mathsf{bdd}(\mathsf{vars}(N\cdot P))}$
    print $s$
    LET $B$ be a BDD such that $[\![B]\!]_{\mathsf{vars}(N\cdot P)} = \{s\}$
8:  $Im := (\exists \mathsf{vars}(N\cdot P).\ B \wedge \mathsf{trans}(\mathcal{N}\cdot\mathcal{P}))[\mathsf{unprime}]$
    $\sigma(i+1) := \sigma(i+1) \wedge Im$
  END FOR
  pick any state $s \in [\![\mathsf{last}(\sigma)]\!]_{\mathsf{bdd}(\mathsf{vars}(N\cdot P))}$
12: print $s$
END FUNCTION
FUNCTION mk_counter_exs ($R$ : $\mathsf{bdd}(\mathsf{vars}(N\cdot P))$, $\sigma$ : $\mathsf{bdd}(\mathsf{vars}(N\cdot P))$ sequence)
  $PreIm := \exists \mathsf{vars}'(N\cdot P).\ \sigma(0)[\mathsf{prime}] \wedge \mathsf{trans}(\mathcal{N}\cdot\mathcal{P}) \wedge R$
  image_cut($PreIm$)
  IF $PreIm \wedge \mathsf{init}(\mathcal{N}\cdot\mathcal{P}) \equiv \mathsf{F}$ THEN
    $\sigma := \langle PreIm \rangle \sigma$
    mk_counter_exs($R \wedge \neg PreIm, \sigma$))
  ELSE
    $\sigma := \langle PreIm \wedge \mathsf{init}(\mathcal{N}\cdot\mathcal{P}) \rangle \sigma$
  END IF
END FUNCTION
```

Algorithm 5.4: Counter-example generation.

5.6 BDD-based Symbolic Searching Optimisations

The optimisation of symbolic breadth-first searching using BDDs has been extensively documented in the literature [50, 100]. Although a detailed analysis of these optimisations is beyond the scope of this thesis, this section aims to overview those optimisations that are most relevant to the proposition-oriented verification procedure **pcheck**.

5.6.1 Frontier-Set Reduction

Each iteration of a symbolic breadth first search can be equated to the computation of $R_{n+1} := R_n \vee (\exists V.\ R_n \wedge T)[\mathsf{unprime}]$, where T is a transition relation and V is a set of current-state variables. The purpose of this computation is to extend an existing set, R_n of reachable states with the image-set $(\exists V.\ R_n \wedge T)[\mathsf{unprime}]$ of states reachable from R_n using transitions in T.

At each iteration the difference, $R_n - R_{n-1}$, is referred to as a *frontier-set*, and denotes those states reachable in no less or more than n transitions of an initial state. Using an inductive argument it is possible to show that $R_n \vee (\exists V.\ R_n \wedge T)[\mathsf{unprime}] = R_n \vee (\exists V.\ (R_n - R_{n-1}) \wedge T)[\mathsf{unprime}]$, and therefore that R_{n+1} need only be computed using the image of the frontier-set and not the image of R_n. This inductive argument generalises to any set F_i such that $R_n - R_{n-1} \subseteq F_i \subseteq R_n$. *Frontier-set reduction* is a technique that uses a special BDD operator **reduce** [28] to attempt to pick the F_i with the smallest BDD encoding. Execution of **reduce** at each iteration takes time, however the reduction in size of the frontier-set BDD can significantly reduce the time taken to perform image computations.

5.6.2 Relational Product

The core operation behind a symbolic breadth-first search is the image-set computation $\exists V.\ R_n \wedge T$. Image-set computation generalises to the expression $\exists X.\ A \wedge B$, known as a *relational product* [50]. Relational products can be efficiently computed using a single atomic BDD operation which conjoins the binary trees for A and B simultaneously with quantifying out those variables in X. The use of an atomic relational product operator has been shown to dramatically reduce symbolic breadth-first search times, and can be directly applied to Line 5 of procedure **pcheck**, see Algorithm 5.1.

5.6.3 BDD Variable Ordering

Practical BDD function libraries use a fixed BDD variable ordering and share pointers to equivalent subtrees. These libraries are referred to as Reduced-Ordered-BDD (ROBDD) libraries and permit constant-time BDD negation and equality computation. Although the semantics of BDDs are independent of any chosen variable ordering, the size of a BDD and the performance of boolean operations on it is dependent on the chosen variable ordering. Heuristics for improving ROBDD variable orderings can, in certain cases, significantly reduce breadth-first search times. However, since re-ordering of ROBDD variables requires reconstruction of all BDDs in existence, these reduction in search times must be tempered by the time penalty incurred in having performed the re-orderings in the first place [84].

5.6.4 Conjunctively Partitioned Transition Relation

A network product $N{\cdot}P$ is a form of product automaton formed from a set of primitive components, $N = \{C_1, \ldots, C_n\}$, and a set of proposition automata $P = \{M_1, \ldots M_m\}$. When encoded symbolically using BDDs, a network product transition relation $\mathsf{trans}(\mathcal{N}\cdot\mathcal{P})$ equates to the conjunction of a set of smaller transition relations:

$$\mathsf{trans}(\mathcal{N}\cdot\mathcal{P}) \equiv (\mathcal{C}_1.\mathcal{T}) \wedge \ \ldots \ \wedge \ (\mathcal{C}_n.\mathcal{T}) \wedge (\mathcal{M}_1.\mathsf{trans}) \ \wedge \ \ldots (\mathcal{M}_m.\mathsf{trans})$$

The explicit computation of $\mathsf{trans}(\mathcal{N}\cdot\mathcal{P})$ is referred to as a *monolithic* transition relation and can limit the performance of symbolic breadth-first searching procedures. However, the explicit computation of $\mathsf{trans}(\mathcal{N}\cdot\mathcal{P})$ is not necessary, and it is also possible to compute $\mathit{reached\_states} \wedge \mathsf{trans}(\mathcal{N}\cdot\mathcal{P})$ iteratively using $n + m$ conjunctions of the individual component transition relations that comprise $\mathsf{trans}(\mathcal{N}\cdot\mathcal{P})$. The implicit repesentation of $\mathsf{trans}(\mathcal{N}\cdot\mathcal{P})$ as the conjunction of component transition relations is referred to as a *conjunctively partitioned* transition relation [50].

5.6.5 Early Quantification

In general, a conjunctively partitioned transition relation $T = \{T_1, \ldots T_n\}$ encodes a product $T_1 \wedge \ldots \wedge T_n$, where each T_i is an individual component BDD over a set $V \cup V'$ of boolean variables such that $V \cap V' = \emptyset$. If R is a BDD over V encoding a set of reachable states then the image computation $\exists V.\ R \wedge (\bigwedge_{T_i \in T} T_i)$ can be computed by considering each of the T_i in any order. *Early quantification* of a conjunctively partitioned transition relation atempts to distribute the existental quantifications for each $v \in V$ over the product $R \wedge \bigwedge_{T_i \in T} T_i$ [50]. Distribution of existential quantification in this way permits each $v \in V$ to be quantified out as soon as possible, but is dependant on the order in which each of the T_i are considered.

For any given ordering of the T_i it is possible to determine a maximal set V_i for each $T_i \in T$ such that

$$\exists V.\ R \wedge (\textstyle\bigwedge_{T_i \in T} T_i) \ \equiv \ \exists V_n.\ (\ldots.\ (\exists V_2.\ (\exists V_1.\ R \wedge T_1) \wedge T_2) \ldots)$$

Early quantification reduces each breadth-first search step of a conjunctively partitioned transition relation to a sequence of relation products, one for each component transition relation. A common heuristic for ordering the T_i is to first pick the T_i that depends on the smallest number of variables in V [50].

5.7 Protocol Proposition Automata

Proposition-expressions encapsulate the application of proposition-oriented behaviours to both a sequential notation, regular-expressions, and to a notation for concurrency, trace-expressions. The objective of this application is to provide a simple platform on which the benefits of proposition-oriented behaviours can be demonstrated in the context of asynchronous design. However, regular-expressions and trace-expressions are themselves simple notations on which significant further research has been based. In particular, process spaces [76], DI-algebra [49], and the XDI model [57] all extend event-oriented behaviours to encapsulate relationships between system and environment, and to express notions of liveness and progress.

Protocol automata are a simple extension to complete proposition automata in which a second dead state, rej, is introduced. rej relates closely to rejections in process spaces and to the miracle state in XDI models, and is motivated by a desire to enable a single proposition automaton to describe both a circuit specification and its associated environmental assumptions: Network executions supported by paths ending in state rej denote failures due to the environment, whereas network executions supported by paths ending in state err denote failures due to the circuit. Protocol automata enable a proposition-oriented trace-expression t to assert this contract between circuit and environment, provided those wires $w \in \Sigma_t$ that are circuit inputs can be identified: If $w \in \Sigma_t$ is a circuit input then network executions that are unsupported due to events on w denote failures due to the environment. If $w \in \Sigma_t$ is not a circuit input then network executions that are unsupported due to events on w denote failures due to the circuit.

Protocol automata differ from process spaces and XDI models in that the relationship between rej and err is not embraced explicitly in their semantics, and notions of liveneess between circuit and environment therefore cannot be evolved. A superior treatment of system-environemt contracts and liveness in the context of proposition-oriented trace-expressions is beyond the scope of this thesis.

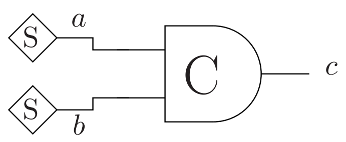


Figure 5.3: A simple C-element component network.

An example application for protocol automata is as follows: Let $S = (a^*; c^*) \parallel (b^*; c^*)$ denote a proposition-oriented trace-expression specification for the component network N, see Figure 5.3. Conformance of N to $\mathsf{verify}\ \mathsf{always}(S)$ is dependant upon events on both a and b interleaving events on c, which is not true since wires a and b may change value arbitarily. An ability to partition network executions using rej and err permits failures due to a, b and failures due to c to be distinguished, thus enabling S to assert instead that the behaviour of c be verified under the assumption that a, b are well behaved.

5.7.1 Protocol Completion

Let t be a proposition-oriented trace-expression over component network N and let $M = \mathbf{mk\_trexp}(t, \mathbf{sigma}(t))$ be a proposition automaton with $L(M) = L_{te}(t)$ that containes no dead states. Let $\mathsf{inputs}(t) \subseteq \Sigma_t$ denote those wires in N that are to be considered as inputs to t from the environment. The protocol completion $\mathbf{protocol}(M, \mathsf{inputs}(t))$ of M is a refinement of the completion $\mathbf{complete}(M)$ of M as shown in Algorithm 5.5. $\mathbf{protocol}(M, \mathsf{inputs}(t))$ differs from $\mathbf{complete}(M)$ in that the network proposition $\neg\mathsf{stable}(\mathsf{inputs}(t))$ is used to partition unsupported network executions according to the circuit-environment contract described above: If a network execution is unsupported due to a component network transition in which no input wire changes value, denoted by $\mathsf{stable}(\mathsf{inputs}(t))$, then that network execution must lead to state err, otherwise it leads to state rej.

Conformance of a component network N to a protocol automaton M can be determined by extending procedures **image_cut** and **image_verify** to include a new $\mathsf{protocol}$ type as shown in Algorithms 5.6 and 5.7. Since a notion of rej is not meaningful if the sense of a proposition automaton is never, protocol automaton need not be assigned a sense since that sense can only be always.

```
FUNCTION protocol (REF $M$ : $\mathsf{propaut}_N$, $i$ : $\mathsf{bexp}(\mathsf{dwires}(N))$)
  kill_dead($M$)
  FOR ALL $s \in M.\mathsf{states}$ DO
    addtrans($M$, $s$, $\neg\mathsf{exits}(p) \wedge \neg\mathsf{stable}(i)$, $\mathsf{rej}$)
    addtrans($M$, $s$, $\neg\mathsf{exits}(p) \wedge \mathsf{stable}(i)$, $\mathsf{err}$)
  END FOR
  addtrans($M$, $\mathsf{err}$, $\mathsf{T}$, $\mathsf{err}$)
  addtrans($M$, $\mathsf{rej}$, $\mathsf{T}$, $\mathsf{rej}$)
  $M.\mathsf{states} := M.\mathsf{states} \cup \{\mathsf{err}, \mathsf{rej}\}$
END FUNCTION
```

Algorithm 5.5: Completion of a proposition automaton.

```
FUNCTION image_cut (REF $Im$ : $\mathsf{bdd}(\mathsf{vars}(N \cdot P))$)
  FOR ALL $M_i \in P$ DO
    IF cut never($M_i$) THEN
      $Im := Im \wedge \neg\mathcal{M}_i.\mathsf{acc}$
    END IF
    IF cut always($M_i$) THEN
      $Im := Im \wedge \neg\mathsf{enc}_{M_i}(\mathsf{err})$
    END IF
    IF protocol $M_i$ THEN
      $Im := Im \wedge \neg\mathsf{enc}_{M_i}(\mathsf{rej})$
    END IF
  END FOR
END FUNCTION
```

Algorithm 5.6: Addition of protocol type to image_cut.

```
FUNCTION image_verify ($Im$ : $\mathsf{bdd}(\mathsf{vars}(N \cdot P))$) : $\mathsf{bdd}(\mathsf{vars}(N \cdot P))$)
  FOR ALL $M_i \in P$ DO
    $err\_states := \mathsf{F}$
    IF verify never($M_i$) THEN
      $err\_states := Im \wedge \mathcal{M}_i.\mathsf{acc}$
    END IF
    IF verify always($M_i$) or protocol $M_i$ THEN
      $err\_states := Im \wedge \mathsf{enc}_{M_i}(\mathsf{err})$
    END IF
    IF $err\_states \not\equiv \mathsf{F}$ THEN
      Violation of specification $M$.
      RETURN $err\_states$
    END IF
  END FOR
  RETURN $\mathsf{F}$
END FUNCTION
```

Algorithm 5.7: Addition of protocol type to image_verify.

5.8 Summary

The purpose of this chapter was to evolve a simple verification procedure for proposition automata over component networks. The operation of this procedure was shown to equate to reachability analysis on a special type of product automaton called a network product, and this demonstration was used to justify an implementation based on symbolic reachability analysis using BDDs. BDD-based optimisations were summarised, and a special type of proposition automata called protocol automata were introduced. The purpose of proposition automata was to permit the description of simple circuit-environment contracts using proposition-oriented trace-expressions.

Chapter 6
Veraci

6.1 Introduction

The purpose of this chapter is to demonstrate the application of proposition-oriented verification to asynchronous circuit design. This demonstration is facilitated through the use of a custom-built verification program, Veraci, implemented using the algorithms described in Chapters 4 and 5. Veraci takes as its input a circuit described in a subset of standard Verilog [99] that has been annotated with one or more *Veraci-fragments*. These Veraci-fragments can be used both to denote assumptions regarding the behaviour of a circuit and to describe specifications that must adhered to by that circuit.

The application of Veraci to asynchronous design is described in part by the syntax of Veraci itself, and in part through a number of small example circuits, each of which aims to demonstrate a different benefit of proposition-oriented behaviours over alternative level and event-oriented notations. The performance of Veraci is also discussed and compared to that of Versify [86], a Petri-net-oriented verification program.

In the sections that follow Section 6.2 describes the syntax and functionality of Veraci. Section 6.3 outlines how a combination of levels and events into network propositions can be used to facilitate bundled-data asynchronous design. Section 6.4 demonstrates how active propositions permit event abstraction in the context of a delay insensitive data-encoding scheme. Sections 6.5 and 6.6 describe how proposition-oriented verification can be applied to safety conditions and to causality without progress. Section 6.7 demonstrates the application of proposition-oriented behaviours to timing assumptions, and Section 6.8 documents an application of Veraci not due to the author. This application includes a demonstration of counter-example generation in Veraci. Section 6.9 concludes the chapter by investigating Veraci performance.

```
assign y = (a | b) & c;
```

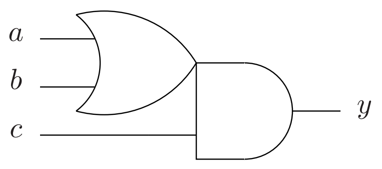
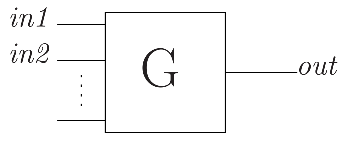
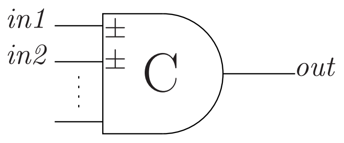


Figure 6.1: An assign statement per complex gate in Veraci.

```
 and G(out, in1, in2, ...)
nand G(out, in1, in2, ...)
  or G(out, in1, in2, ...)
 nor G(out, in1, in2, ...)
 xor G(out, in1, in2, ...)
xnor G(out, in1, in2, ...)
```



```
celem C(out, ±in1, ±in2, ...)
```



```
arbiter A(g1,r1, g2,r2, ...)
```

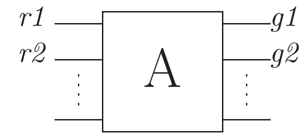


Figure 6.2: Gate instantiations accepted by Veraci.

6.2 Veraci Syntax

Veraci is a command-line driven verifier written in C^{++} [95]. The underlying verification engine on which Veraci is built is based on the procedures described in Chapter 5, and uses an off-the-shelf ROBDD function library written in C [56]. Veraci accepts as its input a circuit described in a subset of standard Verilog that contains only gate instantiations and continuous assignments.

Veraci uses the circuit described by its Verilog input to construct a component network as described in Section 3.4. Top-level inputs to the main module are connected to source components; continuous assignments are translated into complex gates, see Figure 6.1; and gate instantiations are translated as described in Figure 6.2. Veraci's treatment of gates `celem` and `arbiter` as primitive is non-standard to Verilog, but is accepted by Veraci for convenience in asynchronous design. Veraci also permits asymmetric C-elements to be described by prefixing any `celem` inputs with either a `+` or a `-` symbol.

$$
\begin{array}{rll}
\textit{veraci-fragment} ::= & \textit{type} \;\; \textit{sense} \;\; \textit{Pexp} & \text{(for a definition of } Pexp \text{ see section 4.5)} \\
\mid & \mathsf{protocol} \;\; \textit{Texp} & \text{(for a definition of } Texp \text{ see section 3.6.2)} \\
\textit{type} ::= & \mathsf{cut} & \\
\mid & \mathsf{verify} & \\
\textit{sense} ::= & \mathsf{always} & \\
\mid & \mathsf{never} &
\end{array}
$$

Figure 6.3: Veraci-fragment grammar.

6.2.1 Veraci-fragments

Each Verilog module input to Veraci may be augmented with proposition-oriented assertions and specifications, referred to as Veraci-fragments. A Veraci-fragment is distinguished from a circuit component by enclosing it in double-angled brackets, `<<`$\cdots$`>>`, and consists of a proposition-expression or a proposition-oriented trace-expression prefixed by one or more sense and type modifiers, see Figure 6.3. If module `m` contains a Veraci-fragment of the form `<< protocol` t `>>` then $\mathsf{inputs}(t)$ is defined as:

$$\mathsf{inputs}(t) \overset{\text{def}}{=} \{w \mid w \in \Sigma_t \text{ and } w \text{ is an input to } \mathtt{m}\}$$

In this sense each Verilog module, including the main module, considers its inputs to be controlled by the environment. Alternative circuit-environment strategies are possible, but the use of module hierarchy in this way is both convenient and simple.

6.2.2 Network Propositions

A network proposition is a boolean expression over current-next state wire values. Veraci permits any Verilog wire name w to be used in a network proposition either in primed, w', or un-primed, w, form: w denotes the value of wire w in the current component network state and w' denotes the value of wire w in the next component network state. The syntax for network propositions in Veraci is outlined in Table 6.1, and mimics those boolean operators used in standard Verilog.

The syntactic use of '`&`' rather than '$\wedge$', and '`|`' rather than '$\vee$' in network propositions leads to ambiguity with respect to the proposition-expression constructs '&' and '|', used to denote proposition-expression product and alternation respectively. This ambiguity is resolved syntactically in Veraci, where proposition-expression product is denoted instead as '`&.`', and proposition-expression alternation is denoted instead as '`|.`'.

Construct	Meaning
`p` ::= `1`	T
| `0`	F
| w	current-state wire name
| w`'`	next-state wire name
| w`*`	$w \neq w'$
| w`+`	$w^* \wedge \neg w$
| w`-`	$w^* \wedge w$
| `(p)'`	$p[\mathsf{prime}]$
| `(p)*`	$p \neq (p[\mathsf{prime}])$
| `(p)+`	$p^* \wedge \neg p$
| `(p)-`	$p^* \wedge p$
| `~p`	$\neg p$
| `p`$_1$ `&` `p`$_2$	$p_1 \wedge p_2$
| `p`$_1$ `|` `p`$_2$	$p_1 \vee p_2$
| `p`$_1$ `^` `p`$_2$	$p_1 \oplus p_2$
| `p`$_1$ `=>` `p`$_2$	$p_1 \Rightarrow p_2$
| `p`$_1$ `=` `p`$_2$	$p_1 = p_2$

Table 6.1: Network propositions in Veraci.

6.2.3 Excitation Propositions

If N is a component network and $w \in \mathsf{wires}(N)$ then define

$$\mathsf{enabled}(w) \overset{\text{def}}{=} \{s_1 \in \mathsf{states}(N) \mid \exists s_2.\ (s_1, s_2) \in \mathsf{trans}(N) \text{ and } \mathsf{val}(w, s_1) \neq \mathsf{val}(w, s_2)\}$$

to denote the set of all component network states in which w is enabled to change value. Define $\mathsf{excited}(w)$ to be any network proposition over N such that $\mathsf{labels}(\mathsf{excited}(w)) = \{(s_1, s_2) \in \mathsf{alltrans}(N) \mid s_1 \in \mathsf{enabled}(w)\}$. The purpose of $\mathsf{excited}(w)$ is to label any component network transition where w is enabled to change value in the current state. For example if w is the output of a two-input AND-gate with inputs a, b, then $\mathsf{excited}(w) \equiv (a \wedge b \wedge \neg w) \vee (\neg(a \wedge b) \wedge w)$. $\mathsf{excited}$ can be applied to network propositions in Veraci using the `excited` construct in either of two ways as follows:

1. **Qualified**: `excited(w1,...,wn)` $\overset{\text{def}}{=} \mathsf{excited}(\mathtt{w1}) \vee \cdots \vee \mathsf{excited}(\mathtt{wn})$.
2. **Unqualified**: `excited` $\overset{\text{def}}{=} \bigvee_{w \in W}(\mathsf{excited}(w))$ where W is the set of all wire names in the current Verilog module that are not input wires.

For example, `(excited(x) => ~a)` and `(excited & b)` are both valid network propositions in Veraci. The unqualified use of `excited` is intended as shorthand for excitation of the whole current module. Input wires are excluded from W in unqualified uses of `excited` since input excitation is *external* to the current module.

Function-lets for reasoning with dual-rail protocols

```
 dr_plus(a,b)  =  ((a+ & b=0 & b'=0) ^ (b+ & a=0 & a'=0))
dr_minus(a,b)  =  ((a- & b=0 & b'=0) ^ (b- & a=0 & a'=0))
dr_event(a,b)  =  dr_plus(a,b) | dr_minus(a,b)
dr_clear(a,b)  =  ~a & ~b
 dr_data(a,b)  =  a ^ b
dr_error(a,b)  =  a & b
```

Function-lets for reasoning with inverted dual-rail protocols

```
 dri_plus(a,b)  =  ((a- & b=1 & b'=1) ^ (b- & a=1 & a'=1))
dri_minus(a,b)  =  ((a+ & b=1 & b'=1) ^ (b+ & a=1 & a'=1))
dri_event(a,b)  =  dri_plus(a,b) | dri_minus(a,b)
dri_clear(a,b)  =  a & b
 dri_data(a,b)  =  a ^ b
dri_error(a,b)  =  ~a & ~b
```

Table 6.2: Function-let declarations used in this thesis.

6.2.4 Semi-modularity Propositions

If N is a component network and $w \in \mathsf{wires}(N)$ is a wire in N then w is said to be semi-modular if whenever w is enabled to change value it is never *disabled* unless it also changes value. If every wire in N is semi-modular then N is said to be semi-modular. If $w \in \mathsf{wires}(N)$ then the semi-modularity of w can be expressed as the network proposition $\mathsf{semi}(w) \overset{\text{def}}{=} \mathsf{excited}(w) \wedge \neg(\mathsf{excited}(w)[\mathsf{prime}]) \Rightarrow w^*$. $\mathsf{semi}(w)$ asserts that if w is excited in the current component network state but not in the next component network state then w must change its value. In Veraci it is possible to use $\mathsf{semi}(w)$ directly in network propositions using the construct '`semi`' in either qualified or unqualified form:

1. **Qualified**: `semi(w1,...,wn)` $\overset{\text{def}}{=} \mathsf{semi}(\mathtt{w1}) \wedge \cdots \wedge \mathsf{semi}(\mathtt{wn})$.

2. **Unqualified**: `semi` $\overset{\text{def}}{=} \bigwedge_{w \in W}(\mathsf{semi}(w))$ where W is the set of all wire names in the current Verilog module that are not input wire names.

The application of `excited` and `semi` to asynchronous design is discussed further in Sections 6.5 and 6.7.

6.2.5 Function-Let Declarations

The network proposition constructs defined in Table 6.1 can be extended to include new constructs by providing Veraci with any number of *function-let* declarations of the form:

`let` $\mathit{fn\_name}(\mathit{arg1},\ \mathit{arg2},\ \ldots)\ = p$

Each function-let definition defines a new function *fn_name* which can be used as shorthand for the network proposition p. For example, the expression `let plus_low(a,b) = a+ & b=0` can be used to declare the construct `plus_low`(a, b) which labels any component network transition where wire a is rising but wire b is low. Function-let declarations can be packaged into a library for inclusion with any Veraci input file. The function-let declarations used in this thesis are outline in Table 6.2.

6.2.6 Delay Models

Veraci requires that each Verilog module definition be identified as either `SI` or `DI`, see Figure 6.4(a). A module that is declared as SI is verified under a speed independent delay model in which wire delays are assumed to be zero and gate delays are assumed to be arbitrary but finite. A module that is declared as DI is verified under a delay insensitive delay model in which both gate and wire delay are assumed to be arbitrary but finite.

Veraci does not permit a delay insensitive module to be instantiated inside of a speed independent module. If a speed independent module is instantiated inside of a delay insensitive module then Veraci creates a foam-rubber-wrapper [68] by placing wire delays on each input to the speed independent module, see Figure 6.4(b).

6.2.7 Circuit Initialisation

The set of possible initial states of a component network is determined by the valid initial configurations for each of its individual primitive components. The primitive components described in Section 3.4 adopt a *quiescent* initialisation model in that valid initial configurations are ones where no output is excited to change its value. The exceptions to this rule are the arbiter component, which always initialises with all outputs at level 0, and the source component which is always excited and can initialise in any state.

Veraci permits a designer to place further constraints on the initial configurations of a component network. This is done through the use of special "`initial` p" fragments where p is any boolean expression over current-state wire names only. If N is the component network to be verified by Veraci then the application of `initial` p to N causes Veraci to remove all component network states not in $\bigcup_{x\in[\![p]\!]_{\mathsf{wires}(N)}}(\mathsf{smap}(p))$ from the initial states, $\mathsf{init}(N)$, of N.

(a)

```
module SI latch(q, qb, s, r);
   ⋮
endmodule

module DI latch(q, qb, s, r);
   ⋮
endmodule
```

(b)

```
module SI latch(q, qb, s, r);
   ⋮
endmodule

module DI latch2(q, qb, s, r);
   ⋮
   latch L(q, qb, s, r);
endmodule
```

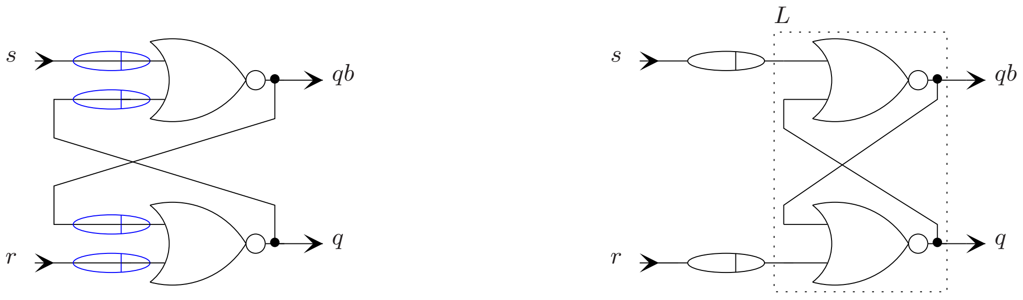


Figure 6.4: Different delay models in Veraci.

6.2.8 Counter-example Generation

Veraci uses its input file to determine both a component network and a set of proposition automata. The component network is determined from Verilog constructs, and the proposition automata are determined from Veraci fragments. Veraci uses the component network and proposition automata combined to compute a network product from which conformance is determined by a symbolic BDD-based breadth first search as described in Section 5.5. If the component network does not conform to the Veraci fragments then Veraci computes a minimal length counter-example which is displayed to the user as an erroneous sequence of events from some possible initial circuit state.

6.3 Combining Levels and Events

The foundation of this thesis was a desire to unify levels and events under a common behavioural paradigm. The purpose of this section is to demonstrate a simple benefit of level-event unification by example. The example chosen is the specification of a four-phase bundled-data multiplexer circuit, **bdmux**, see Figure 6.5. **bdmux** was originally developed by Peeters [81] for use with the Philips in-house asynchronous design language Tangram [8], where its function was to implement the non-arbitrating merge of

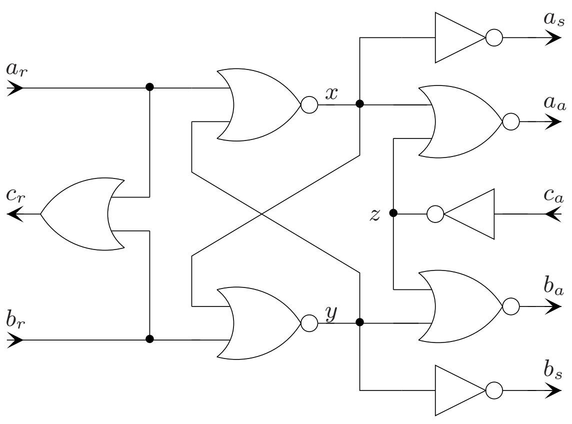


Figure 6.5: Single-rail multiplexer.

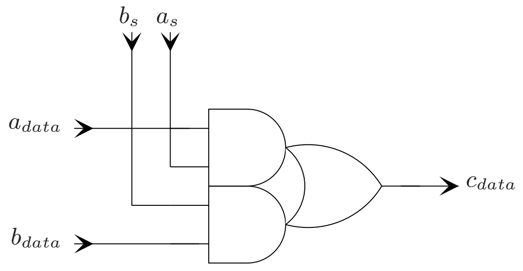


Figure 6.6: Data value multiplexing using a_s and b_s.

two bundled-data channels, $(a_r, a_a), (b_r, b_a)$, onto a single bundled-data channel c. This merger can be described using the conventional trace-expression:

$$[\ (b_r^+; c_r^+; c_a^+; b_a^+; b_r^-; c_r^-; c_a^-; b_a^-) \mid (a_r^+; c_r^+; c_a^+; a_a^+; a_r^-; c_r^-; c_a^-; a_a^-)\ ]$$

which asserts that data is exchanged either from a to c or from b to c an arbitrary number of times. **bdmux** is interesting because it also defines two outputs a_s and b_s that denote *level-sensitive* control wires used to multiplex values on the datapath, see Figure 6.6. In order to describe the behaviour of a_s and b_s using event-oriented notations it is necessary to augment the trace-expression above to describe when a_s and b_s should rise and fall. However, a_s and b_s only change value during a data exchange if they need to and this extension is therefore non-trivial. A conventional trace-expression that describes the merger operation *and* includes the behaviour of a_s and b_s is as follows:

$$
\begin{array}{ll}
[& [b_r^+; c_r^+; c_a^+; b_a^+; b_r^-; c_r^-; c_a^-; b_a^-];\ a_r^+; (a_s^+ \parallel b_s^- \parallel (c_r^+; c_a^+; a_a^+; a_r^-)); c_r^-; c_a^-; a_a^-; \\
 & [a_r^+; c_r^+; c_a^+; a_a^+; a_r^-; c_r^-; c_a^-; a_a^-];\ b_r^+; (b_s^+ \parallel a_s^- \parallel (c_r^+; c_a^+; b_a^+; b_r^-)); c_r^-; c_a^-; b_a^- \quad] \\
| & \\
[& [a_r^+; c_r^+; c_a^+; a_a^+; a_r^-; c_r^-; c_a^-; a_a^-];\ b_r^+; (b_s^+ \parallel a_s^- \parallel (c_r^+; c_a^+; b_a^+; b_r^-)); c_r^-; c_a^-; b_a^-; \\
 & [b_r^+; c_r^+; c_a^+; b_a^+; b_r^-; c_r^-; c_a^-; b_a^-];\ a_r^+; (a_s^+ \parallel b_s^- \parallel (c_r^+; c_a^+; a_a^+; a_r^-)); c_r^-; c_a^-; a_a^- \quad]
\end{array}
$$

The problem with describing the behaviour of a_s, b_s using event-oriented notations is that a_s, b_s are level-sensitive not event-driven. In the context of Tangram, a_s and b_s are defined according to a 'late' data-valid scheme: the data bundled on output channel c must have the right value multiplexed on it by the time c_r falls. Proposition-oriented verification permits the *levels* on a_s, b_s to be bound to certain other events by extending the notion of an 'event' to include any active proposition. An alternative specification for the merger operation using a Veraci-fragment is as follows:

```
protocol [ (br+;cr+;ca+;ba+;br-;(cr- & ~as & bs);ca-;ba-) |.
           (ar+;cr+;ca+;aa+;ar-;(cr- & ~bs & as);ca-;aa-) ]
```

This Veraci-fragment cleanly expresses the intended use of a_s, b_s in a late data-valid scheme. It is not however equivalent to the trace-expression outlined previously since it does not assert exactly when a_s, b_s can change value, only that that a_s, b_s hold the right value at the right time. A protocol automaton for this Veraci-fragment, excluding transitions to the dead states `rej` and `err`, is shown in Figure 6.7. An important observation to make of Figure 6.7 is that neither a_s nor b_s is in the sort for `(cr- &  as & bs)` or `(cr- &  bs & as)`, and therefore that the protocol automaton can only observe but not constrain the values on a_s, b_s.

6.4 Event Abstraction

Active propositions abstract conventional definitions of an event to include a more generic class of component network transitions than x^*, x^+ or x^-. The purpose of this section is to demonstrate the practical benefits of active propositions to the design of asynchronous circuits that employ a dual-rail delay insensitive data encoding scheme [12], although the demonstration given generalises directly to arbitrary delay insensitive data encoding schemes. If G is the generic dual-rail logic gate shown in Figure 6.8 then the correct operation of G is dependent on G only asserting a data value on its output after both its

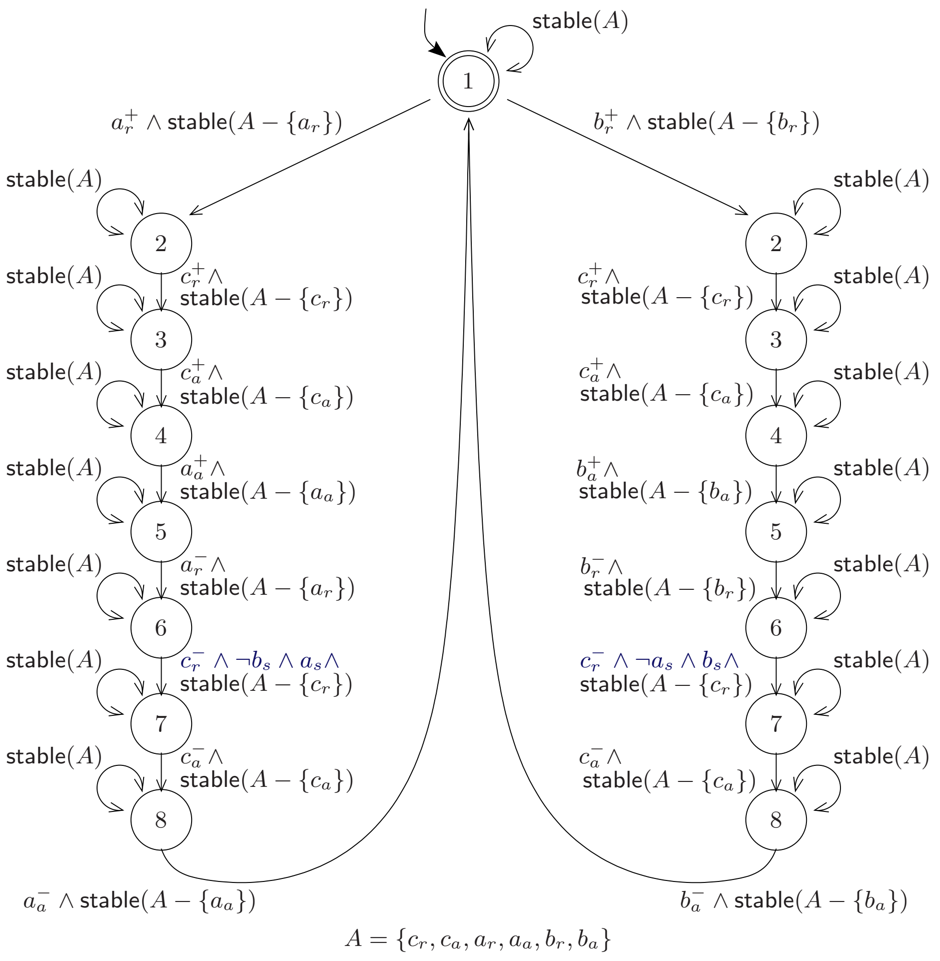

stable(A)
1
$a_r^+ \wedge$ stable($A - \{a_r\}$)
$b_r^+ \wedge$ stable($A - \{b_r\}$)
2
$c_r^+ \wedge$ stable($A - \{c_r\}$)
3
$c_a^+ \wedge$ stable($A - \{c_a\}$)
4
$a_a^+ \wedge$ stable($A - \{a_a\}$)
$b_a^+ \wedge$ stable($A - \{b_a\}$)
5
$a_r^- \wedge$ stable($A - \{a_r\}$)
$b_r^- \wedge$ stable($A - \{b_r\}$)
6
$c_r^- \wedge \neg b_s \wedge a_s \wedge$ stable($A - \{c_r\}$)
$c_r^- \wedge \neg a_s \wedge b_s \wedge$ stable($A - \{c_r\}$)
7
$c_a^- \wedge$ stable($A - \{c_a\}$)
8
$a_a^- \wedge$ stable($A - \{a_a\}$)
$b_a^- \wedge$ stable($A - \{b_a\}$)
$A = \{c_r, c_a, a_r, a_a, b_r, b_a\}$

Figure 6.7: A mixed levels-events protocol automaton.

inputs have asserted a data value, and upon G only clearing the data value on its output after both its inputs have cleared their data value.

These two dependencies combined are known as Seitz's rules [90], and can be likened to the semantics of a C-element in which events on wires have been *abstracted* to events on dual-rail data values. An example of this likeness is shown in Figure 6.10, where the dual-rail AND-gate component shown in Figure 6.9 is verified as conforming to Seitz's rules: The protocol definition, `[(dr_event(x0,x1) || dr_event(y0,y1)); dr_event(z0,z1)]`, is structurally identical to the Veraci-fragment `[(a*||b*);c*]`, which is a suitable specification for any C-element module with inputs a, b and output c.

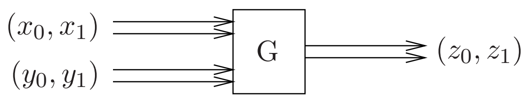


Figure 6.8: A generic 2-input dual-rail logic gate.

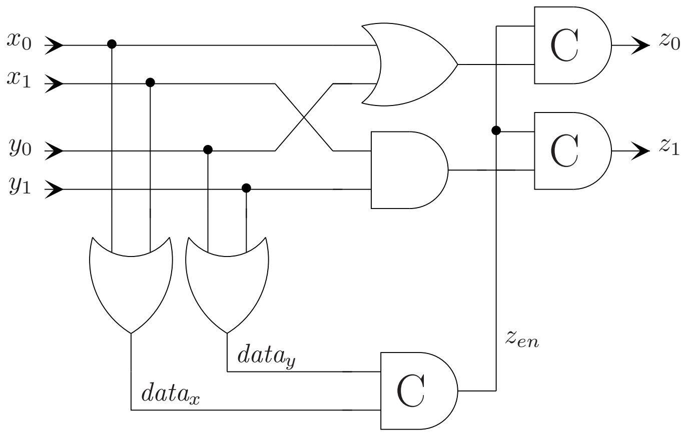


Figure 6.9: speed independent dual-rail AND-gate.

```
module SI DRand(x0,x1, y0,y1, z0,z1);
  input  x0,x1, y0,y1;
  output z0,z1;
  wire   data_x, data_y, z_en, z0_pre, z1_pre;

  celem (z0,   z0_pre, z_en);
  celem (z1,   z1_pre, z_en);
  celem (z_en, data_x, data_y);

  assign data_x = x0 | x1;
  assign data_y = y0 | y1;
  assign z0_pre = x0 & y0;
  assign z1_pre = x1 | y1;
<<
  initial dr_clear(x0, x1) & dr_clear(y0, y1)
  protocol [ (dr_event(x0,x1)  || dr_event(y0,y1));  dr_event(z0,z1) ]
>>
endmodule
```

Figure 6.10: Veraci program for a speed independent dual-rail AND-gate.

6.5 Extended Safety Conditions

The purpose of this section is merely to observe that Veraci-fragments are able to express safety conditions, which are assertions that must hold for every reachable state of a system. In the context of Veraci, a safety condition can be expressed using a network proposition p, and can be verified using the Veraci-fragment `verify always [`p`]`. Veraci can also be used to *enforce* a safety condition p using the Veraci-fragment `cut always [`p`]`.

Network propositions reason over current-next state pairs not just current states. In this sense `verify always [`p`]` and `cut always [`p`]` actually assert that p hold for every reachable component network *transition* not state. The extension of safety conditions to reachable transitions rather than reachable states has the benefit that it is possible to express safety conditions that require both the current and next component network states to be known. An example of such an extended safety condition is semi-modularity:

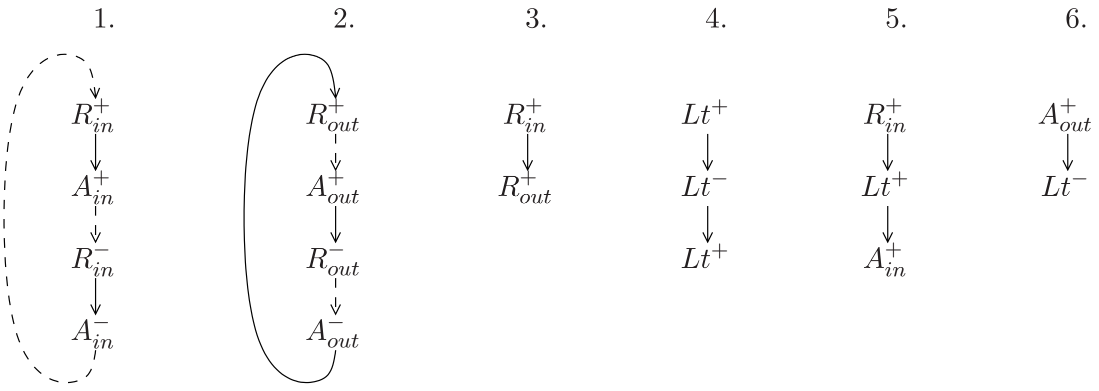
If `m` is a Verilog module input to Veraci then `m` is said to be semi-modular if every wire in `m` is semi-modular, see Section 6.2.4. Using Veraci the semi-modularity of `m` can be verified merely by placing the Veraci-fragment `verify always [semi]` at some point in the definition of `m` as follows:

```
module m(...);
  ...
  << verify always [semi] >>
endmodule
```

6.6 Causality Fragments

The purpose of this section is to demonstrate how proposition-expressions can be used to reason with fragmented assertions that do not describe cyclic behaviours. The example circuits used to facilitate this demonstration are the four-phase bundled-data latch controllers described by Furber and Day [39].

A four-phase bundled-data latch controller is a circuit used to control a single bundled-data pipeline stage using a four-phase handshake protocol on both the input and output data channels. The latch controllers described by Furber and Day are considered as having two input terminals, R_{in}, A_{out}, and three output terminals, R_{out}, A_{in}, Lt. Terminals R_{in}, A_{in} denote the input data channel handshake control wires, and terminals R_{out}, A_{out} denote the output data channel handshake control wires. Terminal Lt denotes a signal used to



$$\begin{tikzcd}
1. & 2. & 3. & 4. & 5. & 6. \\
R_{in}^{+} \arrow[d] & R_{out}^{+} \arrow[d, dashed] & R_{in}^{+} \arrow[d] & Lt^{+} \arrow[d] & R_{in}^{+} \arrow[d] & A_{out}^{+} \arrow[d] \\
A_{in}^{+} \arrow[d, dashed] & A_{out}^{+} \arrow[d] & R_{out}^{+} & Lt^{-} \arrow[d] & Lt^{+} \arrow[d] & Lt^{-} \\
R_{in}^{-} \arrow[d] & R_{out}^{-} \arrow[d, dashed] & & Lt^{+} & A_{in}^{+} & \\
A_{in}^{-} \arrow[uuu, dashed, bend left=60] & A_{out}^{-} \arrow[uuu, bend left=60] & & & &
\end{tikzcd}$$

Figure 6.11: Furber-Day specification fragments for a four-phase latch controller.

open and close transparent latches on a bundled-data datapath: if Lt is high then the datapath latches are closed, and if Lt is low then the datapath latches are open.

In their paper, Furber and Day describe several different four-phase latch controllers, each of which which provides a different level of decoupling between the input and output data channels. Each of these latch controllers is manually synthesised from a different closed STG, and each of these closed STG specifications is itself constructed as a composition of six independent STG 'fragments' identified by Furber and Day as sufficient to ensure correct latch controller operation. A copy of these six fragments is shown in Figure 6.11 and can be interpreted as follows:

1. A_{in} and R_{in} obey a four-phase handshake protocol.
2. A_{out} and R_{out} obey a four-phase handshake protocol.
3. Input and output handshakes never get out of step.
4. Rising and falling events on Lt alternate.
5. It is never possible to have have an A_{in}^{+} after a R_{in}^{+} without the datapath latches closing in between, denoted by an intervening Lt^{+}.
6. It is never possible to have two A_{out}^{+}s without the datapath latches opening in between, denoted by an intervening Lt^{-}.

Verification that any of Furber and Day's latch controllers conforms to its original closed STG specification is not difficult, and can be done using an existing verification tool such as Versify [86]. The application of Veraci to four-phase latch controllers is interesting because using Veraci it is possible to express the six generic STG 'fragments' directly as

Veraci-fragments and therefore to verify that all of Furber and Day's latch controllers do indeed conform to the *same* original specification.

Verification using the six fragments shown in Figure 6.11 is difficult because Fragments 3-6 do not identify cyclic behaviours of the form `[`$\cdots$`]`, and do not describe both rising and falling events for wires on which they depend. For example in Fragment 6, the behaviour of event Lt^{-} is described but the behaviour of event Lt^{+} is not. This difficulty can be overcome in Veraci by translating Fragments 3-6 into expressions of the form `verify never [1];`x`;([~`y`] &.` B`)` where x, y, and B are defined as follows:

$x =$ A network proposition identifying a start event.
$B =$ A Veraci-expression identifying a set of behaviours to observe after the start event.
$y =$ A network proposition identifying an action that must happen at some point during the observation period.

Expressions of this form are referred to here as *causality*-fragments since they assert that a certain action, y, must happen during certain fragments, B, of network executions. Causality-fragments are built from two different types of proposition-oriented sub-expression:

- `[1];`E where E is any proposition-expression. Ignores zero or more component network transitions before beginning to match E.

- `[`p`] &.` E where E is any proposition-expression and p is any network proposition. Matches network executions that match E but also never violate the safety condition asserted by p.

In the context of Figure 6.11, Fragments 3-6 can be expressed using the causality-fragments outlined in Figure 6.12. Fragment 4 has no associated causality-fragment since it asserts only that rising and falling events on wire Lt alternate, which, since Veraci does not consider liveness, is always true. Of particular interest is Fragment 3, which asserts that the two handshake channels do not get out of step. An initial interpretation suggests that the required causality-fragment for Fragment 3 should be `[1];Rin+;([~Rout+] &. trace(Rin+))`. However, in practice an R_{out}^{+} need only happen before the latch controller acknowledges the start of a *second* input data item. The required causality-fragment can therefore only assert that R_{out}^{+} happens before a second A_{in}^{+} event is traced on terminal A_{in}. This can be achieved by observing those network execution fragments matching `trace(Ain+;Ain-;Ain+)`.

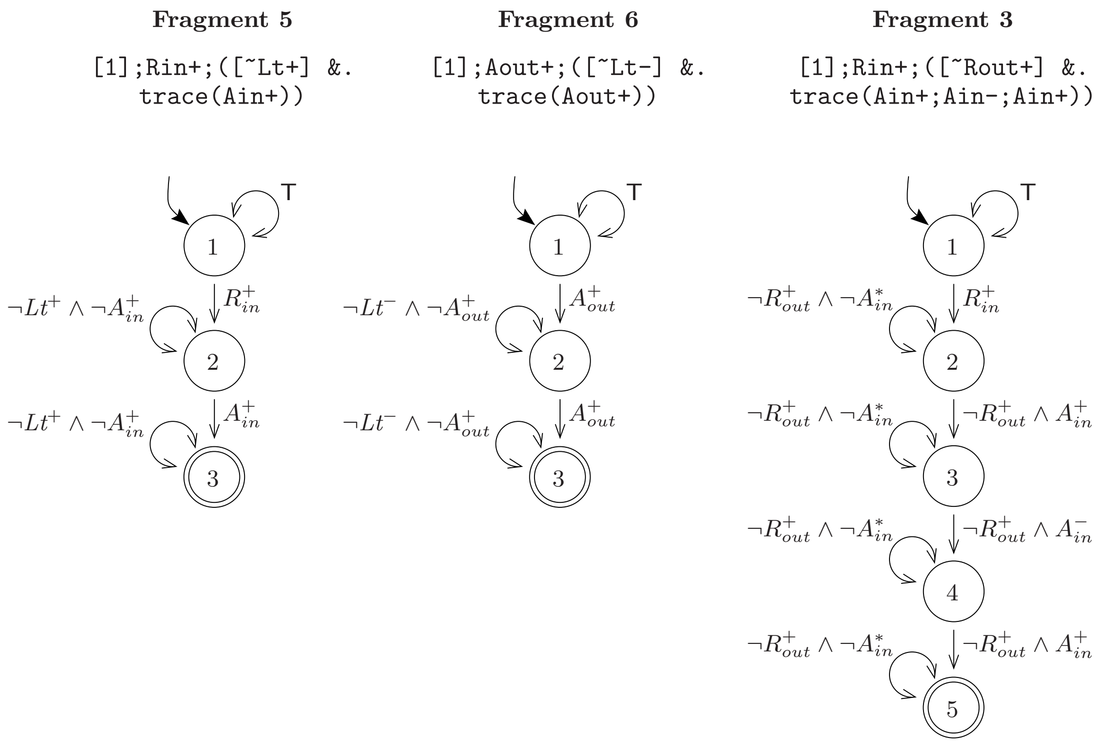

Fragment 5
[1];Rin+;([~Lt+] &.
trace(Ain+))
Fragment 6
[1];Aout+;([~Lt-] &.
trace(Aout+))
Fragment 3
[1];Rin+;([~Rout+] &.
trace(Ain+;Ain-;Ain+))
T
1
R_{in}^{+}
$\neg Lt^{+} \wedge \neg A_{in}^{+}$
2
A_{in}^{+}
$\neg Lt^{+} \wedge \neg A_{in}^{+}$
3
T
1
A_{out}^{+}
$\neg Lt^{-} \wedge \neg A_{out}^{+}$
2
A_{out}^{+}
$\neg Lt^{-} \wedge \neg A_{out}^{+}$
3
T
1
R_{in}^{+}
$\neg R_{out}^{+} \wedge \neg A_{in}^{*}$
2
$\neg R_{out}^{+} \wedge A_{in}^{+}$
$\neg R_{out}^{+} \wedge \neg A_{in}^{*}$
3
$\neg R_{out}^{+} \wedge A_{in}^{-}$
$\neg R_{out}^{+} \wedge \neg A_{in}^{*}$
4
$\neg R_{out}^{+} \wedge A_{in}^{+}$
$\neg R_{out}^{+} \wedge \neg A_{in}^{*}$
5

Figure 6.12: Causality fragments in Veraci.

```
module SI full_dec_latch(Rin, Ain, Rout, Aout);
  input Rin, Aout;
  output Rout, Ain;
  wire Lt, neg_Aout, neg_Rout, A, B, neg_B;

  assign neg_Aout = ~Aout;
  assign neg_Rout = ~Rout;
  assign    neg_B = ~B;
  assign       Lt = A;

  celem (A, +Rin, neg_B, neg_Rout, -neg_Aout);
  celem (B, -Lt, Ain);
  celem (Rout, +neg_Aout, A);
  celem (Ain, +Lt, neg_B, -Rin);
<<
  initial ~Rin & ~Aout

  protocol [Rin+;Ain+;Rin-;Ain-]      // fragment 1
  protocol [Rout+;Aout+;Rout-;Aout-]  // fragment 2

  verify never [1];Aout+;([~Lt-]   &. trace(Aout+))          // fragment 6
  verify never [1];Rin+; ([~Lt+]   &. trace(Ain+))           // fragment 5
  verify never [1];Rin+; ([~Rout+] &. trace(Ain+;Ain-;Ain+)) // fragment 3
>>
endmodule
```

Figure 6.13: Veraci program for the fully-decoupled latch controller.

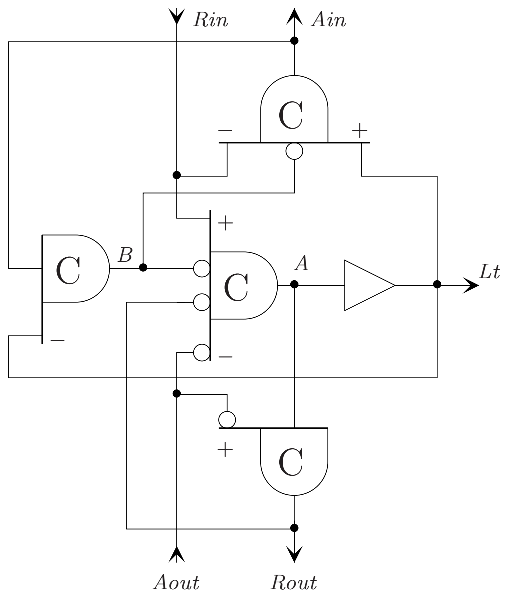


Figure 6.14: Fully-decoupled latch controller circuit.

An example Veraci program for verifying a bundled-data four-phase latch controller is shown in Figure 6.13. This Veraci program describes Furber and Day's *fully-decoupled* latch controller, and can be used to verify that the circuit shown in Figure 6.14 does indeed conform to the six generic STG fragments which originally inspired it.

6.7 Timing Assumptions

The purpose of this section is to outline some of the ways in which Veraci-fragments can be used to verify circuits that contain timing assumptions.

6.7.1 Fundamental-mode operation

Fundamental-mode operation [102] is an assertion about the behaviour of a circuit's environment. This assertion guarantees that a circuit is always allowed to stabilise between successive events on its input wires. Fundamental-mode operation simplifies asynchronous design considerably but places timing restrictions on the environment of a circuit.

If `m` is a Verilog module input to Veraci, and `m` has input wires `i1`, $\cdots$,`in`, then the fundamental-mode operation of `m` can be assumed by augmenting the definition of `m` with the safety condition: `cut always [excited => ~i1* &` $\cdots$ `& ~in*]`

This Veraci-fragment ensures that no input wire changes value when `m` has internal excitation. Fundamental-mode operation can be localised either by restricting `excited` to apply to only a subset of wires in `m`, or alternatively by removing certain input wires from the right-hand side of the implication.

6.7.2 Gate Delay Removal

All circuits modelled by Veraci exhibit arbitrary but finite inertial delay for each underlying primitive component. Arbitrary gate delay is a very conservative assumption and practical circuits may need to violate this assumption in certain cases. For example, the ability to assert that certain gate delays should be ignored is often required by the synthesis tool Petrify when an inverter precedes a complex gate input [27].

The removal of gate delay in Veraci can be viewed as a form of safety constraint: if a zero-delay gate output is ever enabled to change value then it must do so immediately. For example, if `w` is the output of a gate `G` then the removal of delay from gate `G` can be achieved by augmenting the Verilog module in which `G` is instanced with the Veraci-fragment `cut always [excited(w) => w*]`.

6.7.3 Relative Timing Assumptions

Relative timing assumptions are assertions regarding the relative delays between events. The application of relative timing to two divergant paths in a circuit has been considered at length by Negulescu [78] using a technique called *chain-constraints*. The purpose of a chain-constraint is to enforce a timing assumption of the form $ex_1x_2 \cdots x_n < ey_1y_2 \cdots y_m$ where $ex_1x_2 \cdots x_n$ and $ey_1y_2 \cdots y_m$ denote two divergent sequences of events x_i, y_j with a common start event e. Each such timing assumption asserts that after event e the sequence $x_1x_2 \cdots x_n$ must complete before the sequence $y_1y_2 \cdots y_m$.

Negulescu implements chain-constraints as a process-space automaton which mimics an $n + m + 1$ dimensional hypercube. This automaton has 2^{n+m+1} states each of which identifies a particular level on each of the $n+m+1$ wires. A certain state of the hypercube automaton is identified as a *base* state from which 'tracking' of the two divergent event sequences x_i and y_j can begin. If the y_m event is ever tracked before the x_i sequence completes then the hypercube automaton enters an a special error state which can then be detected by Negulescu's verifier, Firemaps. A chain-constraint is a conservative timing assumption in that it identifies a specific sequence of events from a specific base state. If the initial levels on the $n + m + 1$ wires are wrong or an x_i, y_j event ever happens out of order then the chain-constraint 'aborts' tracking.

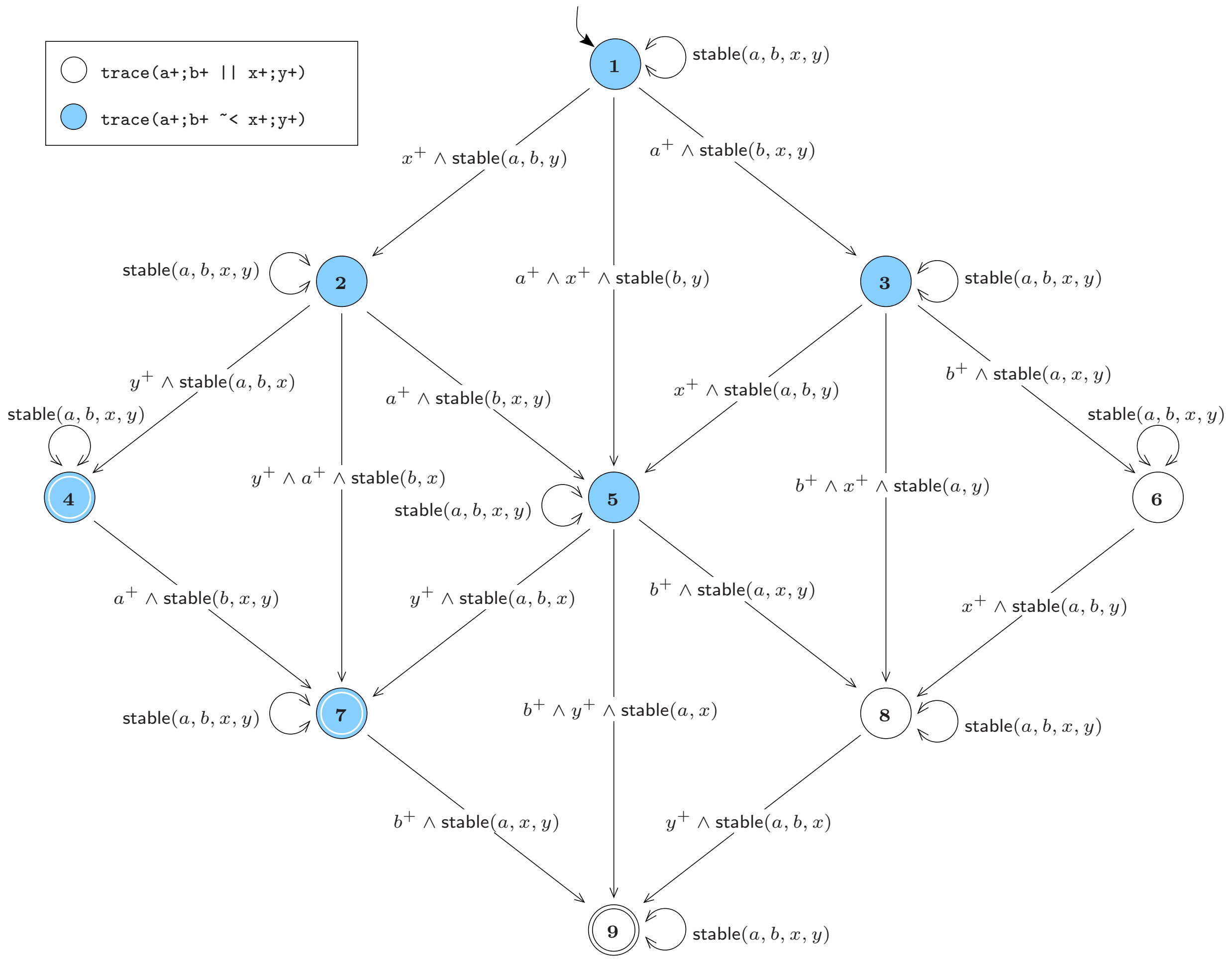


Figure 6.15: Proposition-automata outlining the relationship between $\not\prec$ and $\|$.

The purpose of this section is to outline an alternative approach to chain-constraints that is based on the the biased composition construct $\not\prec$, denoted `~<` in Veraci. The motivation behind biased-composition was the observation that the tracking of two concurrent event paths equates to the parallel composition of two event paths, except that the objective is to detect when one path completes before the other, and not when both paths have completed.

For example, the Veraci-expressions `trace(a+;b+ || x+;y+)` and `trace(a+;b+ ~< x+;y+)` can be compared as shown in Figure 6.15: the biased composition of `a+;b+` and `x+;y+` matches any network execution where `x+;y+` completes before `a+;b+`, denoted by States 4 and 7. The proposition-automaton for `trace(a+;b+ ~< x+;y+)` contains those states in the proposition-automaton for `trace(a+;b+ || x+;y+)` that are predecessors of States 4 and 7.

To convert the biased composition `trace(a+;b+ ~< x+;y+)` into a relative timing assumption a common start event must be introduced from which tracking must begin. In the context of proposition-oriented verification this start event can be any active proposition, p. If $pa^+b^+ < px^+y^+$ is a relative timing assumption then implementation of $pa^+b^+ < px^+y^+$ as a Veraci-fragment can be achieved as follows:

```
cut never [1];trace(p;a+;b+ ~< p;x+;y+)
```

which may also be written:

```
cut never [1];trace(p;(a+;b+ ~< x+;y+))
```

The use of biased composition to implement relative timing assumptions has two key advantages over Negulescu's chain-constraints:

- If $x_1 \cdots x_n$ and $y_1 \cdots y_m$ are two sequences of events, then the biased composition of x_i and y_i has $n \times m$ states whereas a hypercube automaton for x_i and y_i has $2^n \times 2^m$ states. Chain-constraints require $2^n \times 2^m$ states to track the *levels* on every wire. These levels are needed to identify a *base* state from which tracking begins. Proposition-oriented verification permits p to be augmented with level constraints, see Section 6.3, and level identification by proposition automaton state is therefore not needed.

- Biased composition can be applied to more complex sub-expressions than just event sequences. In particular relative timing assumptions with a common start event can often be combined into a single Veraci-fragment containing only one biased composition construct.

As an example application of chain-constraints to asynchronous design Negulescu and Peeters consider the bundled-data multiplexer circuit, **bdmux**, see Section 6.3, which requires the following relative timing assumptions in order to function correctly:

$$\begin{aligned}
a_r^+ x^- a_s^+ &< a_r^+ c_r^+ c_a^+ z^- a_a^+ a_r^- c_r^- \\
a_r^+ x^- y^+ b_s^- &< a_r^+ c_r^+ c_a^+ z^- a_a^+ a_r^- c_r^- \\
a_r^+ x^- y^+ &< a_r^+ c_r^+ c_a^+ z^- \\
b_r^+ y^- b_s^+ &< b_r^+ c_r^+ c_a^+ z^- b_a^+ b_r^- c_r^- \\
b_r^+ y^- x^+ a_s^- &< b_r^+ c_r^+ c_a^+ z^- b_a^+ b_r^- c_r^- \\
b_r^+ y^- x^+ &< b_r^+ c_r^+ c_a^+ z^-
\end{aligned}$$

In the context of proposition-oriented verification with Veraci these six fragments can converted into six Veraci-fragments as follows:

```
cut never [1];trace((ar+ & x & ~y);(x-;y+ ~< cr+;ca+;z-))
cut never [1];trace((ar+ & x & ~y);(x-;as+ ~< cr+;ca+;z-;aa+;ar-;cr-))
cut never [1];trace((ar+ & x & ~y);(x-;y+;bs- ~< cr+;ca+;z-;aa+;ar-;cr-))

cut never [1];trace((br+ & ~x & y);(y-;x+ ~< cr+;ca+;z-))
cut never [1];trace((br+ & ~x & y);(y-;bs+ ~< cr+;ca+;z-;ba+;br-;cr-))
cut never [1];trace((br+ & ~x & y);(y-;x+;as- ~< cr+;ca+;z-;ba+;br-;cr-))
```

Furthermore, these six Veraci-fragments can be compacted into four by combining like sub-expressions, resulting in a completed Veraci program for **bdmux** as shown in Figure 6.16.

Implementation of relative timing using biased composition also has relevance with respect to recent work by Cortadella [25] and Peña [82] on the application of relative timing assumptions to state transition graphs using event structures. In particular the underlying state graph model for an event structure that implements the biased composition $x_1^*; x_2^*; \ldots; x_n^* \not< y_1^*; y_2^*; \ldots; y_m^*$ also has $n \times m$ states. Biased composition is different to an event structure since its semantics is given with respect to two arbitrary proposition-oriented trace-expressions, whereas event structures are a representation of partial orderings between events. A detailed discussion of biased composition in the context of event structures is beyond the scope of this thesis.

6.8 Clock Generator

The purpose of this section is to document an application of Veraci that is not due to the author, and to present a demonstration of counter-example generation in Veraci. The application chosen is that of a clock generator, **clkgen**, see Figure 6.17. The objective of **clkgen** is to generate a synchronous clock, *clk*, using delay line, D, which may be stopped and started under the control of a four-phase handshake protocol on *sleep*, *sleeping*, and may be calibrated according to an external source when not in use. Decoupling between the delay line and arbitration on *sleep* is achieved using a C-element to combine an arbitrated clock-grant, *gnclk*, with the output of the delay line, *delay_nclk*.

clkgen has been implemented in silicon by Taylor using a two-input AND-gate to detect sleeping according to the product $sleeping = nclk \wedge gsleep$ [98]. Unfortunately, this product is insufficient to ensure the inactivity of D, and the silicon implementation of **clkgen** therefore exhibits erroneous glitching during calibration [72].

```
module SI BDmux(cr, ca,  ar, aa,  br, ba,  as, bs);
  input ar, br, ca;
  output cr, aa, ba, bs, as;
  wire x, y, z;

  assign cr  = (ar | br);
  assign x = ~(ar | y);
  assign y = ~(br | x);
  assign as = ~x;
  assign bs = ~y;
  assign aa = ~(x | z);
  assign ba = ~(y | z);
  assign z = ~ca;
<<
  initial ~ar & ~br & ~ca

  cut never [1];trace((ar+ & x & ~y);(x-;y+ ~< cr+;ca+;z-))
  cut never [1];trace((ar+ & x & ~y);(x-;(as+||y+;bs-) ~< cr+;ca+;z-;aa+;ar-;cr-))

  cut never [1];trace((br+ & ~x & y);(y-;x+ ~< cr+;ca+;z-))
  cut never [1];trace((br+ & ~x & y);(y-;(bs+||x+;as-) ~< cr+;ca+;z-;ba+;br-;cr-))

  protocol [ (br+;cr+;ca+;ba+;br-;(cr- & ~as' & bs');ca-;ba-) |.
             (ar+;cr+;ca+;aa+;ar-;(cr- & ~bs' & as');ca-;aa-) ]
>>
endmodule
```

Figure 6.16: Veraci program for the bundled-data multiplexer control circuit.

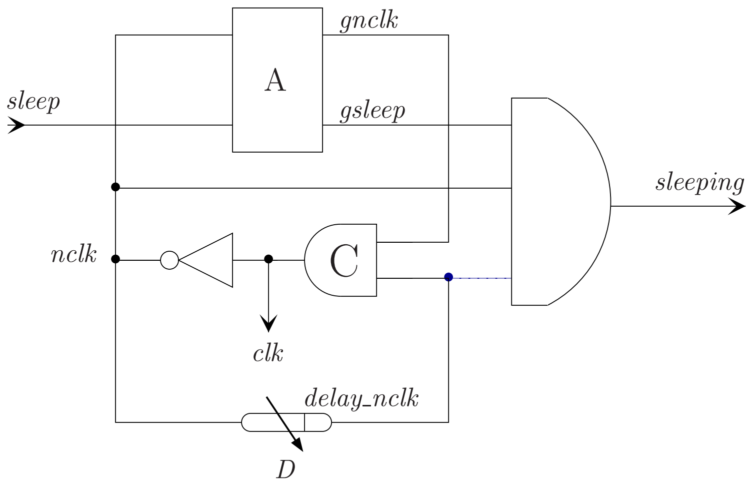


Figure 6.17: Clock generator circuit.

The correct operation of **clkgen** under the augmented product $sleeping = nclk \wedge gsleep \wedge delay\_nclk$ has since been verified by Taylor using the Veraci program shown in Figure 6.18 [98]. This Veraci program uses three Veraci-fragments to concisely specify the desired operation of **clkgen** as follows:

- `protocol [sleep*; sleeping* ]`
  Requests on *sleep* alternate with acknowledgements on *sleeping*.

- `verify always [ sleep & sleeping => ~excited(delay_nclk) ]`
  delay_nclk is inactive whenever **clkgen** is sleeping.

- `verify always [ ~gsleep => excited(clk,nclk,delay_nclk,gnclk) ]`
  Clock generation is active whenever **clkgen** is awake: If *sleep* is not granted then there is always excitation somewhere in the two decoupled clock generation loops.

If the underlined "`& delay_nclk`" is removed from Figure 6.18, then Veraci generates a counter-example to demonstrate an erroneous execution in which `sleeping+` occurs when `nclk=1` but `delay_nclk=0`, see Figure 6.19. The execution predicted by this counter example equates directly to the observed erroneous operation of **clkgen** in silicon.

Note that each line of a counter-example output by Veraci lists a set of events that occur simultaneously under a Multiple-Winner execution model, whereas events on adjacent lines are considered ordered in time from top to bottom.

6.9 Veraci Performance

The underlying verification engine behind Veraci implements the symbolic verification procedure **pcheck** as described in Chapter 5. Veraci is also capable of applying either of the following two BDD-based search-optimisations on demand:

- **Option -r** Frontier-set reduction.

- **Option -q** Early quantification using a partitioned transition relation.

All proposition automata constructed by Veraci are minimised prior to their symbolic BDD encoding, and all symbolic searching performed by Veraci uses an atomic relational product operator wherever possible. Veraci execution times for each of the example circuits contained in this thesis are summarised in Table 6.3.

```
module SI clkgen(clk, sleeping, sleep);
  output sleeping, clk;
  input  sleep;
  wire   nclk, delay_nclk, gsleep, gnclk;

  assign       nclk = ~clk;
  assign   sleeping =  nclk & gsleep & delay_nclk;

  celem    (clk, gnclk, delay_nclk);
  buf     D(delay_nclk, nclk);
  arbiter A(gsleep, sleep, gnclk, nclk);

<<
  initial ~sleep & ~clk
  protocol [sleep*; sleeping* ]

  // when sleeping delay line inactive
  verify always [ (sleep & sleeping)' => ~excited(delay_nclk)]

  // when awake delay line active
  verify always [ ~gsleep => excited(clk,nclk,delay_nclk,gnclk) ]
>>
endmodule
```

Figure 6.18: Veraci program for the clock generator clkgen.

```
(violation on line 18)
    verify always [ sleep & sleeping => ~excited(delay_nclk)]

Initial State:
        nclk = 1
        delay_nclk = 1
        sleeping = 0
        clk = 0
        gsleep = 0
        gnclk = 0
        sleep = 0

Path:
        gnclk+  sleep+
        clk+
        nclk-
        delay_nclk-  gnclk-
        clk-
        nclk+  gsleep+
        sleeping+
```

Figure 6.19: Example of a counter-example output from Veraci.

Execution platform: 400MHz Sun UltraSPARC II

Circuit	Time (s)
Bundled-data multiplexer	1.37
Dual-rail AND-gate	0.05
Fully-decoupled latch controller	0.24
Clock generator	0.01
Ask-do module	1.24
A-odd module	0.25
Dual-rail full-adder	0.25

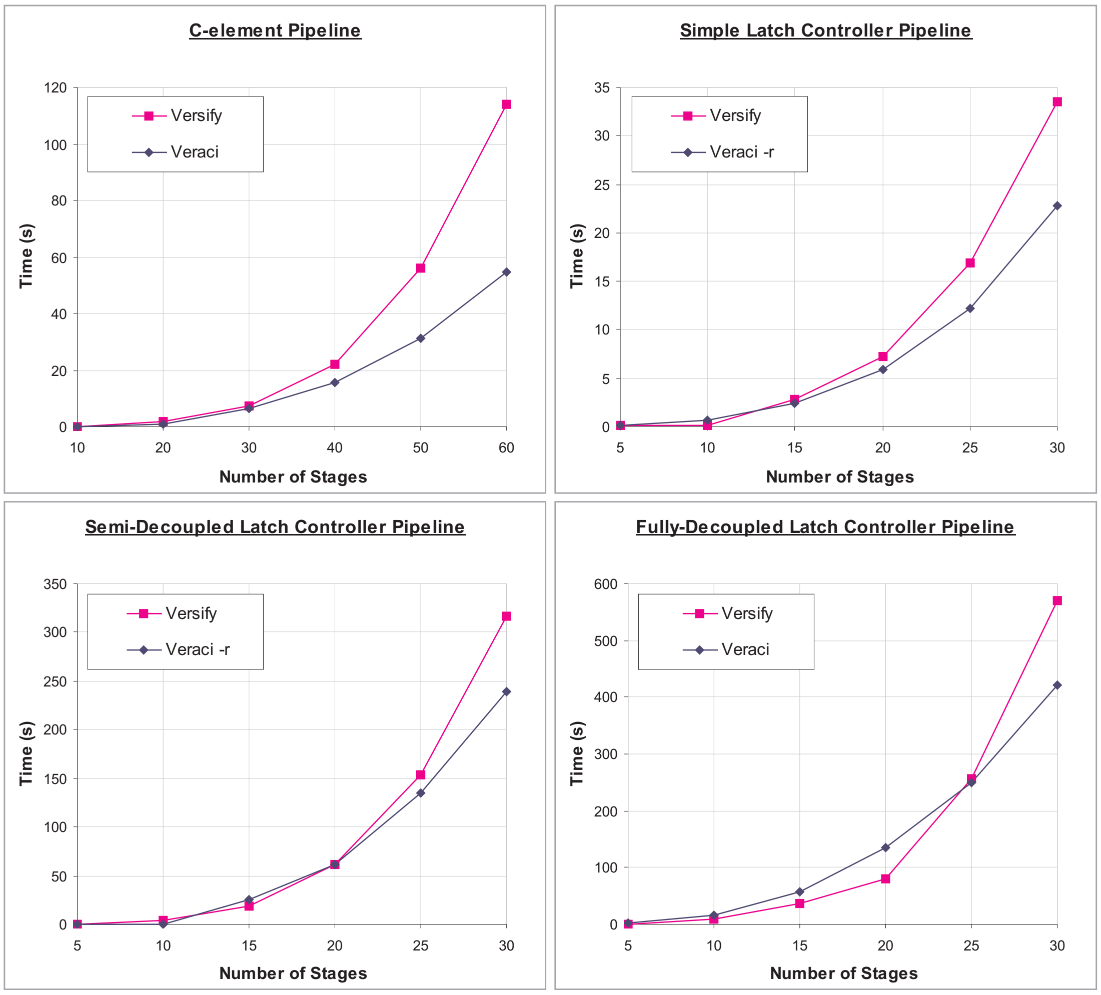
Table 6.3: Veraci verification times for example circuits in this thesis.

The verification times shown in Table 6.3 are all small, and do not give any indication of the relative performance of Veraci with respect to other verification programs. However, since proposition-oriented behaviours are more expressive than their level and event-oriented counterparts it is not in general possible to apply a Veraci program to an alternative verification program. An improved evaluation of Veraci performance can be obtained by benchmarking Veraci execution times according to a scalable circuit, and to pick a specification for this circuit that can be applied to alternative verification programs.

The benchmarks shown in Figure 6.20 summarise the evaluation of Veraci performance with respect to an alternative event-oriented verification tool also based on BDDs called Versify [86]. Each of these benchmarks relates Veraci and Versify execution times for an asynchronous pipeline constructed from a particular type of control element. The specification used for each benchmark was `protocol [Rin*;Ain*] || [Rout*;Aout*]` where `Rin`,`Ain` denoted the pipeline input channel and `Rout`,`Aout` denoted the pipeline output channel.

Proposition oriented verification makes both current and next-state variables explicit in the semantics of a network proposition. The ability to reason over (current,next)-state pairs enables proposition-oriented verification to unify levels and events, leading to notations that are more flexible than conventional event-oriented notations. This increased flexibility was expected to incur a cost in performance, yet the benchmarks in Figure 6.20 indicate a persistent eventual performance gain of Veraci over Versify.

A possible explanation for this performance gain is that Veraci adopts a Multiple-Winner execution model, whereas Versify adopts a Single-Winner execution model: Multiple-Winner execution models can step through several concurrent actions in one circuit model transition. This extra functionality equates to a transition relation with more transitions,



Execution platform: 400MHz Sun UltraSPARC II

Figure 6.20: A comparison between Veraci to Versify.

however when encoded as a BDD, a larger transition relation need not imply a larger BDD. In the context of Figure 6.20, a fully-decoupled latch controller pipeline has more sequential behaviour internal to each stage than a simple C-element pipeline, however any pipeline structure is an inherently concurrent system. Under the proposed explanation, both types of pipeline therefore benefit from a Multiple-Winner execution model, however the C-element pipeline will benefit the most first.

An extension from Single to Multiple-Winner execution models relates directly to an extension from events to propositions in that both extensions abandon the notion of symbol exclusivity: two different network propositions can be matched simultaneously, and two events can happen simultaneously under a Multiple-Winner execution model. To construct a representation of the conventional trace-expression $a \parallel b$ under a Multiple-

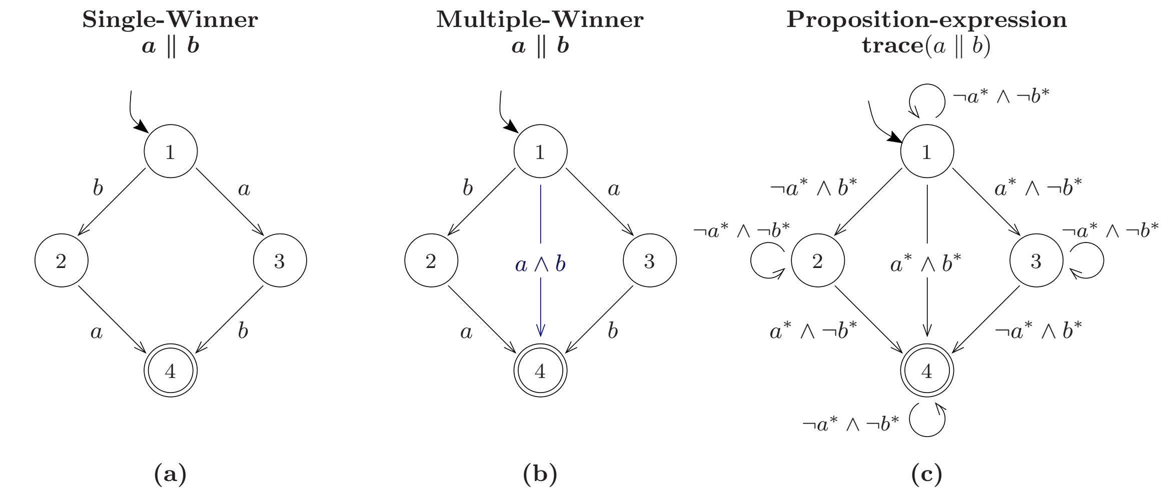


Figure 6.21: Trace-expressions and a Multiple-Winner execution model.

Winner execution model, the possibility of both a and b happening simultaneously must be accounted for, see Figure 6.21(a,b). This accountability necessitates the introduction of an explicit mechanism, such as proposition-oriented behaviours, to denote their simultaneous occurrence, see Figure 6.21(c). Proposition-oriented behaviours are significant in this context since they are the first such mechanism to be defined for trace-expressions.

6.10 Summary

The purpose of this chapter was to introduce Veraci, a proposition-oriented verification program, and to use Veraci to demonstrate the benefits of proposition-oriented behaviours over their level and event-oriented counterparts. These benefits included level-event unification, event abstraction, and relative timing assumptions using biased composition. An example application of Veraci that is not due to the author was also presented and used to facilitate the demonstration of counter-example generation in Veraci. The performance of Veraci was also evaluated, and an initial comparision made to a competing verification program called Versify.

Chapter 7
Asynchronous Montgomery Exponentiation

7.1 Introduction

Montgomery exponentiation is an efficient method for computing exponentiations in modular arithmetic [63]. These exponentiations embody the core operation responsible for encrypting and decrypting messages in RSA public-key cryptography [91]. Asynchronous circuits that use some form of delay-insensitive data-encoding scheme have a number of potential benefits over synchronous designs when subjected to power analysis attacks [71] and are therefore of considerable interest to the manufactures of cryptographic devices such as smartcards. Unfortunately, there have as yet been no published implementations of Montgomery exponentiation using asynchronous logic that take steps towards demonstrating these benefits. The Montgomery exponentiator presented in this chapter is intended to fulfil two aims:

- To demonstrate an implementation of Montgomery exponentiation using delay insensitive dual-rail asynchronous logic.
- To investigate the application of Veraci to a real design project.

In the sections that follow, Section 7.2 describes some preliminary mathematics for the chapter. Section 7.3 explains the application of Montgomery multiplication and exponentiation to RSA cryptography, and Section 7.4 explains the suitability of Montgomery multiplication to implementations in hardware. Section 7.5 extends this demonstration to describe *asynchronous* Montgomery exponentiation in hardware, and documents the design and implementation of a dual-rail asynchronous Modular exponentiator, MOD_EXP, in a $0.18\mu m$ standard-cell process. Section 7.6 concludes the chapter by highlighting the application of Veraci to the design of MOD_EXP.

7.2 Mathematical Preliminary

7.2.1 Modular Arithmetic

For every integer $a > 0$ define $\mathbb{Z}_a \overset{\text{def}}{=} \{0, \ldots, |a| - 1\}$ to denote the set of *residues* modulo a. If a and b are integers then define (b div a) and (b mod a) to be the unique integers such that $a = (b \text{ div } a)a + (b \bmod a)$ and $(b \bmod a) \in \mathbb{Z}_a$. If $(b \bmod a) = 0$ then a is said to *divide* b, written $a|b$.

Greatest Common Divisor. If a, b, and c are integers such that $a|b$ and $a|c$ then a is a common factor of b and c. Let $\mathsf{gcd}(b, c)$ denote the greatest common factor of b and c. For every pair of integers a and b, $gcd(a, b) = s_0a + t_0b$ for two other integer constants s_0 and t_0: let $V = \{v \mid v = sa + tb$ for some integers $s, t\}$. Let $h = s_0a + t_0b$ be the smallest element in V which is greater than zero and let $a = qh + p$ for some $p \in \mathbb{Z}_h$. If $a = qh + p$ then $p = a - qh = (1 - qs_0)a + (0 - qt_0)b$ and hence $p \in V$. If $p \in V$ and $p \in \mathbb{Z}_h$ then $p = 0$ since h is the smallest element in V which is greater than zero. If $p = 0$ then $a = qh$ and therefore $h|a$. By a similar argument $h|b$. Furthermore, since $h = s_0a + t_0b$ any other value which divides both a and b must also divide h. Hence $h = s_0a + t_0b = \mathsf{gcd}(a, b)$ is the greatest value such that $h|a$ and $h|b$.

Co-Prime. If b and c are integers and $\mathsf{gcd}(b, c) = 1$ then b and c are said to be *co-prime*. If a is co-prime to c and b is co-prime to c then ab is co-prime to c. Define $\mathbb{U}_c \overset{\text{def}}{=} \{b \in \mathbb{Z}_c \mid \mathsf{gcd}(b, c) = 1\}$ to denote the set of *units* modulo c and define $\phi(c) \overset{\text{def}}{=} |\mathbb{U}_c|$ to denote the number of units modulo c. If c is prime then $\phi(c) = c - 1$.

Multiplicative Inverse. If a and b are co-prime then $\mathsf{gcd}(a, b) = 1$ and therefore $1 = sa + tb$ for two integer constants s and t. If $1 = sa + tb$ then $sa \equiv 1 \pmod{b}$ and $tb \equiv 1 \pmod{a}$. If $sa \equiv 1 \pmod{b}$ then s is said to be a *multiplicative inverse* of a modulo b. If the modulus b is unambiguous then a^{-1} can be used to denote the multiplicative inverse of a modulo b.

Congruence. If a, b and n are integers then define $a \equiv b \pmod{n}$ if and only if (a mod n) = (b mod n). If $a \equiv b \pmod{n}$ then a is said to be *congruent* to b modulo n.

Euler's Theorem. If a is co-prime to n then $a^{\phi(n)} \equiv 1 \pmod{n}$: for every $u \in \mathbb{U}_n$, au is co-prime to n since both a and u are co-prime to n. If $\{u_1, u_2\} \subseteq \mathbb{U}_n$ and $au_1 \equiv au_2 \pmod{n}$ then $n|a(u_1 - u_2)$ and therefore $n|(u_1 - u_2)$ since n and a are co-prime. However, $n|(u_1 - u_2)$ is impossible since $0 < u_1 < n$ and $0 < u_2 < n$, and therefore any $\{u_1, u_2\} \subseteq$

$\mathbb{U}_n$ must have $au_1 \not\equiv au_2 \pmod{n}$. Consequently, $\{(au \bmod n) \mid u \in \mathbb{U}_n\} = \mathbb{U}_n$ and $\prod_{u\in\mathbb{U}_n}(au) \equiv a^{\phi(n)} \prod_{u\in\mathbb{U}_a}(u) \equiv \prod_{u\in\mathbb{U}_a}(u) \pmod{n}$. Furthermore, since each $u \in \mathbb{U}_a$ is co-prime to n, $\prod_{u\in\mathbb{U}_a}(u)$ must also be co-prime to n and therefore $a^{\phi(n)} \equiv 1 \pmod{n}$.

Chinese Remainder Theorem. If m and n are co-prime then for any pair of integers a, b there exists a unique x modulo mn such that $x \equiv a \pmod{m}$ and $x \equiv b \pmod{n}$: since $\mathsf{gcd}(m, n) = 1$ there must exist unique integers s, t such that $sm + tn = 1$. If $sm + tn = 1$ then $tn \equiv 1 \pmod{m}$ and $sm \equiv 1 \pmod{n}$. Let $x = bsm + atn$. Since $x \equiv ant \pmod{m}$ and $tn \equiv 1 \pmod{m}$ have $x \equiv a \pmod{m}$. A similar argument applies for $x \equiv b \pmod{n}$. x is unique modulo mn since any other x' such that $x' \equiv a \pmod{m}$ and $x' \equiv b \pmod{n}$ must have $x - x' \equiv 0 \pmod{m}$ and $x - x' \equiv 0 \pmod{n}$. Hence $x - x' \equiv 0 \pmod{mn}$ and x and x' are indistinguishable modulo mn.

ϕ is multiplicative If m and n are co-prime integers then $\phi(mn) = \phi(m)\phi(n)$: for each $(a, b) \in \mathbb{U}_m \times \mathbb{U}_n$ there exists a unique x such that $x \equiv a \pmod{m}$ and $x \equiv b \pmod{n}$. Since a is co-prime to m and b is co-prime to n, x must be co-prime to both m and n. Hence x is co-prime to mn. Furthermore, x is unique modulo mn and no other pair $(a', b') \in \mathbb{U}_m \times \mathbb{U}_n - \{(a, b)\}$ can identify the same x. Consequently, $\mathbb{U}_m \times \mathbb{U}_n$ and $\mathbb{U}_{mn}$ are in one-to-one correspondence and $|\mathbb{U}_m| \cdot |\mathbb{U}_n| = |\mathbb{U}_{mn}|$ as required.

Further information on modular arithmetic and primality can be found in Giblin [41].

7.2.2 RSA Public-Key Cryptography

Let $N = pq$ be the product of two prime numbers p and q. Pick an integer $d \in \mathbb{U}_{\phi(N)}$ and let $e \in \mathbb{Z}_{\phi(N)}$ denote the multiplicative inverse of d modulo $\phi(N)$. Since $ed \equiv 1 \pmod{\phi(N)}$ there must exist an integer constant k such that $ed = k\phi(N) + 1$.

Let $0 \leq M < N$ represent a message. To encode M compute $E(M) \overset{\text{def}}{=} M^e \bmod N$. To decode M compute $D(M) \overset{\text{def}}{=} M^d \bmod N$. $D(E(M)) = M$ can be shown as follows: $D(E(M)) \equiv M^{ed} \equiv M^{k\phi(N)+1} \equiv (M^{\phi(N)})^k M \equiv (M^{\phi(p)\phi(q)})^k M$. If p does not divide M then since p is prime $\mathsf{gcd}(p, M) = 1$ and $M^{\phi(p)} \equiv 1 \pmod{p}$. Consequently, $(M^{\phi(N)})^k M \equiv M \pmod{p}$. Conversely, if p divides M then $(M^{\phi(N)})^k M \equiv M \equiv 0 \pmod{p}$ is trivially true, and therefore $(M^{\phi(N)})^k M \equiv M \pmod{p}$ holds for all M. A similar argument applies for $(M^{\phi(N)})^k M \equiv M \pmod{q}$. If $(M^{\phi(N)})^k M \equiv M \pmod{p}$ and $(M^{\phi(N)})^k M \equiv M \pmod{q}$ then by the Chinese Remainder Theorem, $E(D(M)) \equiv M^{ed} \equiv (M^{\phi(N)})^k M \equiv M \pmod{pq}$ as required.

The pairs (e, N) and (d, N) denote public and private keys respectively. Decoding and encoding are symmetric and $E(D(M))$ can therefore be used instead of $D(E(M))$ as a means of digitally *signing* a message. RSA public-key cryptography is due to Rivest, Shamir and Adleman [91].

7.2.3 Fixed Precision Integers

Let $X = \langle x_n \dots x_0 \rangle_b$ denote an $n+1$ digit integer in base b. Require that each $x_i \in \{0 \dots b-1\}$ and that $X = b^n x_n + b^{n-1} x_{n-1} + \cdots + b x_1 + x_0$. If $X = \langle x_n \dots x_0 \rangle_b$ then the *base* of X is b and the *precision* of X is $n+1$.

7.3 Montgomery Exponentiation

The underlying operation performed by the RSA encoding and decoding functions computes the remainder $(M^E \bmod N)$ when M^E is divided by N for a particular M and E. Computation of $(M^E \bmod N)$ is referred to as *modular exponentiation* and may be efficiently computed as a sequence of at most $2\log_2(E)$ modular multiplications, see procedure **bin_mod_exp** in Algorithm 7.1. Procedure **bin_mod_exp** is referred to as binary modular exponentiation and belongs to the *repeated square-and-multiply* category of general purpose exponentiation algorithms [63]. Binary exponentiation can be shown to be correct by observing that if $E = \langle e_{n-1} \dots e_0 \rangle_2$ then $e = e_0 + 2(e_1 + 2(e_2 + \cdots + 2(e_{n-1}) \cdots))$.

```
FUNCTION bin_mod_exp ($N : \mathbb{N}$, $M : \mathbb{Z}_N$, $E : \mathbb{N}$) : $\mathbb{Z}_N$
    LET $E = \langle e_t \dots e_0 \rangle_2$
    $A := 1$
    FOR $i := t$ down to 0 step 1 DO
5:      $A := (A \cdot A \bmod N)$
6:      IF $e_i = 1$ THEN $A := (A \cdot M \bmod N)$ END IF
    END FOR
    RETURN $A$
END FUNCTION
```

Algorithm 7.1: Binary modular exponentiation.

7.3.1 Montgomery Multiplication

The efficiency of procedure **bin_mod_exp** is dependent on the efficiency with which the modular multiplications in Lines 5 and 6 are performed. *Montgomery multiplication* [70] is an algorithm for efficient computation of modular products in which no explicit division of the result by the modulus N is required.

To perform a Montgomery multiplication one must first pick an $R > N$ such that $\mathsf{gcd}(R, N) = 1$. For any $X \in \mathbb{Z}_N$ and $Y \in \mathbb{Z}_N$ Montgomery multiplication computes the product $(XYR^{-1} \bmod N)$ where $R^{-1} \in \mathbb{Z}_N$ denotes the multiplicative inverse of R modulo N. Montgomery multiplication does not avoid trial division altogether it merely translates the divisor from N to R at the cost of augmenting the product with the factor R^{-1}. Montgomery multiplication is useful since if R is a power of two then division by R is trivial when X, Y, N are represented in binary. Furthermore, in the context of RSA cryptography, $R = 2^k$ is always possible for a large enough k since N is always odd and therefore $\mathsf{gcd}(2^k, N) = 1$ is necessarily true.

```
FUNCTION mmult ($b : \mathbb{N}$, $N : \mathbb{N}$, $X : \mathbb{Z}_N$, $Y : \mathbb{Z}_N$) : $\mathbb{Z}_N$
   ASSERT $\mathsf{gcd}(b, N) = 1$
   LET $N = \langle N_{n-1} \dots N_0 \rangle_b$ and $X = \langle x_{n-1} \dots x_0 \rangle_b$ and $Y = \langle y_{n-1} \dots y_0 \rangle_b$
   $A = \langle a_n \dots a_0 \rangle_b := 0$
   FOR $i := 0$ up to $n - 1$ step 1 DO
      $Q := ((a_0 + x_i y_0)(b - N_0)^{-1} \bmod b)$
      $A := (A + x_i Y + QN)/b$
8: END FOR
   IF $A \geq N$ THEN $A := A - N$ END IF.
   RETURN $A$
END FUNCTION
```

Algorithm 7.2: Montgomery Multiplication.

Procedure $\mathbf{mmult}(b, N, X, Y)$ in Algorithm 7.2 outlines the Montgomery multiplication of two n-digit numbers X and Y modulo N in base b. **mmult** can be shown to return $A = (XYR^{-1} \bmod N)$ where $R = b^n$ and where $\mathsf{gcd}(R, N) = \mathsf{gcd}(b, N) = 1$ as follows:

Each successive computation $A := (A + x_i y + QN)/b$ is exact: $Q \equiv (a_0 + x_i y_0)(b - N_0)^{-1} \equiv (A + x_i Y)(-N)^{-1} \pmod b$ since only the least significant digits of A, Y, and N contribute to the value of $(A + x_i Y)(-N)^{-1}$ modulo b. If $Q \equiv (A + x_i Y)(-N)^{-1} \pmod b$ then $QN \equiv -(A + x_i Y) \pmod b$ and therefore $A + x_i y + QN \equiv 0 \pmod b$.

The value of A is bounded by $N + Y < 2N$ and can therefore be stored with precision $n + 1$: let A_i denote the i^{th} value of A. If $A_i < N + Y$ then from Line 8, $A_{i+1} < (A_i + (b-1)Y + (b-1)N)/b < ((N+Y) + (b-1)(Y+N))/b < b(Y+N)/b < N+Y$,

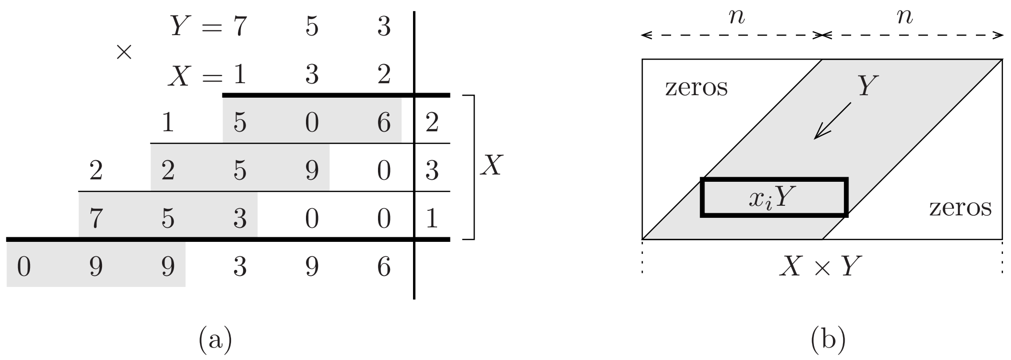


Figure 7.1: Conventional paper-based long multiplication.

and since $A_0 = 0$, by induction $A < N + Y$. Furthermore, if $b^i A_i \equiv Y\langle x_{i-1} \ldots x_0\rangle_b \pmod{N}$ then:

$$\begin{aligned}
bA_{i+1} &\equiv A_i + x_iY \pmod{N} && [\text{ Line 8 }]\\
b^i bA_{i+1} &\equiv b^iA_i + b^ix_iY \pmod{N} &&\\
b^{i+1}A_{i+1} &\equiv Y\langle x_{i-1}\ldots x_0\rangle_b + b^ix_iY \pmod{N} && [\text{ Inductive hypothesis }]\\
&\equiv Y\langle x_i\ldots x_0\rangle_b \pmod{N} &&
\end{aligned}$$

and therefore by induction $b^iA_i \equiv Y\langle x_{i-1}\ldots x_0\rangle_b \pmod{N}$. Consequently,

$$\begin{aligned}
b^nA_n &\equiv Y\langle x_{n-1}\ldots x_0\rangle_b \pmod{N} &&\\
RA_n &\equiv XY \pmod{N} && [\ R = b^n\ ]\\
R^{-1}RA_n &\equiv R^{-1}XY \pmod{N} &&\\
A_n &\equiv XYR^{-1} \pmod{N} && [\ RR^{-1} \equiv 1 \pmod{N}\ ]
\end{aligned}$$

and since $A < 2N$ at most one subtraction is necessary, see Line 10, to ensure that $A = (XYR^{-1} \bmod N)$ as required.

In conventional paper-based long multiplication, the product $X \times Y$ is computed as a sequence of additions of increasing significance, see Figure 7.1(a). Each row of a long multiplication table for two n-digit numbers X and Y identifies a multiple x_iY of the multiplicand Y, computed using a single digit x_i of the multiplier X, see Figure 7.1(b). Montgomery multiplication can be likened to a form of long multiplication where a special multiple of N is added at each row. The particular multiples of N that are chosen result in a product where the most significant $n+1$ digits denote $(XYR^{-1} \bmod N)$ or $(XYR^{-1} \bmod N)+N$ and where the least significant n digits are all zero, see Figure 7.2.

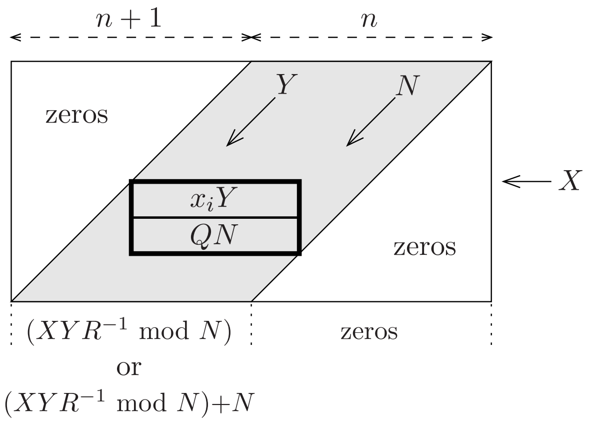


Figure 7.2: Visualisation of Montgomery multiplication.

7.3.2 Binary Montgomery Exponentiation

The factor R^{-1} cannot be removed from a Montgomery product using Montgomery multiplication alone, and Montgomery multiplication is therefore of no benefit when applied to a single modular multiplication. However, if $Z = XY$ then the identity $(XR)(YR)R^{-1} \equiv (XY)R \equiv ZR \pmod{N}$ implies that Montgomery multiplication is well-defined over the integers $\{iR \mid i \in \mathbb{Z}_N\}$, referred to as *Montgomery residues*. Furthermore since R and N are co-prime, if $i \neq j$ then $iR \not\equiv jR \pmod{N}$, and hence $\{iR \mid i \in \mathbb{Z}_N\} = \mathbb{Z}_N$. Montgomery multiplication can therefore be equated to conventional modular multiplication except in a world where the integers modulo N are ordered $0, (1R \bmod N), (2R \bmod N), \ldots, ((N-1)R \bmod N)$ rather than $0, 1, 2, \ldots, N-1$.

If a long sequence of modular multiplications is to be performed then the time required to convert between conventional residues and Montgomery residues becomes insignificant when compared to the time saved by permitting Montgomery multiplications to be performed. Binary Montgomery exponentiation is a form of binary modular exponentiation in which Montgomery multiplications are employed, see procedure **mexp** in Algorithm 7.3. Procedure **mexp** differs from procedure **bin_mod_exp** in that the initial value of A is set to the Montgomery residue for 1, $(R \bmod N)$, and each multiplication by M is performed as a multiplication by the Montgomery residue for M, $M_R = (MR \bmod N)$. The extra multiplication in Line 12, is used to convert the final value of A back from the Montgomery residue $(M^ER \bmod N)$ into the conventional residue $(M^E \bmod N)$.

```
FUNCTION mexp ($N : \mathbb{N}$, $M : \mathbb{Z}_N$, $E : \mathbb{N}$) : $\mathbb{Z}_N$
  ASSERT $N$ is odd
  LET $N = \langle N_n \ldots N_0 \rangle_2$ and $M = \langle m_n \ldots m_0 \rangle_2$
  LET $E = \langle e_t \ldots e_0 \rangle_2$
  LET $R = 2^{n+1}$
  $M_R := (MR \bmod N)$                              [ compute by conventional means ]
  $A := (R \bmod N)$
  FOR $i := t$ down to 0 step 1 DO
    $A :=$ mmult(2, $N$, $A$, $A$)
    IF $e_i = 1$ THEN $A :=$ mmult(2, $N$, $A$, $M_R$) END IF
  END FOR
12: $A :=$ mmult(2, $N$, $A$, 1)
  RETURN $A$
END FUNCTION
```

Algorithm 7.3: Binary Montgomery exponentiation.

7.4 Montgomery Multiplication in Hardware

Montgomery multiplication is ideally suited to digital methods since if the base $b = 2$ then division by b equates merely to a bit-shift of 1, see procedure **hw_mmult** in Algorithm 7.4 [37]. If $b = 2$ then to ensure that every division by 2 is exact, **hw_mmult** must guarantee that each A_i is even before dividing by 2 in Line 8. Since any N with $\mathsf{gcd}(N, 2) = 1$ must be odd, an odd A can always be made even by computing $A := A + N$ without changing the value of A modulo N, see Line 7.

Procedure **hw_mmult** can be implemented in hardware as an array of single-bit full-adders, see Figure 7.4. The physical structure of this array equates directly to the visualisation depicted in Figure 7.2: each row consists of up to two additions, one for the product x_iY and the other for the modulus N. Each of these two additions is constructed from n full-adders, see Figure 7.3, and carry rippling is computed concurrently with row traversal by passing each c_{out} both down and to the right. An extra full-adder, see the shaded boxes in Figure 7.4, is required for each N addition since intermediate values of A are bounded by $2N$ not N.

Figure 7.4 can be related to conventional carry-sum hardware multiplication [113] but where there are $2n$ rows rather than n: each horizontal cross-section identifies a set of carry-sum pairs to be passed to the next row, and an explicit carry ripple is only required during the final additions in the last row of the array.

```
FUNCTION hw_mmult ($N : \mathbb{N}$, $X : \mathbb{Z}_N$, $Y : \mathbb{Z}_N$) : $\mathbb{Z}_N$
   ASSERT $N$ is odd
   LET $N = \langle N_{n-1} \dots N_0 \rangle_2$ and $X = \langle x_{n-1} \dots x_0 \rangle_2$ and $Y = \langle y_{n-1} \dots y_0 \rangle_2$
   $A = \langle a_n \dots a_0 \rangle_2 := 0$
   FOR $i := 0$ up to $n - 1$ step 1 DO
      IF $x_i = 1$ THEN $A := A + Y$ END IF
7:    IF $a_0 = 1$ THEN $A := A + N$ END IF
8:    $A := \langle 0 a_n \dots a_1 \rangle_2$                    [ Note that $a_0 = 0$ ]
   END FOR
   IF $A \geq N$ THEN $A := A - N$ END IF.
   RETURN $A$
END FUNCTION
```

Algorithm 7.4: Montgomery Multiplication in Hardware.

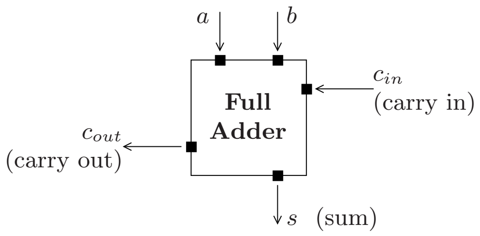


Figure 7.3: Full-adder cell description.

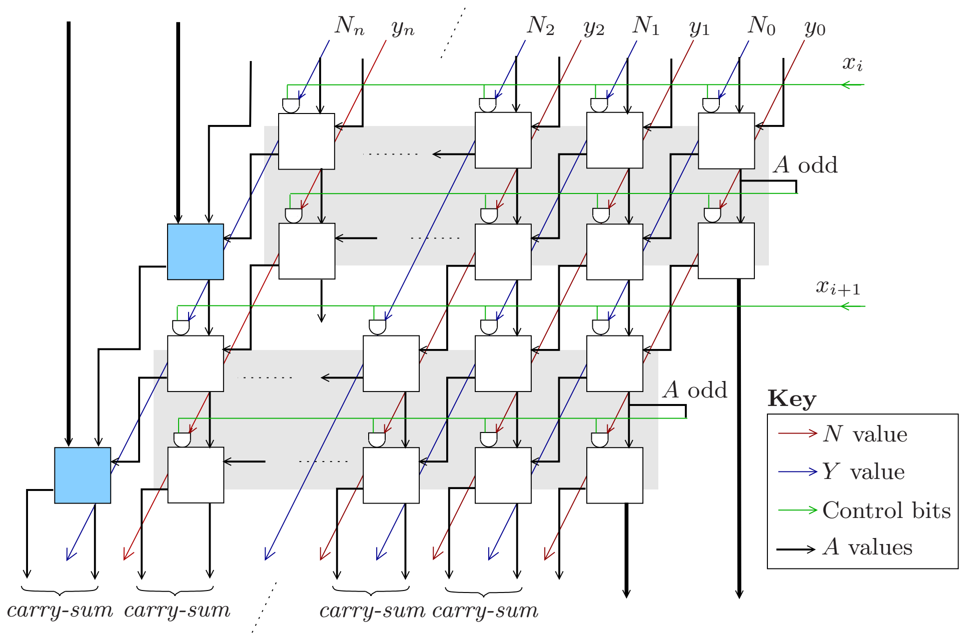


Figure 7.4: Carry-Sum Montgomery multiplication.

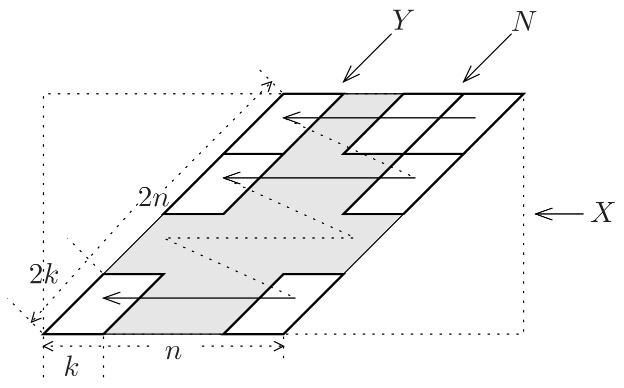


Figure 7.5: Systolic Montgomery multiplication window traversal.

7.4.1 Window-Based Traversal

If the precision, n, becomes too large then the area cost of implementing a complete $n \times n$ carry-sum can become unacceptable. Computation of $(XYR^{-1} \bmod N)$ using a smaller, $k \times k$ array can be achieved by performing a systematic traversal of the complete $n \times n$ array using a $k \times k$ *window*, see Figure 7.5. Traversal begins with the least significant bits of both X and Y and computes n/k iterations of the FOR loop in **hw_mmult** for each complete traversal of a row in the $n \times n$ array. With each movement of the window position across a row, k new bits of the current value, A_i, of A are introduced and k new bits of the next value, A_{i+k}, of A are computed. The $k + 1$ most-significant bits are computed as k carry-sum pairs which are subsequently fed back into the top of the next window position as the initial carry-sum value for A_i, see Figure 7.6. This feedback is necessary since each window position computes *half* of the additions required for $2k$ bits of A_{i+k} and not *all* of the additions required by k bits of A_{i+k}. No carry rippling is required until the last row of the last window position since the carry-sum feedback path incorporates any carry ripple between one window position and the next.

7.5 Montgomery Exponentiation in Hardware

The purpose of this section is to describe MOD_EXP an implementation of the binary Montgomery exponentiation procedure **mexp** in which each Montgomery multiplication is computed using a delay insensitive asynchronous dual-rail datapath with with data-independent power-consumption. MOD_EXP can perform either 1024 or 2048 bit modular exponentiations, and uses a 32×32 bit window-based traversal method for each Montgomery multiplication.

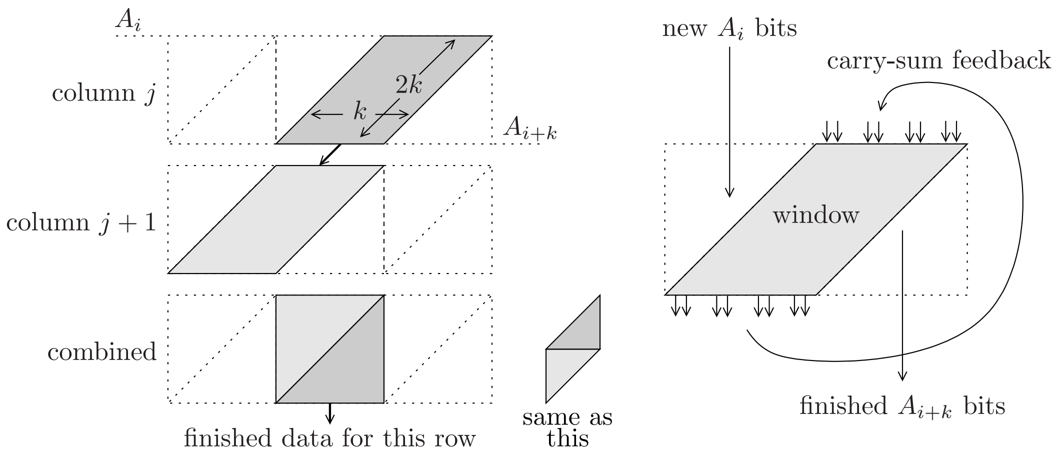


Figure 7.6: Systolic Montgomery multiplication window stepping.

MOD_EXP has been fabricated as a 1.4×1.3mm block on a 0.18μm CMOS standard-cell process, and uses conventional single-port memories for its main register banks. Post-production tests indicate that MOD_EXP is fully functional, taking 0.15 seconds to perform a 1024 bit modular exponentiation and 1.1 seconds to perform a 2048 bit modular exponentiation. MOD_EXP is integrated into a larger test chip containing a number of asynchronous XAP microcontrollers [17], each of which can control MOD_EXP through a synchronous memory-mapped IO interface.

7.5.1 Datapath Handshake Protocols

The use of a return-to-zero protocol in a delay-insensitive encoding scheme significantly reduces the area and complexity of individual logic elements, however it requires that computation consist of two phases: an *assert* phase in which data values are computed and a *clear* phase in which previous data values are removed. In the context of window-based traversal a return-to-zero protocol doubles the cycle-time of each window position. This doubling can be avoided by pipelining the carry-sum array into two stages so that asserting and clearing happen concurrently and in alternation as shown in Figure 7.7. Since the area overhead of a single pipeline stage is substantially less than the area overhead of using a non-return-to-zero protocol, the carry-sum array in MOD_EXP was therefore implemented in this way.

7.5.2 MOD_EXP Architecture

An overview of MOD_EXP operation is outlined in Algorithm 7.5, and a summary of its top-level architecture shown in Figure 7.8. MOD_EXP contains four data registers

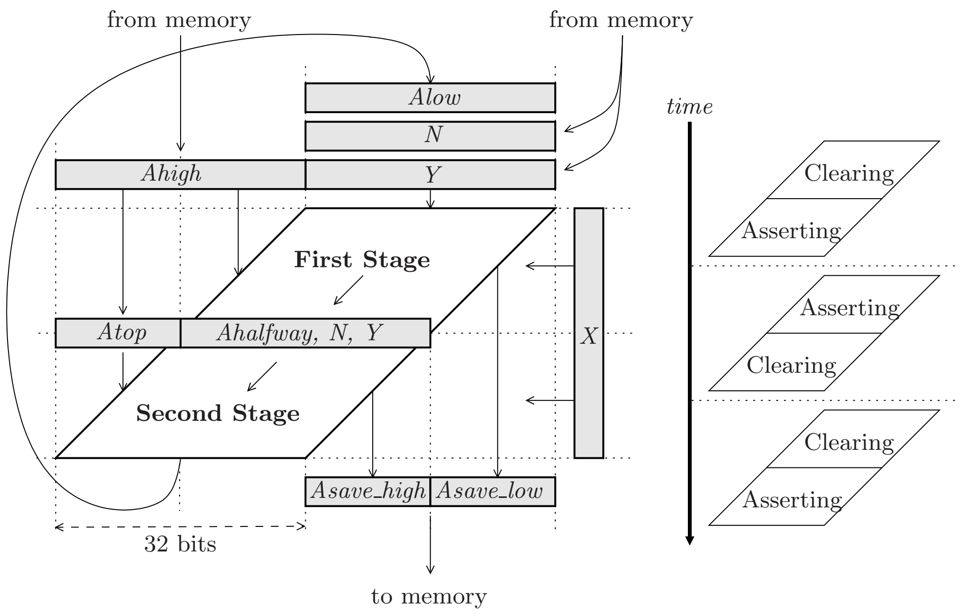


Figure 7.7: Pipelined Montgomery multiplication.

N, $(R^2 \bmod N)$, M_R, E and two work registers P, Q used to store both intermediate values and the final result. A global flag *long* selects either 1024 or 2048 bit operation, and a global flag *inP* is used to determine which work register the result of each Montgomery multiplication lies in. MOD_EXP makes use of a special value, $(R^2 \bmod N)$, which must be externally provided, to compute $(MR \bmod N)$ as the Montgomery product $(M)(R^2 \bmod N)(R^{-1}) \equiv MR \pmod{N}$. The external computation of $(R^2 \bmod N)$ is preferable to the external computation of $(MR \bmod N)$ since $(R^2 \bmod N)$ is constant for any given N and therefore need only be computed once for multiple exponentiations with the same modulus.

The correct operation of MOD_EXP requires two work registers rather than one in order to ensure that the work register used by MMULT is never the same as either the multiplier X or the multiplicand Y. The IF statements in Lines 13, 16, and 20 are used to set two pointers A and B such that A always points to the conflict-free work register. The initial value $\langle 1\overline{N[n-1]}\ldots\overline{N[1]}1\rangle_2$ assigned to P in Line 10 can be shown to equal $(R \bmod N)$ by observing that $\langle 0\overline{N[n-1]}\ldots\overline{N[1]}1\rangle_2 + \langle 0N[n-1]\ldots N[1]1\rangle_2 = 2^n = R$, and that since N is odd, $N[0] = 1$ is always true. The same value $\langle 1\overline{N[n-1]}\ldots\overline{N[1]}1\rangle_2$ is used a second time in Line 31 to identify the value $-N$ as the two's-complement of N. If the addition of A and $\langle\overline{N[n-1]}\ldots\overline{N[1]}1\rangle_2$ results in a carry overflow, $c = 1$, then $A - N$ is positive and $(XYR^{-1} \bmod N)$ lies in B. If $B - N$ is negative then $(XYR^{-1} \bmod N)$ lies in A

```
FUNCTION **mod_exp**
   REGISTER $long$, $inP$ : bit
   REGISTER $N$, $(R^2 \bmod N)$, $(MR \bmod N)$, $E$ : bit[2048]
   REGISTER $P$, $Q$ : bit[2049]
   ASSERT $M$ value is loaded into register $P$
   IF $long$ THEN n := 2048 ELSE n := 1024 END IF
   $inP := 1$
   **MMULT**$((R^2 \bmod N), P)$
   IF $inP$ THEN $(MR \bmod N) := P$ ELSE $(MR \bmod N) := Q$ END IF
   $P := \langle 0\overline{N[n-1]}\ldots\overline{N[1]}1\rangle_2$
   $inP := 1$
   FOR $i := 2047$ down to $2048 - n$ step 1 DO
      IF $inP$ THEN LET $B = P$, $A = Q$ ELSE LET $B = Q$, $A = P$ END IF
      **MMULT**$(B, B)$
      IF $E[i]$ THEN
         IF $inP$ THEN LET $B = P$, $A = Q$ ELSE LET $B = Q$, $A = P$ END IF
         **MMULT**$(B, (MR \bmod N))$
      END IF
   END FOR
   IF $inP$ THEN LET $B = P$, $A = Q$ ELSE LET $B = Q$, $A = P$ END IF
   **MMULT**$(B, 1)$
if $inP = 1$ then the $(M^E \bmod N)$ is in $P$ otherwise it is in $Q$
   SUB FUNCTION **MMULT** $(X, Y)$
      ASSERT $A$ does not point to $X$ or $Y$
      $A := 0$
      FOR $j := 0$ up to $n - 1$ step 1 DO
         IF $X[j]$ THEN A := $A + Y$ END IF
         IF $A[0]$ THEN A := $A + N$ END IF
         $A := \langle 0A[n]\ldots A[1]\rangle_2$
      END FOR
      $\langle cB[n]\ldots B[0]\rangle_2 := A + \langle 1\overline{N[n-1]}\ldots\overline{N[1]}1\rangle_2$
      IF $\neg c$ THEN $inP := \neg inP$ END IF
   END SUB FUNCTION
END FUNCTION
```

Algorithm 7.5: Functional operation of mod_exp.

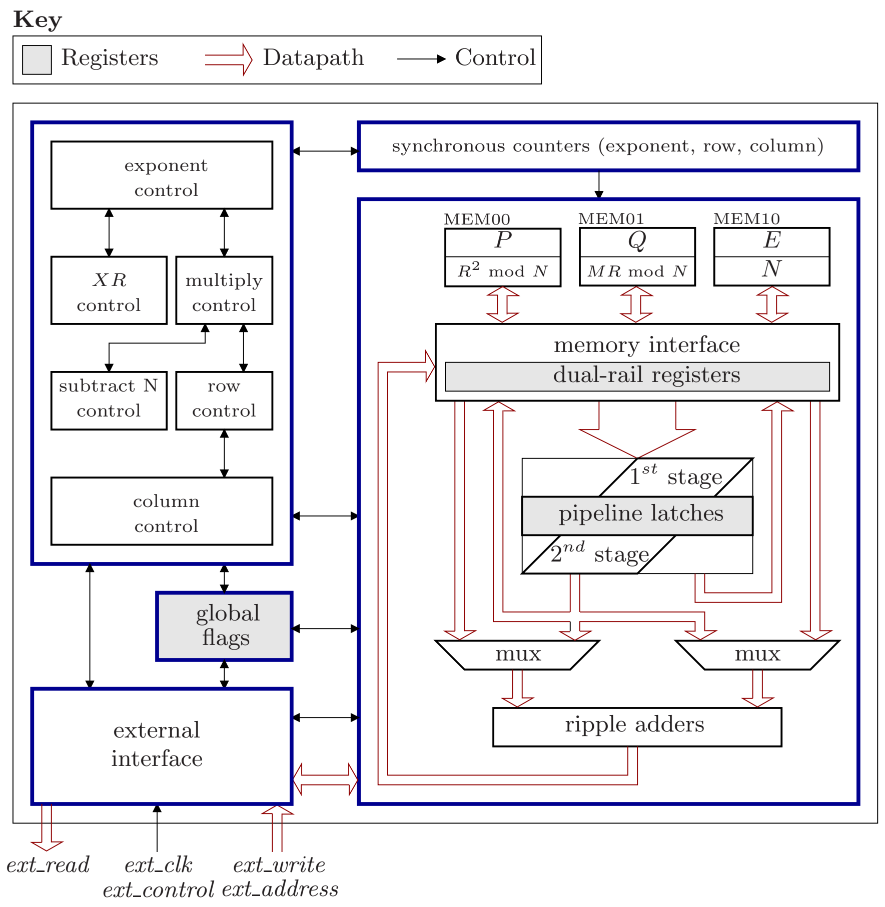


Figure 7.8: mod_exp top-level architecture.

and *inP* must be flipped so that it points to the correct result register.

With each new window position, 32 new bits of A_i must be obtained from memory as *Ahigh* and 32 completed bits of A_{i+32} must be written back out to memory from *Asave_high*-*Asave_low*. Furthermore, 32 new bits of both Y and N must be read from memory. New X bits need only be loaded each time the array is moved into the start position for a new row. The particular distribution of registers across the three memories MEM00, MEM01, and MEM10, ensures that no Montgomery multiplication views more than one of A, X, or Y as living in the same memory. This assurance permits the four required memory accesses per window position to be paired into two conflict-free single-port access cycles as follows:

1. (Read from register A, Read from register N)

2. (Write to register A, Read from register Y)

Each of these access cycles is linked respectively to the assert-clear cycle a pipeline stage in the carry-sum array. This mode of operation is safe provided, as in the case of MOD_EXP, the cycle-time for each pipeline stage is large with respect to the minimum memory cycle time. A similar linking is also applied to the counters used for indexing column, row, and exponent bit positions. This linking is safe provided, as in the case of MOD_EXP, the complete array cycle time is large with respect to the minimum counter cycle-times.

MOD_EXP is controlled through a synchronous external interface which permits all of its registers and flags to be directly read and written. A special pair of flags, *go_exp* and *done_exp*, are used to activate and detect completion of MOD_EXP according to a four-phase handshake protocol. *go_exp* and *done_exp* are not shown in Algorithm 7.5. An outline of the final place-and-routed MOD_EXP design is shown in Figure 7.9.

7.5.3 Other Architectures

Research into methods of Montgomery exponentiation is active both in industry and academia, and a comprehensive treatment of Montgomery exponentiation is beyond the scope of this thesis. The window-based traversal method described here was chosen for its simplicity, and for the ease with which it could be applied to a *scalable* asynchronous implementation. Window-based traversal methods can also be applied to windows of size $k \times l$ where $k \neq l$, however if $k = l$ then all word sizes are constant and accesses to register memories for Y and N are minimised.

If communication between every array position in a Montgomery carry-sum array is pipelined, then a *systolic* implementation of is achieved [109]. Systolic Montgomery multipliers can process up to $2n$ multiplications simultaneously at throughputs bounded only by the cycle time of a single full-adder: the longest pipeline path is between the top right-hand and bottom left-hand corner of the matrix, and a new result may be produced by every full-adder every cycle. Systolic Montgomery multiplication generalises to any base, and when the height of the array is one, reduces to a conventional linear pipeline structure that is both compact and efficient [110]. Recent publications [44, 108] have also demonstrated that the speculative subtraction of N at the end of each Montgomery multiplication can be avoided by adding one bit to the size of each work register and iterating over $n + 1$ steps rather than n.

7.5.4 MOD_EXP Performance and Resistance to attack

MOD_EXP is not a high-performance architecture, and is slow when compared to commercial full-custom cryptographic accelerators designed for use in e-commerce applications.

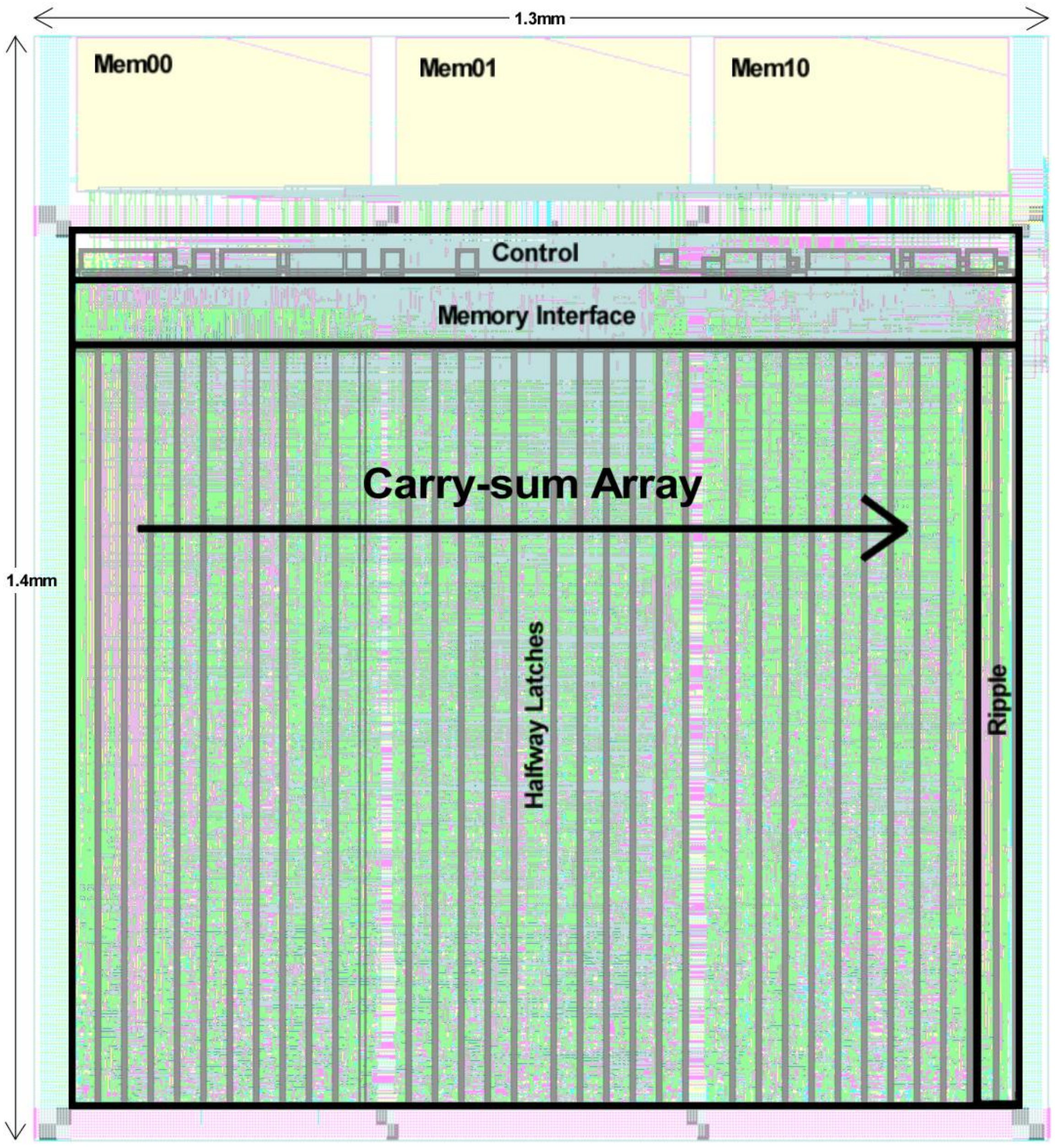


Figure 7.9: Taped-out mod_exp design.

For example, the PCCC-ISES from Securealink [88] takes 0.003s per 1024 bit modular exponentiation, and the Intel Itanium is predicted to advance this to 0.001s [47]. However, when compared to other embedded public-key encoder-decoders for use in smartcards and smartcard readers the performance of MOD_EXP is competitive: The PCC810 smartcard reader from Securealink [89] has identical performance to MOD_EXP, and the PrivateCard smartcard from Algorithmic Research [2] takes 0.6s per 1024 bit modular exponentiation. Furthermore, in the context of resistance to attack, MOD_EXP has two potential benefits over competing synchronous implementations:

Data-independent power consumption [52]
The dual-rail datapath in MOD_EXP consumes identical power to compute both a logic one and a logic zero. Furthermore the speculative subtraction at the end of each Montgomery multiplication is always computed, making MOD_EXP power consumption identical across multiplications.

Data-independent timing [51]
The execution time of MOD_EXP depends only on the number of ones in the exponent. Since the execution time of individual Montgomery multiplications is data-independent and since speculative subtractions are always performed [111], the location of these ones cannot be determined using timing attacks alone.

7.6 Application of Veraci to MOD_EXP

The purpose of this section is to demonstrate how Veraci was used to facilitate the design of MOD_EXP. This demonstration consists of three example Veraci programs, each of which was used to verify a particular building block from inside of MOD_EXP. MOD_EXP serves as an interesting case study for Veraci because it contains a mixture of both bundled-data and delay-insensitive communication protocols. Furthermore, some of the building blocks in MOD_EXP also employ relative timing assumptions or fundamental-mode environmental constraints.

7.6.1 Symmetric Dual-Rail Addition

A key motivation behind the use of a delay-insensitive data encoding scheme in MOD_EXP is the ability achieve a data-independent power signature [52]. This ability stems from the design of delay-insensitive gates that are *symmetric*. A symmetric delay-insensitive

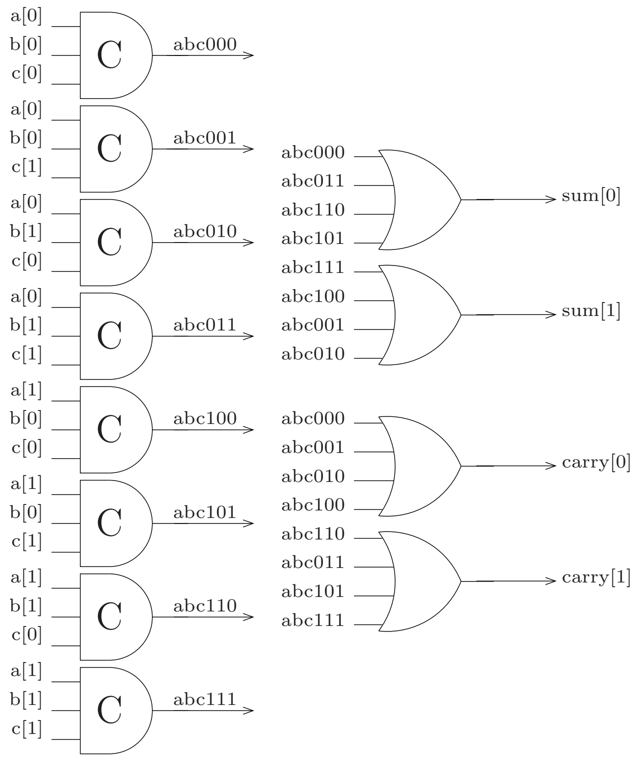


Figure 7.10: Symmetric dual-rail adder circuit.

gate is a gate that consumes the same amount of power to compute output values for all possible input value combinations.

The power consumed by MOD_EXP is dominated by the array of 32×64 dual-rail full-adders used to compute a Montgomery product. The design of dual-rail full-adders are considered at length by Martin [61], however his designs assume a full-custom design flow and are not symmetric. The dual-rail full-adder design presented here was motivated by a desire to achieve symmetric power consumption using gates available in a conventional standard-cell library.

A dual-rail full-adder is a combinational function with three dual-rail inputs a, b, c and two dual-rail outputs *sum*, *carry* such that *carry* $\overset{\text{def}}{=} (a \wedge b) \vee (c \wedge (a \vee b))$ and *sum* $\overset{\text{def}}{=} a \oplus b \oplus c$. A symmetric delay-insensitive dual-rail full-adder can be constructed directly from a sum-of-products expansion for *sum* and *carry* as shown in Figure 7.10.

Although simple, the circuit in Figure 7.10 is both large and slow, particularly if the available standard-cell library cannot offer a compact implementation of a 3-input C-element. In the context of MOD_EXP, the circuit in Figure 7.10 can be significantly improved by

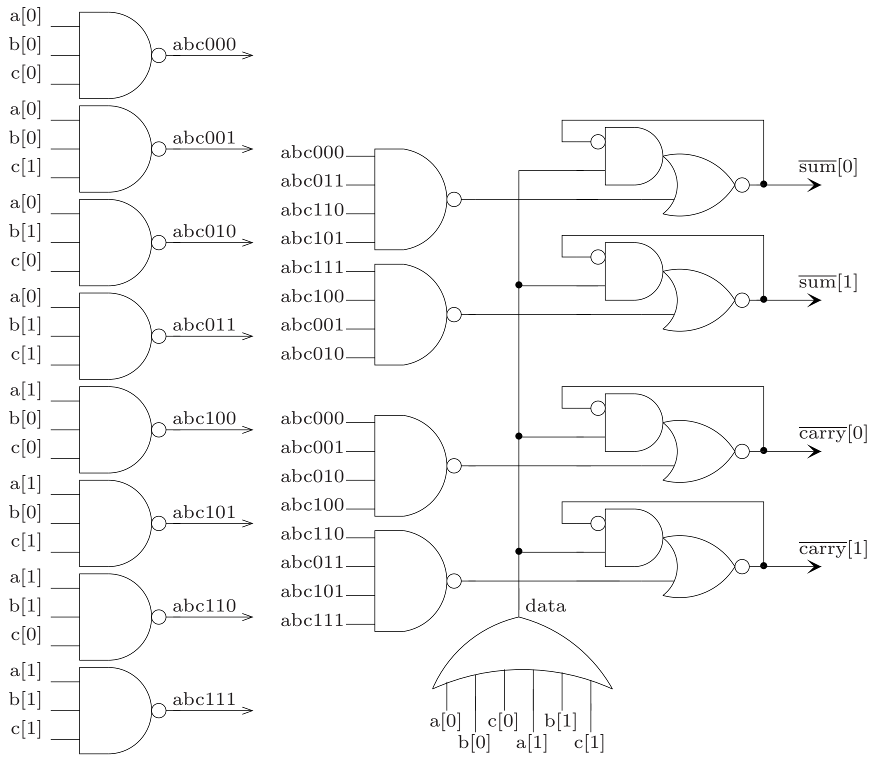


Figure 7.11: Optimised symmetric dual-rail adder circuit.

asserting that the data-clear cycle-time is sufficiently long for the C-elements to be replaced with NAND gates as shown in Figure 7.11. This assertion is comfortably realised by MOD_EXP since the entire carry-sum array is pipelined into only two stages.

The circuit in Figure 7.11 differs from the circuit shown in Figure 7.10 in that the dual-rail protocol on its outputs is inverted with respect to the dual-rail protocol on its inputs. The use of NAND gates as opposed to C-elements also requires that extra circuitry be included to ensure that each output does not clear before all the inputs have cleared. This extra circuitry equates to the use of half-latches on each output wire, and the use of a special *data* signal that clears these half-latches once all the inputs have cleared.

A Veraci program for the optimised dual-rail full-adder circuit is outlined in Figure 7.11. This Veraci program can be used to verify that the optimised full-adder only asserts and clears data-values on its outputs according to Seitz's rules, see Section 6.4. The Veraci-fragment `cut always [ IN_DATA- => data ]` is used to enforce the assumption that data-clear cycle-times exceed the time taken for *data* to be set: at the instant in time when the last of the three dual-rail inputs clears, *data* must be at level 1.

The circuit in Figure 7.11 has a mirror implementation in which a NOR-NOR expansion

```
module SI DRadd(s, carry, a, b, cin);
  output [1:0] s, carry;
  input  [1:0] a, b, c;
  ...
<<
  initial ~(a[0] | a[1] | b[1] | b[1] | c[1] | c[1])

  let A = dr_event(a[0],a[1])
  let B = dr_event(b[0],b[1])
  let C = dr_event(c[0],c[1])

  let CARRY = dr_event(carry[0],carry[1])
  let   SUM = dr_event(sum[0],sum[1])

  protocol [ (A || B || C) ; (CARRY || SUM) ]

  // large cycle-time assumption
  let IN_DATA = dr_data(a[0],a[1]) & dr_data(b[0],b[1]) & dr_data(c[0],c[1])
  cut always [ IN_DATA- => data ]
>>
endmodule
```

Figure 7.12: Veraci program for the symmetric dual-rail adder circuit.

is used rather than a NAND-NAND expansion. This mirror implementation takes an inverted dual-rail protocol on its inputs and generates outputs that conform to a non-inverted dual-rail protocol. The carry-sum array in MOD_EXP therefore alternates between NOR-NOR and NAND-NAND implementations from one row to the next in order to avoid converting between inverted and non-inverted protocols.

7.6.2 A-odd Detection

Each row of the 32×32 bit carry-sum array in MOD_EXP consists of up to two additions, one for the product x_iY and the other for the modulus N. Each addition of N is dependent on the least significant bit, a_0, of the running sum A, see Algorithm 7.4: if $a_0 = 1$ then A is odd and N must be added to A. Each time a new value for a_0 is computed this value is required by every full-adder in a complete 1024×1024 or 2048×2048 bit carry-sum row, see Figure 7.4. Since MOD_EXP traverses the complete carry-sum array using a 32×32 bit window, the value for a_0 must also be latched until traversal of an entire carry-sum row has completed.

In MOD_EXP, the delivery of a_0 across each row of the 32×32 bit array is implemented using a bundled data protocol controlled by the circuit **Aodd**, see Figure 7.14. The purpose of **Aodd** is to take the dual-rail value *a0*, denoting a_0, and to compute a pair of

```
module DI Aodd(data, ready, odd, a0, clear);
  output      data, ready, odd;
  input [1:0] a0;
  input       clear;
  ...
<<
  initial ~clear & ~a0[0] & ~a0[1]

  // desired value for odd when ready = 1
  let valid = a0[0] & ~odd | a0[1] & odd

  protocol [ dr_plus(a0[0],a0[1]);(ready+ & valid);dr_minus(a0[0],a0[1]);
             [ dr_plus(a0[0],a0[1]);dr_minus(a0[0],a0[1]) ];
             clear+;ready-;clear- ]

  // fundamental-mode operation
  cut always [excited => ~a0[0]* & ~a0[1]* & ~clear*]
>>
endmodule
```

Figure 7.13: Veraci program for the A-odd generation circuit.

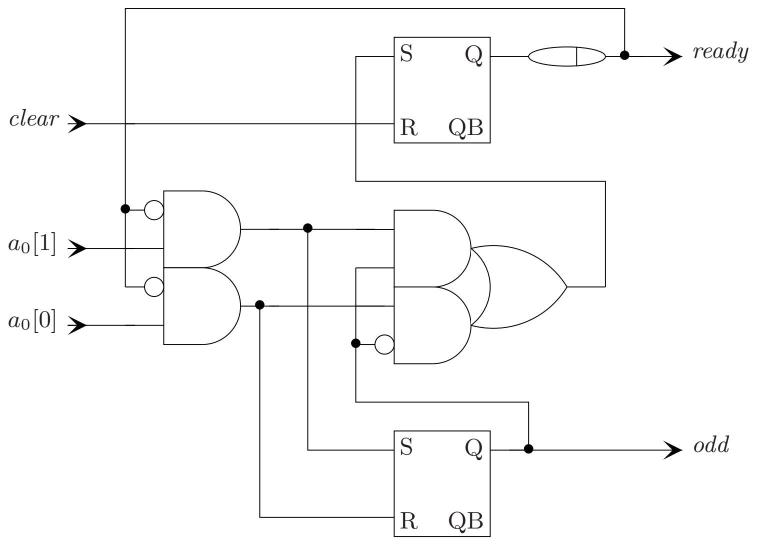


Figure 7.14: A-odd generation circuit.

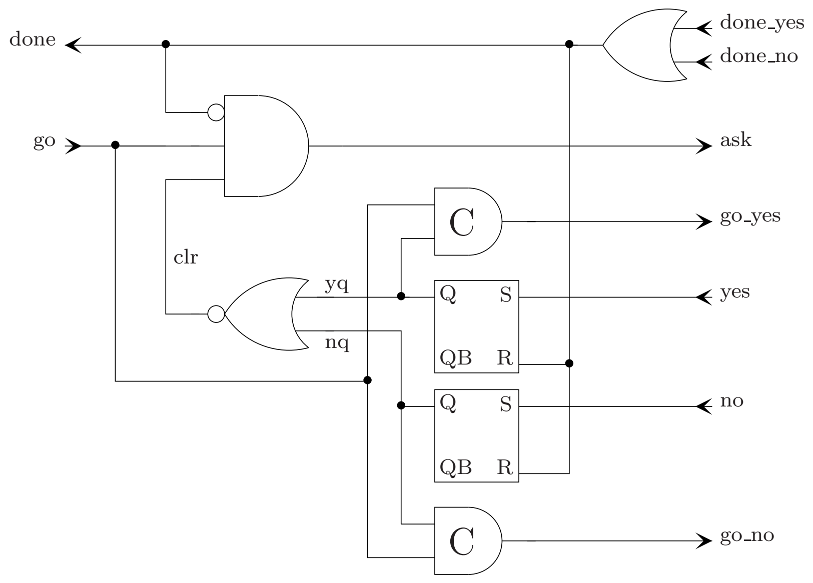


Figure 7.15: Ask-do control circuit.

values, (*ready*,*odd*) such that if *ready* = 1 then *odd* = a_0. Each time *ready* is set, the value of *odd* will not change until **Aodd** is cleared by asserting the signal *clear*. Once cleared, **Aodd** waits for the first new data value on *a0* and then re-assigns the desired new value to *odd* .

If MOD_EXP computes a 1024 bit modular exponentiation then there are 32 window positions per 1024 $\times$ 1024 bit carry-sum row. If MOD_EXP computes a 2048 bit modular exponentiation then there are 64 window positions per carry-sum row. Consequently, the value of *odd* is only re-assigned once every 32 or 64 systolic window positions and the performance of **Aodd** is therefore not critical. The **Aodd** implementation shown in Figure 7.14 assumes a fundamental-mode of operation, and can be verified by the Veraci program outlined in Figure 7.13.

7.6.3 Ask-Do Control

The mixture of bundled-data and dual-rail protocols used by MOD_EXP extends to the top-level control logic where a number of global flags are used to control sequencing and repetition. The ask-do circuit shown in Figure 7.15 is an example of a custom designed handshake module that was used extensively in MOD_EXP.

The purpose of an ask-do circuit is to perform one of two different four-phase handshakes depending on the result of a special query handshake, controlled by *ask*. A Veraci program for the ask-do circuit is outlined in Figure 7.16. Correctness of the ask-do circuit is based on the assumption that the time between `done+` and `done-` is sufficient for whichever latch was set to clear. This assumption can be expressed as two relative timing assumptions, one for the case where `ask+` has been acknowledged by `yes+`, setting `yq`; and one for the case where `ask+` has been acknowledged by a `no+`, setting `nq`:

```
cut never [1];trace(done+;(yq-;Lyes.q_bar- ~< done-))
cut never [1];trace(done+;(nq-;Lno.q_bar-  ~< done-))
```

These two timing assumptions are structurally identical and differ only in their attention to either latch `Lyes` or latch `Lno`. Since the protocol specification for an ask-do circuit asserts that events `yes+`, `no+` be mutually exclusive, events `yq-`, `nq-` and `Lyes.q_bar-`, `Lno.q_bar-` must also be mutually exclusive. These two timing assumptions can therefore be conveniently combined into the single veraci-fragment:

```
cut never [1];trace(done+;((yq ^ nq)-;(Lyes.q_bar ^ Lno.q_bar)- ~< done-))
```

Realisation of this timing assumption in the context of MOD_EXP was not difficult since the time between `done+` and `done-` included two external paths, and therefore at least four gate delays. Conversely, the time between `done+` and the reset of either `Lyes` or `Lno` included only a local path of two gate delays.

7.7 Summary

The purpose of this chapter was to document implementation of the asynchronous dual-rail Montgomery Exponentiator MOD_EXP in a $0.18\mu m$ standard-cell process, and to highlight the application of Veraci to its design. The objective of MOD_EXP was to demonstrate the application of asynchronous design to Montgomery exponentiation, and in particular to take steps towards achieving data-independent power and timing from a delay insensitive dual-rail datapath. MOD_EXP was also shown to be useful as a non-trivial case study for Veraci due to its internal mix of data-encoding schemes and relative timing assumptions.

7.7.1 Practical Experiences with Veraci

Short verification times and a Verilog-based input-file format make Veraci ideally suited to use as an interactive design tool: A circuit that is considered to be incorrect by Veraci can be corrected either by admitting to certain timing assumptions or by inserting additional

```
module SI latch(q, q_bar, s, r);
  output q, q_bar;
  input  s, r;

  assign     q = ~(r | q_bar);
  assign q_bar = ~(s | q);
  << initial q_bar & ~q >>
endmodule

module SI ask_do(done, ask, go_yes, go_no, go, yes, no, done_yes, done_no);
  output done, ask, go_yes, go_no;
  input go, yes, no, done_yes, done_no;
  ...
  latch Lyes(yq, yqb, yes, done);
  latch  Lno(nq, nqb, no, done);
  ...
<<
  initial ~go & ~yes & ~no & ~done_yes & ~done_no

  protocol [ go+;ask+; (
     yes+;(ask-;yes- || ((go_yes+;done_yes+);(done+;go- || go_yes-;done_yes-))) |.
     no+; (ask-;no-  || ((go_no+;done_no+);  (done+;go- || go_no-;done_no-)))
             ); done- ]

   // assert that there is enough time to clear the latches
   cut never [1];trace(done+;((yq ^ nq)-;(Lyes.q_bar ^ Lno.q_bar)- ~< done-))
>>
endmodule
```

Figure 7.16: Veraci program for the ask-do control circuit.

circuitry. Although this chapter did not document any specific examples of errors detected by Veraci during the design of MOD_EXP, such errors were common during the interactive design of many modules. In particular, both Aodd and the Ask-Do modules proved problematic when the synthesis of speed-independent solutions was attempted using Petrify [24]. The adoption of an interactive design methodology using Veraci permitted the author to assert certain timing assumptions that simplified these circuits considerably, yet could easily be accommodated by the chosen MOD_EXP architecture.

Chapter 8
Conclusions

The purpose of this thesis is to demonstrate that proposition-oriented verification is both flexible and practical. The original inspiration behind proposition-oriented behaviour stemmed from the observation that common verbal building-blocks for describing circuit behaviours include levels *and* events. Conventional level and event-oriented behaviours assert that only levels *or* events are necessary to describe a circuit execution, and therefore that a choice be made between the two. This thesis challenges this assertion, arguing that a choice between behavioural models should not be driven by mathematical economy, but by the accuracy with which these models mimic their verbal counterparts.

8.1 Summary

This thesis consists of five work chapters, Chapters 3 to 7. Four of these five work chapters document the evolution of a proposition-oriented verification methodology and its implementation in the automatic verification program Veraci. The evolution begins in Chapter 3, where proposition-oriented behaviour is formalised in the context of gate networks as a sequence of network propositions. These sequences are then used as a set-theoretic foundation on which two simple proposition-oriented notations are introduced. The first notation extends regular-expressions to reason over network propositions, and the second notation extends trace-expressions to reason over network propositions.

Chapter 4 builds on the set-theoretic model presented in Chapter 3 and demonstrates an algorithmic translation from proposition-oriented specifications into special finite automata called proposition automata. Proposition automata differ from conventional finite automata in that their input symbols are network propositions and are not therefore mutually-exclusive. This lack of exclusivity necessitates modification of conventional finite automata construction algorithms before they can be applied to proposition automata. Chapter 4 addresses this issue and, in particular, articulates the notion of a mutually-exclusive cover set as the mapping between proposition automata and conventional finite automata.

Chapter 5 describes a verification procedure for component networks and proposition automata that is based on a symbolic representation using Binary Decision Diagrams. This verification procedure classes proposition automata into two types and two senses: the type of a proposition automaton determines whether it is a constraint or a specification, and the sense of a proposition automaton determines whether it is an assertion of the valid or an assertion of the invalid.

Chapter 6 documents implementation of the verification procedure presented in Chapter 5 as the design tool Veraci. Chapter 6 then uses Veraci to quantify a number of advantages of proposition-oriented behaviour over level and event-oriented behaviours: Sections 6.3 and 6.4 document level-event unification and event-abstraction; Sections 6.5 and 6.6 document the application of network propositions to safety and causality without progress; and Section 6.7 introduces a new trace-expression operator called *biased* composition, which is shown to efficiently encode relative timing assumptions similar to the chain constraints of Negulescu and Peeters [78]. Chapter 6 concludes with an initial investigation on the performance of Veraci in comparison to Versify [86], an automatic event-oriented verification program also based on BDDs. In the context of the simple conflict-free benchmarks used, the result of this investigation is a consistent eventual performance gain by Veraci over Versify.

Chapter 7 moves away from formal verification to the design of a 2048 bit dual-rail asynchronous Montgomery exponentiator for use cryptographic devices. This exponentiator, called MOD_EXP, intends to fulfil two aims: to demonstrate the application of asynchronous logic to Montgomery exponentiation, and to investigate the application of Veraci to a non-trivial design project. The first aim is satisfied by the successful tape-out an subsequent working silicon for MOD_EXP in a 0.18μm standard-cell process, and the second aim is satisfied by example Veraci programs used to verify hand-crafted circuit modules from inside of MOD_EXP.

8.2 Discussion

The formal verification of asynchronous circuits is an active area of research, and there are many different notations each of which has its strengths and weaknesses. The most notable theoretical difference between proposition-oriented behaviours and either their event or level-oriented counterparts is a loss of exclusivity between alphabet symbols. The most notable practical difference between proposition-oriented behaviours and either their event or level-oriented counterparts is an ability to freely mix levels and events. These two differences do not however imply that proposition-oriented behaviours are more expressive than their level and event-oriented counterparts. The reason for this is that it is always possible to infer a set of level or event-oriented behaviours from a single proposition-oriented behaviour, and conversely to infer a proposition-oriented behaviour from any level or event-oriented behaviour. The significance of proposition-oriented behaviours is their ability to offer the designer a *perceived* increase in flexibility at little cost to the underlying verification engine.

The example Veraci programs in Chapters 6 and 7 are intended as demonstrations of the ways in which levels and events can be usefully mixed. They are also intended to emphasise that practical asynchronous circuits may depend on a variety of assertions including speed-independence, delay-insensitivity, fundamental-mode operation, and relative timing assumptions.

Some possible relatives of network propositions that can be found in the literature include the generalised STGs of Vanbekbergen [103] and Lamport's Temporal Logic of Actions [55]. Generalised STGs relate to proposition-oriented behaviours as a form of syntactic sugaring applied to STGs: arcs in a generalised STG can be augmented with boolean guards, and transitions in a generalised STG may be identified as *level-transitions* which assert only the next-state value of a wire. For example, a level-transition to 1 on wire x, denoted x^1, equates to x^+ only if the current value of $x = 0$. Lamport's TLA relates to proposition-oriented behaviours in that an explicit notion of actions as propositions over current-next state pairs is also used. TLA actions differ from active propositions in that their purpose is effectively to build transition relations rather than to abstract the generic concept of an event in a relative-time execution model. TLA actions are therefore free of any correctness conditions and include any network proposition as opposed to only active propositions.

The final contribution of this work is the dual-rail asynchronous Montgomery exponentiator, MOD_EXP, which is significant since it is the first Montgomery exponentiator to be fabricated using asynchronous logic. Although the performance of MOD_EXP could have been improved, it is competitive when compared to existing embedded RSA encoder-decoders. Furthermore, working silicon from the foundry makes MOD_EXP an effective platform on which to validate predictions regarding the resistance of delay-insensitive asynchronous circuits to both data-dependent timing and power attacks [71]. A discussion on these claims is beyond the scope of this thesis since power analysis tests have yet to be performed on MOD_EXP.

8.3 Further Work

Proposition-oriented behaviours are a generic concept orthogonal to many other formal methods in asynchronous design. In this sense the proposition-oriented methodology evolved in Chapters 3 to 6 is no more than a proof of concept platform on which further formal methods can be applied.

8.3.1 Other Proposition-Oriented Notations

Network propositions need not apply to regular-expressions and trace-expressions alone. The application of network propositions to other formalisms such as process spaces and DI algebrae is entirely possible, and represents an interesting topic for further research.

8.3.2 Liveness and Progress

Liveness and progress are assertions that certain behaviours *will* happen. This thesis does not consider liveness, and the proposition-oriented verification methodology behind Veraci is only capable of reasoning with behaviours that can *already* happen. Liveness is an important property and the ability to confirm liveness can improve a formal methodology considerably. Since proposition-oriented behaviours do not preclude notions of liveness, the extension of Veraci to account for liveness is in principle possible, and in practice would enable Veraci to detect a number of subtle circuit errors that it cannot currently account for.

8.3.3 Synthesis

Synthesis differs from verification in that it infers an implementation automatically from a specification. The synthesis of circuits from proposition-oriented specifications is beyond the scope of this thesis, but represents another interesting topic for further research.

8.3.4 Refinement

Refinement is a relationship between two behavioural models. It is used to assert that one model is a suitable alternative for the other. Refinement relations can be used to formalise structured translation from specification to circuit, and are common to delay-insensitive models of behaviour [49, 106]. This thesis does not consider refinement relationships between different proposition-oriented specifications, however this it not to say that such refinement is impossible. Of particular relevance to further research is the application of event abstraction to refinement, in which different delay-insensitive data encoding schemes might be viewed as refinements of generic delay-insensitive data values.

8.3.5 Execution Models

It is the opinion of the author that any performance gain between Veraci and Versify stems primarily from the use of a Multiple-Winner execution model instead of a Single-Winner execution model. The comparison between Veraci and Versify in Chapter 6 is not proof of this claim, but merely an observation that that the relative performance between Single and Multiple-Winner execution models using BDD-based verification requires further research.

References

[1] Alfred V. Aho, Revi Sethi, and Jeffrey D. Ullman. *Compilers – Principles, Techniques, and Tools*. Addison Wesley, 1986.

[2] Algorithmic Research Inc. PrivateCard: an advanced cryptographic public/private key smarcard. http://www.issos.com/catalog/arx/PrivateCard Complete Datasheet.pdf.

[3] W. J. Bainbridge and S. B. Furber. Delay insensitive system-on-chip interconnect using 1-of-4 data encoding. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 118–126. IEEE Computer Society Press, March 2001.

[4] H. Barringer, D. Fellows, G. D. Gough, P. Jinks, B. Marsden, and A. Williams. Design and simulation in Rainbow: A framework for asynchronous micropipeline circuits. In A. G. Bruzzone and U. J. H. Kerckhoffs, editors, *Proceedings of the European Simulation Symposium*, volume 2, pages 567–571. Society for Computer Simulation International, October 1996.

[5] Kees van Berkel. *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*, volume 5 of *International Series on Parallel Computation*. Cambridge University Press, 1993.

[6] Kees van Berkel and Arjan Bink. Single-track handshaking signaling with application to micropipelines and handshake circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 122–133. IEEE Computer Society Press, March 1996.

[7] Kees van Berkel, Ferry Huberts, and Ad Peeters. Stretching quasi delay insensitivity by means of extended isochronic forks. In *Asynchronous Design Methodologies*, pages 99–106. IEEE Computer Society Press, May 1995.

[8] Kees van Berkel, Joep Kessels, Marly Roncken, Ronald Saeijs, and Frits Schalij. The VLSI-programming language Tangram and its translation into handshake circuits. In *Proc. European Conference on Design Automation (EDAC)*, pages 384–389, 1991.

[9] T. L. Booth. *Sequential Machines and Automata Theory*. John Wiley and Sons, Inc., New York, 1967.

[10] K.S. Brace, R.L. Rudell, and R.E. Bryant. Efficient Implementation of a BDD Package. In *27th ACM/IEEE Design Automation Conference*, pages 40–45, Orlando, Florida, June 1990. ACM/IEEE, IEEE Computer Society Press.

[11] Stephen D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, 1984.

[12] Erik Brunvand. *Translating Concurrent Communicating Programs into Asynchronous Circuits*. PhD thesis, Carnegie Mellon University, 1991.

[13] R.E. Bryant. Graph-based algorithms for boolean function manipulation. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, 1990.

[14] R.E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[15] J. A. Brzozowski and H. Zhang. Delay-insensitivity and semi-modularity. Technical Report CS-97-11, Dept. of Comp. Science, Univ. of Waterloo, March 1997.

[16] Janusz A. Brzozowski and Carl-Johan H. Seger. *Asynchronous Circuits.* Springer-Verlag, 1995.

[17] Cambridge Consultants Ltd. The xap asic processor. http://www.camcon.co.uk/xap.html.

[18] A. Camilleri, M.J.C. Gordon, and T.F. Melham. Hardware verification using higher order logic. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs*, pages 41–66, Amsterdam, September 1986. North-Holland.

[19] C.-L. Chang and R.C.-T. Lee. *Symbolic Logic and mechanical Theorem Proving.* Academic Press, 1973.

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

[21] E.M. Clarke and E.A. Emerson. Synthesis of Synchronization Skeletons for Branching Time Temporal Logic. In *Logics of Programs: Workshop*, volume 131 of *Lecture Notes in Computer Science*, Yorktown Heights, New York, May 1981. Springer-Verlag.

[22] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic. In *Proceedings of the tenth Annual ACM Symposium on Principles of Programming Languages*, 1983.

[23] William S. Coates, Jon K. Lexau, Ian W. Jones, Scott M. Fairbanks, and Ivan E. Sutherland. FLEETzero: An asynchronous switch fabric chip experiment. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 173–182. IEEE Computer Society Press, March 2001.

[24] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Transactions on Information and Systems*, E80-D(3):315–325, March 1997.

[25] Jordi Cortadella, Michael Kishinevsky, Steven M. Burns, and Ken Stevens. Synthesis of asynchronous control circuits with automatically generated timing assumptions. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 324–331, November 1999.

[26] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alex Yakovlev. Methodology and tools for state encoding in asynchronous circuit synthesis. In *Proc. ACM/IEEE Design Automation Conference*, 1996.

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

[28] O. Coudert, C. Berthet, and J.C. Madre. Verification of synchronous sequential machines using symbolic execution. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 365–373, Grenoble, France, June 1989. Springer-Verlag.

[29] Al Davis and Steven M. Nowick. Asynchronous circuit design: Motivation, background, and methods. In Graham Birtwistle and Al Davis, editors, *Asynchronous Digital Circuit Design*, Workshops in Computing, pages 1–49. Springer-Verlag, 1995.

[30] Al Davis and Steven M. Nowick. An introduction to asynchronous circuit design. In A. Kent and J. G. Williams, editors, *The Encyclopedia of Computer Science and Technology*, volume 38. Marcel Dekker, New York, February 1998.

[31] David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. ACM Distinguished Dissertations. MIT Press, 1989.

[32] D.L. Dill and E.M. Clarke. Automatic verification of asynchronous circuits using temporal logic. *IEE Proceedings*, 133 Part E(5):276–282, September 1986.

[33] Jo Ebergen and Robert Berks. VERDECT: A verifier for asynchronous circuits. *IEEE Technical Committee on Computer Architecture Newsletter*, October 1995.

[34] Jo C. Ebergen. A technique to design delay-insensitive vlsi circuits. Research Report CS-R8622, Centrum voor Wiskunde en Informatica, July 1986.

[35] Jo C. Ebergen. A formal approach to designing delay-insensitive circuits. *Distributed Computing*, 5(3):107–119, 1991.

[36] Jo C. Ebergen and Ad M. G. Peeters. Modulo-N counters: Design and analysis of delay-insensitive circuits. In Jørgen Staunstrup and Robin Sharp, editors, *Designing Correct Circuits*, volume A-5 of *IFIP Transactions*, pages 27–46. Elsevier Science Publishers, 1992.

[37] Stephen E. Eldridge and Colin D. Walter. Hardware implementation of montgomery's modular multiplication algorithm. *IEEE Transactions on Computers*, 42(6):693–699, 1993.

[38] S. B. Furber, A. Efthymiou, and Montek Singh. A power-efficient duplex communication system. In Alex Yakovlev and Reinder Nouta, editors, *Asynchronous Interfaces: Tools, Techniques, and Implementations*, pages 145–150, July 2000.

[39] Stephen B. Furber and Paul Day. Four-phase micropipeline latch control circuits. *IEEE Transactions on VLSI Systems*, 4(2):247–253, June 1996.

[40] D. A. Gaubatz. *Logic Programming Analysis of Asynchronous Digital Circuits.* PhD thesis, Cambridge University, 1991.

[41] Peter Giblin. *Primes and Programming: An Introduction to Number Theory with Computing.* Cambridge University Press, 1993.

[42] M.J.C. Gordon. HOL: A machine oriented formulation of higher order logic. Technical Report 68, Computer Laboratory, University of Cambridge, May 1985.

[43] A. Gupta. Formal hardware verification methods: A survey. *Journal of Formal Methods in System Design*, 1:151–238, 1992.

[44] Gael Hachez and Jean-Jacques Quisquater. Montgomery exponentiation with no final subtractions: Improved results. In *Cryptographic Hardware and Embedded Systems*, pages 293–301, 2000.

[45] Scott Hauck. Asynchronous design methodologies: An overview. Technical Report TR 93-05-07, Department of Computer Science and Engineering, University of Washington, Seattle, 1993.

[46] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[47] Intel Inc. Intel itanium processor: High performance on security algorithms (rsa decryption kernel). http://developer.intel.com/design/itanium/downloads/itaniumssl.pdf.

[48] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[49] M. B. Josephs and J. T. Udding. An overview of DI algebra. In T. N. Mudge, V. Milutinovic, and L. Hunter, editors, *Proc. Hawaii International Conf. System Sciences*, volume I, pages 329–338. IEEE Computer Society Press, January 1993.

[50] J.R. Burch, E.M. Clarke, and D.E. Long. Symbolic Model Checking for Sequential Circuit Verification. In *Proceedings of VLSI'91*, pages 49–58, Edinburgh, Scottland, August 1990.

[51] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, RSA, DSS, and other systems. In *CRYPTO*, pages 104–113, 1996.

[52] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *CRYPTO*, pages 388–397, 1999.

[53] A. Kondratyev, J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Taubin. The use of Petri nets for the design and verification of asynchronous circuits and systems. *Journal of Circuits, Systems and Computers*, 8(1), 1998. ftp://ftp.u-aizu.ac.jp/u-aizu/async/pn-review98.ps.gz.

[54] Leslie Lamport. "Sometime" is sometimes "Not never" – On the temporal logic of programs. In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 174–185, Las Vegas, Nevada, 1980.

[55] Leslie Lamport. A temporal logic of actions. Technical Report SR57, Digital Equipment Corporation, Systems Research Center, April 1990.

[56] D. E. Long. CMU BDD package. http://emc.cmu.edu/pub/bdd/bddlib.tar.Z., 1993.

[57] Willem C. Mallon and Jan Tijmen Udding. Building finite automatons from DI specifications. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 184–193, 1998.

[58] Z. Manna and A. Pnueli. Verification of concurrent programs: The temporal framework. In R.S. Boyer and J.S. Moore, editors, *Correctness Problems in Computer Science*, pages 215–273, London, 1982. Academic Press.

[59] Alain J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In William J. Dally, editor, *Advanced Research in VLSI*, pages 263–278. MIT Press, 1990.

[60] Alain J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*, UT Year of Programming Series, pages 1–64. Addison-Wesley, 1990.

[61] Alain J. Martin. Asynchronous datapaths and the design of an asynchronous adder. *Formal Methods in System Design*, 1(1):119–137, July 1992.

[62] K.L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1992. CMU-CS-92-131.

[63] Alfred Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1996.

[64] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer-Verlag, New York, 1980.

[65] R. Milner. *Communication and Concurrency.* Prentice-Hall International, London, 1989.

[66] R. Milner. The polyadic pi-calculus: a tutorial. Technical report, LFCS University of Edinburgh, October 1991.

[67] Faron Moller and Perdita Stevens. Edinburgh Concurrency Workbench user manual (version 7.1). http://www.dcs.ed.ac.uk/home/cwb/.

[68] Charles E. Molnar, Ting-Pien Fang, and Frederick U. Rosenberger. Synthesis of delay-insensitive modules. In Henry Fuchs, editor, *1985 Chapel Hill Conference on Very Large Scale Integration*, pages 67–86. Computer Science Press, 1985.

[69] U. Montanari and F. Rossi. Acta informatica, 1995.

[70] P. L. Montgomery. Modular Multiplication Without Trial Division. *Mathematics of Computation*, 44(170):519–521, April 1985.

[71] S. W. Moore. Protecting consumer security devices. In *Smart Card Programming and Security*, volume 2140 of *Lecture Notes in Computer Science*, page 1, 2001. International Conference on Research in Smart Cards, E-smart 2001, Cannes, France, September 19-21, 2001.

[72] S. W. Moore, G. S. Taylor, R. D. Mullins, and P. Robinson. Report on our first test chip. In *10th UK asynchronous forum*, 2001.

[73] D. E. Muller. Theory of asynchronous circuits. Technical report, University of Illinois Digital Computer Laboratory, December 1955.

[74] David E. Muller and W. S. Bartky. A theory of asynchronous circuits. In *Proceedings of an International Symposium on the Theory of Switching*, pages 204–243. Harvard University Press, April 1959.

[75] Chris J. Myers. *Asynchronous Circuit Design*. Wiley, 2001.

[76] Radu Negulescu. Process spaces. Technical Report CS-95-48, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, December 1995.

[77] Radu Negulescu. *Process Spaces and Formal Verification of Asynchronous Circuits*. PhD thesis, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, August 1998.

[78] Radu Negulescu and Ad Peeters. Verification of speed-dependences in single-rail handshake circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 159–170, 1998.

[79] Steven M. Nowick. *Automatic Synthesis of Burst-Mode Asynchronous Controllers*. PhD thesis, Stanford University, Department of Computer Science, 1993.

[80] Priyadarsan Patra and Donald Fussel. Efficient building blocks for delay insensitive circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 196–205, November 1994.

[81] Ad M. G. Peeters. *Single-Rail Handshake Circuits*. PhD thesis, Eindhoven University of Technology, June 1996.

[82] Marco A. Peña, Jordi Cortadella, Alex Kondratyev, and Enric Pastor. Formal verification of safety properties in timed circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 2–11. IEEE Computer Society Press, April 2000.

[83] R. L. Rudell R. E. Bryant, K. S. Brace. Efficient implementation of a BDD package. In *ACM/IEEE Degign Automation Conference*, pages 40–45, 1990.

[84] R. Rudell. Dynamic Variable Ordering for Ordered Binary Decision Diagrams. In *IEEE /ACM International Conference on CAD*, pages 42–47, Santa Clara, California, November 1993. ACM/IEEE, IEEE Computer Society Press.

[85] Martin Rem, Jan L.A. van de Snepscheut, and Jan T. Udding. Trace theory and the definition of hierarchical components. In *Third CalTech Conference on Very Large Scale Integration*, pages 225–239. Computer Science Press, Inc., 1983.

[86] Oriol Roig. *Formal Verification and Testing of Asynchronous Circuits*. PhD thesis, Univsitat Politècnia de Catalunya, May 1997.

[87] L. Y. Rosenblum and A. V. Yakovlev. Signal graphs: from self-timed to timed ones. In *Proceedings of International Workshop on Timed Petri Nets*, pages 199–207, Torino, Italy, July 1985. IEEE Computer Society Press.

[88] Securealink Inc. PCC-ISES cryptographic accelerator chip. http://www.safenet-inc.com/securealink/pcc-ises.html.

[89] Securealink Inc. PCC810 secure smartcard reader device. http://www.safenet-inc.com/securealink/pcc810.html.

[90] Charles L. Seitz. System timing. In Carver A. Mead and Lynn A. Conway, editors, *Introduction to VLSI Systems*, chapter 7. Addison-Wesley, 1980.

[91] A. Shamir. A fast signature scheme. MIT Laboratory for Computer Science Technical Report, 1978.

[92] T. Shiple. Survey of equivalences for transition systems. University of California, Berkeley, EE 290A Class Report, 1993.

[93] Jan L. A. van de Snepscheut. *Trace Theory and VLSI Design*, volume 200 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[94] Gerald E. Sobelman and Karl Fant. CMOS circuit design of threshold gates with hysteresis. In *Proc. International Symposium on Circuits and Systems*, pages 61–64, June 1998.

[95] B. Stroustrup. *The C++ programming language*. Addison-Wesley, Reading, MA, second edition, 1991.

[96] Ivan Sutherland and Scott Fairbanks. GasP: A minimal FIFO control. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 46–53. IEEE Computer Society Press, March 2001.

[97] Ivan E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.

[98] George Taylor. Personal communication.

[99] D. E. Thomas and Philip Moorby. *The Verilog Hardware Description Language*. Kluwer, USA, 1991.

[100] H. J. Touati, H. Savoj, B. Lin, R.S. Brayton, and A. Sangiovanni-Vincentelli. Implicit State Enumeration of Finite State Machines using BDD's. In *IEEE /ACM International Conference on CAD*, pages 130–132. ACM/IEEE, 1990.

[101] Jan Tijmen Udding. A formal model for defining and classifying delay-insensitive circuits. *Distributed Computing*, 1(4):197–204, 1986.

[102] S. H. Unger. *Asynchronous Sequential Switching Circuits*. Wiley-Interscience, John Wiley & Sons, Inc., New York, 1969.

[103] Peter Vanbekbergen, Chantal Ykman-Couvreur, Bill Lin, and Hugo de Man. A generalized signal transition graph model for specification of complex interfaces. In *Proc. European Design and Test Conference*, pages 378–384. IEEE Computer Society Press, 1994.

[104] M. Vardi and P. Wolper. An automata theoretic approach to automatic program verification. In *Proceedings of the IEEE Lectures in Computer Science*, volume 1, pages 332–345, 1986.

[105] Tom Verhoeff. Delay-insensitive codes—an overview. *Distributed Computing*, 3(1):1–8, 1988.

[106] Tom Verhoeff. *A Theory of Delay-Insensitive Systems.* PhD thesis, Dept. of Math. and C.S., Eindhoven Univ. of Technology, May 1994.

[107] Tom Verhoeff. Analyzing specifications for delay-insensitive circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 172–183, 1998.

[108] C. Walter. Montgomery exponentiation needs no final subtractions. In *Electronics Letters*, volume 35, pages 1831–1832, 1999.

[109] Colin D. Walter. Systolic modular multiplication. *IEEE Transactions on Computers*, 42(3):376–378, 1993.

[110] Colin D. Walter. Improved linear systolic array for fast modular exponentiation. *IEE Computers and Digital Techniques*, 147(5):323–328, 2000.

[111] Colin D. Walter and Susan Thompson. Distinguishing exponent digits by observing modular subtractions. In *Topics in Cryptology CT-RSA 2001*, volume 2020 of *Lecture Notes in Computer Science*, pages 208–222, 2001. The Cryptographers' Track at RSA Conference 2001 San Francisco, CA, USA, April 8-12, 2001.

[112] B. W. Watson. A taxonomy of finite automata construction algorithms. citeseer.nj.nec.com/watson94taxonomy.html, 1994.

[113] N. Weste and K. Eshraghian. *Principles of CMOS VLSI Design.* Addison-Wesley, 1985.

[114] Kenneth Y. Yun and David L. Dill. Automatic synthesis of extended burst-mode circuits: Part i (specification and hazard-free implementation). *IEEE Transactions on Computer-Aided Design*, 18(2):101–117, February 1999.

[115] Kenneth Y. Yun and David L. Dill. Automatic synthesis of extended burst-mode circuits: Part ii (automatic synthesis). *IEEE Transactions on Computer-Aided Design*, 18(2):118–132, February 1999.