COMPOSITIONAL MODEL CHECKING OF PARTIALLY ORDERED STATE SPACES

by

Scott Hazelhurst
B.Sc.Hons, University of the Witwatersrand, Johannesburg, 1986
M.Sc., University of the Witwatersrand, Johannesburg, 1988

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

in

THE FACULTY OF GRADUATE STUDIES
(Department of Computer Science)

We accept this thesis as conforming
to the required standard

. . . .

. . . .

. . . .

. . . .

. . . .

THE UNIVERSITY OF BRITISH COLUMBIA

January 1996

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Computer Science
The University of British Columbia
2366 Main Mall
Vancouver, Canada
V6T 1Z4

Date:

Abstract

Symbolic trajectory evaluation (STE) — a model checking technique based on partial order representations of state spaces — has been shown to be an effective model checking technique for large circuit models. However, the temporal logic that it supports is restricted, and as with all verification techniques has significant performance limitations. The demand for verifying larger circuits, and the need for greater expressiveness requires that both these problems be examined.

The thesis develops a suitable logical framework for model checking partially ordered state spaces: the temporal logic TL and its associated satisfaction relations, based on the quaternary logic $\mathcal{Q}$. TL is appropriate for expressing the truth of propositions about partially ordered state spaces, and has suitable technical properties that allow STE to support a richer temporal logic. Using this framework, verification conditions called *assertions* are defined, a generalised version of STE is developed, and three STE-based algorithms are proposed for investigation. Advantages of this style of proof include: models of time are incorporated; circuits can be described at a low level; and correctness properties are expressed at a relatively high level.

A primary contribution of the thesis is the development of a compositional theory for TL assertions. This compositional theory is supported by the partial order representation of state space. To show the practical use of the compositional theory, two prototype verification systems were constructed, integrating theorem proving and STE. Data is manipulated efficiently by using binary decision diagrams as well as symbolic data representation methods. Simple heuristics and a flexible interface reduce the human cost of verification.

Experiments were undertaken using these prototypes, including verifying two circuits from the IFIP WG 10.5 Benchmark suite. These experiments showed that the generalised STE algorithms were effective, and that through the use of the compositional theory it is possible to verify very large circuits completely, including detailed timing properties.

Table of Contents

List of Figures

List of Tables

List of Important Definitions, Theorems and Lemmas

Acknowledgement

'A person is a person through other people.'

The person I asked to proofread this said it was too sentimental. Perhaps it is, but the experience and time spent on this thesis has been a very important part of my life, and it is important for me to thank some of the many people who have contributed to this experience.

The Computer Science Department and the Integrated Systems Laboratory at UBC have been a wonderful place to work. First, I must thank my supervisor, Carl Seger. Carl has supported and mentored me since I arrived at UBC, and I am very grateful for all that he has done, well beyond what I could have expected. Carl, I have learned a tremendous amount from you.

I also thank the other members of my supervisory committee: Paul Gilmore, Nick Pippenger, Son Vuong, and particularly Mark Greenstreet who has provided enthusiasm, ideas, and support and given freely of his time.

Thanks to all members of the ISD group, especially Mark Aaagaard, Nancy Day, and Mike Donat, Catherine Leung, Hélène Wong and most especially Andy Martin.

There are many others in the department who have been important too. My office-mates, Carol Saunders, Catherine Leung, Nana Kender, and Rita Dilek, have been supportive. Hélène Wong, Marcelo Walter, Peter Smith, Jim Boritz and Alistair Veitch have all made being here a richer experience. The administrative and technical staff and management of the department have provided an environment that was a pleasure to work in. Financial support from UBC, NSERC and ASI made my research possible.

For the last two years, Green College has not only been a stimulating academic environment but also a great community to be part of, and I am very fortunate to have had this experience. There are many College members who have helped me over the years. There are two people whose friendship I shall always count as major accomplishments at UBC. I am very grateful to Patricia Yuen-Wan Lin for sharing the journey and giving me constant encouragement. Peter Urmetzer has been a very good friend throughout. Thank you for all you have done.

Many others helped get me here and have supported me since, and I would like to thank all my friends. I wish to particularly mention Saul Johnson and Annette Wozniak (and Kiah); Conrad Mueller, Ian Sanders, Philip Machanick and all the other Turing Tipplers; Sheila Rock; Stan Szpakowicz; Anthony Low; Georgia and Hugh Humphries.

Finally, my greatest thanks go to my family, my parents, David and Ethel, and sister, Jo Ann. The encouragement and support you have always given me, in many different ways over a long period of time, has always encouraged and sustained me. This experience and what I have learned here is due to your efforts.

To my parents
and the memory of
Maurice Yatt
and Edith Hazelhurst

Woza Moya omuhle
Sibambane ngezandla sibemunye
— Work For All, Juluka, 1983

Chapter 1

Introduction

1.1 Motivation

As computers become ubiquitous in our society, as more parts of our global society are affected directly and indirectly by computers, the need to ensure their safe and correct behaviour increases. The hyperbole encountered in the media tends to make people blasé about the importance of computers and undervalue the revolutionary effect that computers have had. But, as our dependency on computers increases, so does the complexity of computer systems, making it more difficult to design and build correct systems at the same time as it becomes more important to do so. What we can do 'sort of' right far exceeds what we can do properly.

As a scientific and engineering discipline computer science is intimately concerned about making predictions about and knowing the properties of computer systems, and it is here that mathematics and the application of methods of formal mathematics is critical.

Traditional methods of ensuring correct operation of software and hardware are often not able to provide a sufficiently high degree of confidence of correctness. Methods such as testing and simulation of systems cannot hope to provide anywhere near exhaustive coverage of system behaviour, and while sophisticated test generation techniques exist, the sheer size of systems makes testing more and more difficult and expensive.

Verification — a mathematical proof of the correctness of a design or implementation — uses formal methods to obviate these problems. Questions of verification have been at the heart of computer science since the work of Turing and others [124], and the fundamental limits of computation (questions such as computability, tractability and completeness) are of immense consequence when discussing the theoretical and practical limitations of verification. The theoretical importance of verification is reflected in the practical consequences of verification, or lack thereof, which has been illustrated recently by the extremely well-publicised error in a commercial microprocessor [64, 104, 110].

This is not to suggest that formal methods are a panacea, and that other approaches are unimportant. Indeed, in many safety critical or other important applications, there may be social and ethical constraints on what should be built. There are many technical and non-technical factors that will affect the quality of systems that are built. Testing at different levels will continue to be important.

Moreover, there are limitations on what verification can offer. With respect to hardware verification, Cohn points out that neither the actual hardware implementation nor the intentions motivating the device can be subject to formal methods [42]. Verification is inherently limited by the models used. And, verification is expensive computationally and requires a high level of expertise. Although there has been some success in the use of formal methods, there are a number of practical and organisational problems that must be dealt with, especially when formal methods are first used by an organisation [114, 119].

Over a quarter of a century ago, C.A.R. Hoare summed up his view of the use of formal methods [82]:

> The practice of supplying proofs for nontrivial programs will not become widespread until considerably more powerful proof techniques become available, and even then will not be easy. But the practical advantages of program proving will eventually outweigh the difficulties, in view of the increasing costs of programming error. At present, the method which a programmer uses to convince himself of the correctness of his program is to try it out in particular cases and to modify it if the results produced do not respond to his intentions. After he has found a reasonably wide variety of example cases on which the program seems to work, he believes that it will always work. The time spent in this program testing is often more than half the

> time spent on the entire programming project; and with a realistic costing of machine time, two thirds (or more) of the cost of the project is involved in removing errors during this phase.
>
> The cost of removing errors discovered after a program has gone into use is often greater, particularly in the case of items of computer manufacturer's software for which a large part of the expense is borne by the user. And finally the cost of error in certain types of program may be almost incalculable — a lost spacecraft, a collapsed building, a crashed aeroplane, or a world war. Thus, the practice of program proving is not only a theoretical pursuit, followed in the interests of academic respectability, but a serious recommendation for the reduction of costs associated with programming error.

As a manifesto for verification, with minor changes it might well have been written today. On the surface, re-reading this may seem to be cause for pessimism — what has changed in 25 years? However, this is misleading. Verification is very difficult and can be extremely expensive[1]; this complexity, lack of expertise, and conservatism are problems in the greater adoption of formal methods. But, the cost of not performing verification can be much higher[2], and as will be seen in Chapter 2, there have been significant theoretical and practical advances showing that the promise of advantages from formal methods has been realised. The progress that has been made, the increased needs for the use of verification, and the challenges which these needs create, make the comments expressed in this extract more relevant today than it was in 1969: we need more powerful proof techniques, and techniques that are easier to use.

The rest of this chapter is structured as follows. Section 1.2 introduces the use of verification and formal methods. Section 1.3 motivates and describes the underlying approach to verification adopted in this thesis. Section 1.4 describes the research contribution of the thesis, and Section 1.5 outlines the rest of this thesis.

[1] Owre *et al.* estimate that the cost of a partial formal specification and verification of a commercial, 500 000 transistor microprocessor as 'three man-years' of work [105].

[2] Intel estimate the cost of the flaw in the Pentium microprocessor at US$475-million [65].

1.2 Verification and the Use of Formal Methods

Consider an example of a chip which divides two 64-bit numbers. There are 2^{128} possible combinations of input. Exhaustive testing of all these combinations is an impossible feat — even if we were to test 10^9 combinations a nano-second for a million millenia we would be able to test fewer than one per cent of cases. Moreover, this testing would ignore the possible effects of internal state of the chip (it could be that the chip works correctly when initialised, but that the effect of computing some answers updates internal registers so that subsequent computations are incorrect).

This example illustrates the underlying problem in checking for correctness. The number of behaviours of a system, particularly if it is reactive or concurrent, is very large. Not only does this make exhaustive testing impractical, it makes reasoning about computer systems, whether software or hardware, difficult.

Since testing often cannot be comprehensive, verification is appealing in giving a higher confidence in the correctness of systems. The use of formal methods allows a mathematical proof to be given of correctness. Of course, we can only verify what can be modelled mathematically. The verification of the correctness of a chip is the verification of its logical design. We have some mathematical model of the behaviour of the components (gates or transistors) and use this to infer properties of the system. Such a verification is only as good as the model of the components. Models like this must make simplifications about the physical world. While often the simplifications made do not affect our ability to make predictions about the behaviour of the world, it is important to realise the potential problem.

The question of how good the model of the world is, and the problem of realising a logical design as a physical artifact are critical problems. However, they are beyond the scope of this thesis. Focussing on the problem of verifying a logical design is difficult enough, and this will be the focus of this research: this section introduces verification and some of the research problems associated with verification, and Chapter 2 will give a fuller survey of verification.

Verification requires that both the specification and the implementation be described using some mathematical notation with a well-defined formal semantics. There are many choices open to the verifier. Common choices for describing an implementation are finite state machines or labelled transition systems — often these are extracted directly from higher level descriptions such as programs. A common choice for the specification is a temporal logic, which allows the description of the intended behaviour of a system over time. If the implementation is described as a finite state machine and the specification as a set of temporal formulas, verification consists of showing that the finite state machine satisfies these formulas.

The fundamental problem with verification is that the number of states in a model of a system is exponentially related to the number of system components; this is known as the *state explosion problem*. Finding automatic verification techniques is difficult; the general versions of the problem are undecidable [124] and restricted versions remain undecidable, while others are NP-hard [55].

Many verification approaches have been suggested — these will be surveyed in the next chapter. The problems caused by large state spaces manifest themselves in different ways, as can be seen with two of the most popular methods, theorem proving and automatic model checking. A large state space imposes significant computational costs on the verification task. This is a particular problem for automatic model checking techniques, which are based on state exploration methods. Although theorem provers may be less sensitive to the size of the state space in terms of their computational cost, the cost of human intervention is high, often requiring a high degree of expertise and making the verification more difficult and much more lengthy.

Dealing with the state explosion problem motivates much research in verification, and a number of methods to limit the problem have been suggested. Some of the methods examined in this research are:

- The use of good data structures to represent model behaviour is critical. The development and use of Ordered Binary Decision Diagrams in the 1980s was very important in extending the power of automatic verification methods.
- Abstraction. By constructing an abstraction of the model, and proving properties of the abstraction rather than the model, significant performance benefits may be gained. Of course the problem of finding the abstraction, and showing that the properties proved of the abstraction are meaningful of the model are non-trivial.
- Compositionality. Divide and conquer is one of the most common strategies in computer science, and one which can be very helpful with verification. Property decomposition is useful when the cost of verification is highly sensitive to the complexity of the properties to be proved; it provides a way of combining 'smaller' results into 'larger' ones. Structural decomposition allows different parts of the system to be reasoned about separately; these separate results are then used to deduce properties of the entire system.
- Hybrid approaches. Different verification techniques have different advantages and disadvantages, so by combining different approaches it might be possible to overcome the individual disadvantages.

The choice of model of the system is critical. This choice affects the way in which properties are proved, what satisfaction means, and how abstraction and compositionality can be used. The next section motivates and describes the method of representing state space and model behaviour adopted by this thesis.

1.3 Partially-ordered State Spaces

One of the starting points of this thesis is that partially-ordered sets are effective representations of state spaces of systems. This section introduces the necessary mathematical definitions, motivates why partial orders are useful representations, describes how they are used, and then introduces an appropriate verification method.

1.3.1 Mathematical Definitions

A *partial order*, R, on a set $\mathcal{S}$ is a reflexive, anti-symmetric and transitive relation on $\mathcal{S}$, i.e. $R \subseteq \mathcal{S} \times \mathcal{S}$ and

$$\forall s \in \mathcal{S}, (s, s) \in R$$

$$(s_1, s_2), (s_2, s_1) \in R \implies s_1 = s_2$$

$$(s_1, s_2), (s_2, s_3) \in R \implies (s_1, s_3) \in R$$

Typically, an infix notation is used for partial orders. Thus, if $\sqsubseteq$ is a partial order, then $x \sqsubseteq y$ is used for $(x, y) \in \sqsubseteq$. A *preorder* on $\mathcal{S}$ is a reflexive and transitive relation.

If $\sqsubseteq$ is a partial order on $\mathcal{S}$, then it can be extended to cross-products of $\mathcal{S}$ and sequences of $\mathcal{S}$. If $\langle s_1, \ldots, s_n\rangle, \langle t_1, \ldots, t_n\rangle \in \mathcal{S}^n$, then $\langle s_1, \ldots, s_n\rangle \sqsubseteq \langle t_1, \ldots, t_n\rangle$ if $s_i \sqsubseteq t_i$, for $i = 1, \ldots, n$. Similarly for sequences (elements of $\mathcal{S}^\omega$), $s_1 s_2 s_3 \ldots \sqsubseteq t_1 t_2 t_3 \ldots$ if $s_i \sqsubseteq t_i$, for $i = 1, 2, \ldots$

If $\mathcal{S}$ is a set with partial order $\sqsubseteq$, and $s, t \in \mathcal{S}$, then u is the *least upper bound*, or *join*, of s and t if $s, t \sqsubseteq u$ (i.e. it is an upper bound) and if $s, t \sqsubseteq v$, then $u \sqsubseteq v$ (i.e. it is no larger than any other upper bound). In this thesis, the join of s and t will be denoted $s \sqcup t$. In general, it is not the case that every pair of elements in a partially ordered set has a join — a pair of elements could have many least upper bounds, each of which is incommensurable with the others, or no least upper bound at all. Similarly, the *greatest lower bound* of s and t, or the *meet* of s and t, is denoted $s \sqcap t$, and in general not all pairs of elements will have a meet.

A partially ordered set $\mathcal{S}$ is a *lattice* if every pair of elements has a meet and join. By induction, in any lattice any finite subset has a least upper bound and a greatest lower bound. $\mathcal{S}$ is a *complete lattice* if every set of elements — finite or infinite — has a least upper bound and greatest lower bound. In particular, complete lattices have unique universal upper and lower bounds. Note that all finite lattices are complete — this is a result that is used extensively in this thesis.

If $\mathcal{S}$ is a complete lattice over the partial order $\sqsubseteq$, then, under the natural extensions of $\sqsubseteq$:

- a finite cross-product of $\mathcal{S}$, $\mathcal{S}^n$ is a complete lattice; and
- $\mathcal{S}^\omega$, the set of all sequences of $\mathcal{S}$ is a complete lattice.

If $\mathcal{S}_1$ and $\mathcal{S}_2$ are two lattices with partial orders $\leq_1$ and $\leq_2$, and $g\colon \mathcal{S}_1 \rightarrow \mathcal{S}_2$ is a function, then g is *monotonic* with respect to $\leq_1$ and $\leq_2$ if $s \leq_1 t$ implies that $g(s) \leq_2 g(t)$.

If $\mathcal{S}$ is a lattice and $A \subset \mathcal{S}$, then A is *upward closed* if $a \in A$, $x \in \mathcal{S}$ and $a \sqsubseteq x$ implies that $x \in A$. Similarly, A is *downward closed* if $a \in A$, $x \in \mathcal{S}$ and $x \sqsubseteq a$ implies that $x \in A$.

Partial orders are used in two important ways in this thesis. First, given a state space, partial orders are used to compare the information content of states. $s \sqsubseteq t$ implies that s has less information than t; if $s \sqsubseteq t$ and $s \sqsubseteq u$, then informally we can think of s as representing both t and u, it is an abstraction of these two states. It is fairly easy to generate partial order models of systems like circuits from gate level descriptions of circuits, and good partial-order models from switch-level can automatically be extracted in many cases. The second way partial orders are used is to differentiate between levels of truth, a central theme in this thesis.

1.3.2 Using Partial Orders

Formally, a model can be described by $(\langle \mathcal{S}, \sqsubseteq \rangle, \mathbf{Y})$, where $\mathcal{S}$ is a complete lattice under the partial order $\sqsubseteq$ and the behaviour of the model is represented by the next-state function $\mathbf{Y} : \mathcal{S} \rightarrow \mathcal{S}$ which is monotonic with respect to the partial order. The partial order can be extended to sequences of $\mathcal{S}$.

To see why partial orders might be useful, consider as an example of a system which can be in one of five states. A next state function $\mathbf{Y}$ describes the behaviour of the system. The state space could be represented by a set containing five elements. However, there is an advantage in representing the state space with a more sophisticated mathematical structure. In this example, we represent the state space with the lattice shown in Figure 1.1 (note that this is just one possible lattice). States s_4–s_8 are the 'real' states of the system, and the other states are mathematical abstractions ($\mathbf{Y}$ can be extended to operate on all states of the lattice). The partial ordering of the lattice is an information ordering: the higher up in the ordering we are, the more we know about which state the system is in. For example, the model being in state s_1 corresponds to the system being in state s_4 or s_5. State s_9 represents a state that has contradictory information.

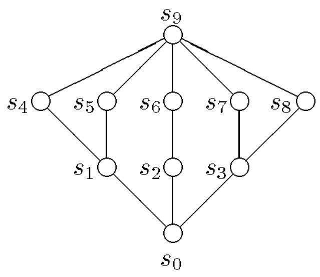


Figure 1.1: Example Lattice State Space

States like s_1 are useful because if one can prove that a property holds of state s_1, then (given the right logical framework) that property also holds of s_4 and s_5. There can be a great performance advantage in proving properties of states low in the lattice.

Furthermore, state s_9 plays an important role, since it represents states about which inconsistent information is known. Although such states do not occur in 'reality', they are sometimes artifacts of a verification process.

A human verifier may introduce conditions which are inconsistent with each other or the operation of the real system. These conditions could lead to worthless verification results — ones that while mathematically valid tell us nothing about the behaviour of the system and may give verifiers a false sense of security.[3] Since it may not be possible to detect these inconsistencies directly, it is useful to have states in which inconsistent properties can hold at the same time. In such states, a property and its negation may both hold, and we should have a way of expressing this.

In this example, the potential savings are not large, but for circuit models extremely significant savings can be made. The state space for a circuit model represents the values that the nodes in the circuit take on, and the next state function can be represented implicitly by symbolic simulation of the circuit.

The nodes in a circuit take on high (H) and low (L) voltage values; there is a natural lattice in which these voltage values can be embedded. It is useful, both computationally and mathematically, to allow nodes to take on unknown (U) and inconsistent or over-defined (Z) values. The set $\mathcal{C} = \{\mathsf{U}, \mathsf{L}, \mathsf{H}, \mathsf{Z}\}$ forms a lattice, the partial order given in Figure 1.2.

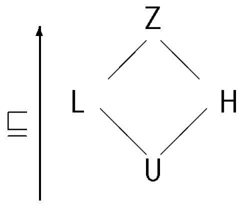


Figure 1.2: The Partial Order for $\mathcal{C}$

The state space for a circuit then is naturally represented by $\mathcal{C}^n$, which is a complete lattice. Consider a circuit with n components and a state, s, of the circuit:

$$s = \langle\, v_1, \ldots, v_m, \underbrace{\mathsf{U}, \ldots, \mathsf{U}}_{n-m} \rangle,$$

where the v_is are boolean values. With the right logical framework, if we can prove that a property g holds of the state s, then we can infer directly that the property holds for all states above it in the information ordering.

[3] 'A truth that's told with bad intent,/ Beats all the lies you can invent.'— William Blake

If we only consider the subset of states, $\{\mathsf{L}, \mathsf{H}\}^n$ (those states with known, consistent voltages on each component), there are 2^{n-m} states above s, of the form $\langle\, v_1, \ldots, v_n \,\rangle$, where the v_is are boolean variables. So, in one step 2^{n-m} 'interesting' proofs are done (this step would also prove properties about states with partial or inconsistent information). Through the judicious use of U values, the number of boolean variables needed to describe the behaviour of the circuit can be minimised, increasing the size of the circuits that can be dealt with directly.

The purpose of model checking is to determine whether a model has a certain property — ideally, a verification method should answer this 'yes' or 'no'. Unfortunately, the performance benefit gained by using only partial information compromises this goal. In the example above, while every property of the circuit will be true or false of states s_4–s_8, there will be some properties which are neither true nor false of states s_0–s_3, since there is insufficient information about those states.

The converse problem exists with a state like s_9. Assigning the same level of credibility and meaningfulness to the truth of property g in state s_9 as the truth of g in s_5 violates common sense understanding of truth.

Both these factors indicate that a two valued logic has insufficient expressiveness when dealing with a partially-ordered state space. To say that something is true or false in states like s_1 and s_9 may be very misleading. And, we shall see later that a two valued logic also has a serious technical defect in this situation.

1.3.3 Symbolic Trajectory Evaluation

Symbolic trajectory evaluation (STE) is a model checking *approach* based on partially ordered state spaces. STE computes the next state relation using symbolic simulation. Not only does this allow the partially-ordered state space structure to be exploited effectively, it supports accurate, low-level models of circuit structures. ('Approach' is emphasised above because a number of different possible STE-based algorithms and implementations exist. Moreover, STE-based algorithms can be used in different logical frameworks.)

Previous work with STE has shown that it is an effective method for many circuits (e.g., see [8, 47]) and it is recognised as one of the few methods with good asymptotic performance on a large class of non-trivial circuits [26].

STE is particularly useful in dealing with large circuits, where the circuit is modelled at a low level (gate or switch level), and where timing is important. Higher-level verifications are important too, but, as Cohn points out, realistic and detailed models of circuits are important to ensure that the mathematical results proved are meaningful [42].

Although successfully applied, these STE-based approaches are not without their problems. First, the underlying problem of the state explosion problem still exists, and as with all verification methods, better and more powerful techniques must be developed as the computation bottle-necks are still there. Second, in existing STE-based approaches, the logic used to express properties is limited; for example, disjunction and negation is not fully supported. While the logic is expressive enough for many problems and the restricted form of the logic leads to very efficient model checking algorithms, there are problems which need a richer logic. Third, previous approaches have used a two valued logic, which, in the context of partially ordered state spaces, is confusing. For a restricted logic, the complication caused by insufficient and contradictory information can be dealt with adequately in an extra-logical way; this is inadequate for a richer logic.

1.4 Research Contributions

No one verification method is suitable for all verification problems. The choice of model (how complete and what level), how correctness is expressed, and the choice of underlying theoretical framework and practical tool depends on many factors: the problem domain; what properties the verifier wishes to prove; the expertise of those involved; what level of confidence in the verification has to be obtained; and very importantly the computational and human resources available.

The research work presented in this thesis is motivated by the promise that trajectory evaluation offers in dealing with large circuits, especially where a detailed model of timing is required. The strength of this is that not only can the high-level algorithmic descriptions of functionality be verified, but the low-level implementation details can be checked too; verification can be done on switch-level or detailed gate level circuit descriptions. This is particularly important when the transformation from high-level description to low-level implementation is error-prone. Timing properties can be verified at the micro-level (e.g. checking that circuit values stabilise by the end of a clock cycle) or at the macro-level (e.g. checking which clock cycle something happens). Since many other verification methodologies have difficulty with detailed verification of large circuits, this is an important line of research to develop. This thesis starts from the premiss that extending the power of STE-based methods by increasing the range and size of systems that can be verified, and the types of properties that can be expressed in a verification is a significant contribution.

The goal of this research is to show that the applicability of symbolic trajectory evaluation can be significantly extended though the development of an appropriate temporal logic for model checking partially-ordered state spaces, and the use of a compositional theory for trajectory evaluation. The specific contributions of this thesis are listed below.

- *Proposing a suitable temporal logic suitable for partially ordered state spaces.*

  Traditional two-valued logics are unsuitable for expressing properties of partially-ordered state spaces; my first thesis is that a four-valued logic is suitable. This logic distinguishes the following four cases: true, false, under-determined, and over-determined. Not only does the four-valued logic provide a framework for representing our knowledge of the degree of truth of a proposition, it is a suitable technical framework. This framework is useful not only for STE-based model checking algorithms, but other verification methods based on partially-ordered state spaces developed in the future.

A qualification is in order — the use of uncertainty to model both state information and degrees of truth is epistemological rather than ontological in nature. The question of uncertainty in the 'real world' is well outside of the scope of this thesis. Uncertainty is used in system models because this offers significant computational advantages. This uncertainty in the model induces an uncertainty in our knowledge of the model. Thus the four-valued logic is useful to reason about our knowledge of the model, and the use of the four valued logic is proposed for its utilitarian value, not as an excursion into general philosophy.

- *Generalisation of symbolic trajectory evaluation based algorithms*

  My second thesis is that using the four-valued framework, STE-based algorithms can be generalised to support a richer logic. Providing a richer logic is important because it supports the verification of a greater range of applications. Moreover, it often makes the specification of properties clearer, which makes the verification more meaningful for the user; and more elegant specifications can also lead to more efficient model checking.

- *A Compositional Theory*

  My third thesis is that a compositional theory for model checking partially-ordered state spaces can be developed, and provides a foundation for overcoming the performance limitations of model checking. The compositional theory allows verification results to be combined in different ways into larger results. The structure of the state space lends itself to the compositional theory, and together with the compositional theory allows very large state spaces to be model checked. The key part of the development of the compositional theory is to show that it is sound; that all results inferred could, at least in principle, be directly obtained through trajectory evaluation or some other model checking algorithm.

- *Development of a practical tool*

  While the proposed four-valued logic and compositional theory have theoretical interest, a major part of the significance of the contribution of the work comes from my fourth thesis: that generalised STE and the compositional theory for model checking partially ordered state spaces using a four-valued logic can be used effectively, making a significant contribution to the size and complexity of circuits that can be verified.

  This is demonstrated by the development of prototype verification tools. These prototypes show that it is effective to combine theorem proving and STE-based model-checking as these approaches complement each other. While the prototypes are not of interest in themselves, they demonstrate that very large circuits can be formally verified using the approaches advocated here. The prototypes are also of interest because of the lessons they provide about tool-making.

1.5 Outline of Thesis

The rest of the thesis is structured as follows. Chapter 2 gives a brief overview of verification, and then reviews related literature. This raises the important issues and problems of verification, motivates choices made in this research, and places the research into context.

Chapter 3 presents the four-valued logic $\mathcal{Q}$, and the temporal logic, TL, based on $\mathcal{Q}$. After defining the logics, the issue of satisfaction — what it means to say that a certain property holds of a state or sequence of states — is explored and different alternatives given.

The theory of generalised symbolic trajectory evaluation is given in Chapter 4 using the theory presented in Chapter 3. Although the theory of trajectory evaluation is general, at this stage its major application area is circuit verification.

Chapter 5 develops the theory of composition for the verification of partially-ordered state spaces. The compositional inference rules are explained, and shown to be sound. Simple examples are given. The compositional theory is very important in increasing the range of systems that can be verified using trajectory evaluation.

Chapter 6 ties the preceding chapters together and shows how the theory can be practically implemented. Issues of data and state representation are discussed, practical model checking algorithms based on STE outlined, as well as how the verification style of model checkers and theorem provers can be combined.

Chapter 7 is devoted to example verifications. A few simple verification examples are given to show the style of verification, and then some large verification examples are given. This chapter shows that the methodology proposed here can be effectively implemented.

Chapter 8 is a conclusion, and the appendix contains some of the more technical proofs and example programs.

A Guide to the Reader

This thesis contains many definitions, theorems and a significant level of mathematical notation. A reader may find the index at the end of the thesis and the List of Important Definitions, Theorems and Lemmas starting at page ix useful in finding cross-references.

The nature of the research requires that the thesis contain many proofs. Many of these proofs are highly technical and uninteresting in themselves; this does not make for the easiest or most captivating reading, for which I apologise. I have tried to make the exposition of proofs as clear as possible, and have adopted the following convention for proofs. Each step in the proof contains three parts, laid out in three columns: a label, a claim, and a justification. The justification may refer to previous steps in the proof using the labels given.

Lemma 1.1 (Example).

If $\mathcal{S}$ is a complete lattice under the partial order $\sqsubseteq$, and $g : \mathcal{S} \to \mathcal{S}$ is monotonic with respect to $\sqsubseteq$, then for all $s, t \in \mathcal{S}$, $g(s) \sqcup g(t) \sqsubseteq g(s \sqcup t)$.

Proof.

(1)	$s \sqsubseteq s \sqcup t$	Definition of join.
(2)	$t \sqsubseteq s \sqcup t$	Definition of join.
(3)	$g(s) \sqsubseteq g(s \sqcup t)$	From (1) by monotonicity of g
(4)	$g(t) \sqsubseteq g(s \sqcup t)$	From (2) by monotonicity of g
(5)	$g(s) \sqcup g(t) \sqsubseteq g(s \sqcup t)$	From (3), (4) by property of join

□

Chapter 2

Issues in Verification

This chapter is intended to place the thesis work in perspective and relate the research to other work. It is not intended as a comprehensive survey of verification, and therefore some simplifications are made and important verification methods skimmed over. For fuller surveys on the topic see [73, 97, 119].

Overview of Chapter

Section 2.1 briefly introduces a method of representing boolean functions. Since boolean expressions are used extensively in verification for a variety of purposes, efficient methods for representing and manipulating them is essential.

The review of verification starts with Section 2.2, which introduces two of the main styles of verification. In one style, verifying a model means checking whether the model has certain properties. In the other method, two models are compared to see whether a certain relationship holds between the models (for example whether they have equivalent observable behaviour).

For each of the styles of verification, there are a number of possible verification techniques. Section 2.3 gives a brief overview of some of the large number of verification techniques, differing in approach and detail, that have been proposed. Section 2.4 examines one of these proof techniques in more detail; the method of symbolic trajectory evaluation proposed by Bryant and Seger forms the basis of this thesis.

Due to the computational complexity of verification, all these methods have limitations, and research continues in trying to improve upon them. Much of this research in improving verification techniques deals with the search for better algorithms and data structures. Although this is very important, the underlying complexity limitations indicate that something more is needed. Two of the most promising lines of research in this regard have been the work in compositionality and abstraction. They are discussed in Sections 2.5 and 2.6.

Section 2.7 concludes the review with a brief discussion of the issues raised in this chapter.

2.1 Binary Decision Diagrams

Many verification techniques — including STE — represent boolean expressions with a data structure called (ordered) Binary Decision Diagrams (BDDs). BDDs are a compact, canonical method for manipulation of boolean expressions [22]. A BDD is a directed, acyclic graph, with internal vertices representing the variables appearing in the expression. BDDs are ordered in the sense that on all paths in the graph, variables appear in the same order.

Using this representation, operations such as conjunction, negation and quantification and equivalence testing can be efficiently implemented.

Boolean expressions are used to represent state information and truth of propositions, so it is critical that they can be manipulated efficiently. BDD-based approaches have been extremely successful. Unfortunately, although BDDs are a very compact representation, there are things that cannot be represented efficiently. This is not surprising; the satisfaction problem [63] can be represented and solved using BDDs so if a BDD representation polynomial in size could be constructed in polynomial time, this would imply that P=NP. Some arithmetic problems cannot be represented efficiently. For example, multiplication of two integers (represented as bit-vectors) requires BDDs exponential in size. For a discussion of the limitations of BDDs, see [21].

One of the critical issues when BDDs are used is the ordering of variables used in the construction of the graph. The size of the resulting BDD may be highly dependent on the variable ordering, so it is vital that a good variable ordering is used. In general human intervention is needed to determine a good ordering, but fortunately for many real problems a good variable ordering can be found, and heuristics for dynamic variable ordering can be applied successfully. So, while the need to find good variable orderings is an issue, it is not a fundamental problem with BDD-based methods.

Although BDD-based approaches are very successful, there are a number of other successful approaches that do not use BDDs. Some of these are described below. The success of BDDs has also motivated research on other data structures for representing data that BDDs cannot represent efficiently (these are mentioned below too).

2.2 Styles of Verification

To say that a program or circuit is verified is to say that there is a proof that certain mathematical statements are true of a model of that system. This section looks at the different ways of expressing these statements, while Section 2.3 looks at how the statements are proved to hold.

Section 2.2.1 introduces the property checking approach. The idea here is that there is a formal language for expressing properties of the system, and verification consists of proving that these properties hold. Section 2.2.2 leads on from this by introducing modal and temporal logics; these are logics that are commonly used to express properties of interest. This is the style of verification adopted in this thesis.

Section 2.2.3 introduces the other style of verification, model comparison. Here, two models of the program or circuit are expressed formally (typically, a specification and an implementation), and verification consists of proving that the two models are equivalent.

2.2.1 Property Checking

One major approach to verification is to determine whether a description of a program[1] has (or does not have) a set of properties. Turing showed that this problem — one of the foundational problems in computer science — is, in general, undecidable: for example, there is no general method for determining whether a program halts, or whether it prints out a zero [124].

One of the landmarks in program verification was the development of the Floyd-Hoare logic used to describe the behaviour of sequential programs (introduced in [82], and see [67] for a good introduction). In this logic, verification results are written in the form $\{A\}P\{C\}$, where A and C are logical formulas and P is the program segment. This *Hoare triple* says that if A holds when P starts executing, then if P completes then C will hold. There have been many other approaches used to describe the behaviour of sequential (see [7] as a good example of this style of proof) and concurrent programs (see [96] for an example).

This style of verification can be used for small programs, and can be appropriate for small, complicated algorithms. However, on a larger scale it is not useful as it is just too tedious to use, especially for hardware systems.

There are many ways of expressing properties. For example, one approach has been to perform reachability analysis on the program (e.g. discovering whether there are any deadlock states). Another approach — the one adopted in this research — is to use some form of logic to express properties. Often modal or temporal logics are used for reactive systems.

2.2.2 Modal and Temporal Logics

Modal logics are systems of logic for describing and reasoning about contingent truths. The type of modal and temporal logics of interest here are used to describe the behaviour of systems

[1] As the term 'system' can be ambiguous since it can refer to the system being verified, or the tool performing verification, the term 'program' is used in a generic sense to describe the system being verified, whether or not the system is represented as a program, a finite state machine, a netlist etc.

that have dynamic or evolving structure. Many of these logics have been proposed; see the works of Galton [62], Emerson [52] and Stirling [120, 121] for overviews.

Typically, a set of formulas of the logic form the specification of the program being verified. The verification task is to test whether the mathematical structure representing the program satisfies this set of formulas.

The wide variety of modal logics reflects both the wide variety of application and complexity of the topic. Modal logics and the mathematical structures over which they are interpreted differ in expressiveness. Issues such as non-determinism and the ability to express recurring properties greatly affect issues such as usefulness, decidability and computational complexity.

Temporal logics are particularly useful in verification. They can be used to specify the behaviour of a system over time. Time can be a 'real' time, or some abstraction thereof; and can also be modelled as continuous or discrete. The method proposed in this thesis has the advantage of being able to model time fairly accurately.

The most powerful logic of interest here is the family of modal μ-calculi, variously attributed to Park and Kozen. The expressive power of the μ-calculi, determined by the modal operators available, have a marked effect on the decidability of logic: for example, the linear time modal μ-calculus is decidable, while the branching time modal μ-calculus is not [55].

Other modal and temporal logics are restricted versions of the μ-calculus. CTL*, CTL, LTL, and the Hennessy-Milner logic are good examples of logics which can be encoded within a version of the μ-calculus. There are a number of ways in which temporal logics can be classified (see [52] for details). The most important question is whether the logic is branching time or linear time (see [53] for some discussion of this).

2.2.3 Model Comparison

In this approach, two models or descriptions are compared to see whether a given relationship holds between them. The most important relationship is equivalence, but there are other useful relationships. One way of using this form of checking is for one description to be a specification of a system and the other description to be an implementation. Showing that a formal relationship holds between the two descriptions shows that the implementation is correct.

General versions of this problem are undecidable. Turing machine equivalence is the best example, and these decidability results apply to popular methods such as process algebras (as CCS can encode Turing machines this has direct relevance to much work in this area). However, there are restricted, useful versions of the problem which are decidable (see, for example [32]).

There are a number of different ways in which models can be represented. What equivalence and more general types of relationships mean and how they are checked depends very heavily on this. Three of the main approaches are:

1. Process algebras such as CCS [102] and CSP [20]. There are many different types of equivalence which depend on how fine-grained an equivalence is desired (see [102] for a discussion of this).

   There are a number of other relationships which are defined as preorders on processes. These can be used to define correct implementations of specifications. See [80] for examples.

   A good example of this approach is the LOTOS specification language which is based on CCS and CSP [15]. Equivalence and implementation relationships can be used to show that one LOTOS program is a correct implementation of another.

2. Language containment. If the descriptions are finite state machines, then equivalence

may be language equivalence. Some verification problems can be posed as language containment problems. See [73] for an overview.

3. Logic. Equivalence is logical equivalence. Other logical relationships such as implication may be suitable for showing that a model is a correct implementation of a specification. See [94] for an example.

Other approaches exist (e.g. [74]).

There is a close relation between equivalence checking and property checking. In CCS, two processes are bisimilar exactly when they satisfy the same set of formulas of the Hennessy-Milner logic [81]. Grumberg and Kurshan have shown a relationship between classes of CTL* formulas and language equivalence or containment problems [71].

2.3 Proof Techniques

Now that we have defined what we mean by program correctness, we can examine proof techniques. We first look at why formal proof techniques are important, and then examine some of these techniques: Section 2.3.1 discusses theorem proving; Section 2.3.2 discusses automatic techniques suitable for proving equivalences; and Section 2.3.3 discusses model checking, an approach that can be used to prove that models of systems satisfy temporal logic formulas. Section 2.4 presents the model checking that forms the basis of this thesis in more detail.

Hand proof techniques are the most ubiquitous for a variety of reasons. They are powerful methods which allow a variety of proof techniques, appropriate informal arguments, and abstractions to be made. However, there are two important reasons why hand proofs are avoided in the context of verification, particularly hardware verification.

First, proofs are extremely tedious to perform. Often they are not complex but have large amounts of intricate detail which is difficult and unpleasant for humans to keep track of. Second, errors are extremely likely. These two factors are related.

Some examples illustrate this. In [102], Milner presents the 'jobshop' example — a specification, implementation and a proof of weak bisimulation between the two. The proof has many errors. Most of these errors are trivial; however, there is a serious theoretical error on which the proof relies which was only detected much later. It must be emphasised that this is a best case scenario for hand proofs: the models and notation are fairly abstract, the proofs fairly short and interesting in themselves, and the person making the proof of undoubted mathematical ability.[2]

There are many other examples like this; they show how fallible and time-consuming hand proofs are (see [112]). The alternative to hand proofs are machine checked and automated proofs. A machine checked proof is a proof that has each step validated by a program that implements some logical inference system. An automated proof is one which is generated without human intervention according to some set of sound rules. Often the performance of these systems may depend on extra information given by the human verifier. These approaches have been applied to both the equivalence and property-checking types of problems.

2.3.1 Theorem Proving

A *theorem prover* is a program that implements a formal logic. Using this program, statements in the logical system can be proved. Typically, the logical system consists of a set of axioms and inference rules, and the program ensures that all theorems are sound in that they are derived from the axioms by application of the inference rules. Although much work has gone into automatic theorem proving the key aspect is mechanically checking each step rather than the automatic derivation of the proof.

Theorem provers can be used to prove theorems about any mathematical system. Within the

[2]For hardware verification the converse is true: the level of abstraction is low with intricate detail, the proofs are long and tedious and of no intrinsic interest, and few people doing verification are Turing Award winners. This criticism extends to other domains too. In a recent paper, Bezem and Groote present the verification of a network protocol in a recent paper [13]. The proof is very lengthy and detailed. The claim that this is not such a problem because the proofs are 'trivial' is unconvincing.

verification area, there is a strong theorem proving community and a range of different theorem provers have been used in verification tasks. Some examples of theorem provers and work done with theorem provers are:

- HOL [68]. HOL is one of the first and best known theorem provers. It was built on work done on the development of LCF [69] in the 1970s (see [60] for a brief history). HOL implements a strongly typed higher-order predicate logic. The user's interface to HOL is through ML [107], a polymorphic, typed functional language. This interface promotes both security (by ensuring through the type system that only theorems proven in HOL can be proved) and flexibility by allowing the programmer access to a fully programmable script language. HOL has been used on the verification of a number of systems.

- Boyer-Moore [17]. This theorem prover is based on a quantifier-free first order logic. It is heavily automated, although a user can (must?) 'train' the prover to deal with particular proofs. An example of a substantial verification effort using this system can be found in [86].

- PVS [106, 105] is also theorem prover based on a typed, higher-order logic. It has a number of decision procedures built in which allow a number of the proof obligations to be discharged automatically. See [112] for an example use of PVS.

Theorem provers can be used for either equivalence or property checking. For example, if both the specification, S, and implementation, I, are logical formulas then asking whether S and I are equivalent means asking whether $S \equiv I$ is a theorem in the logic.

In the property checking approach, Gordon shows how a simple theorem prover can be used to prove program correctness using the Floyd-Hoare logic [67]. Theorem provers have also been used in model checking; some work is directly relevant to this thesis. For example, Bradfield describes a 'proof assistant' for model checking μ-calculus formulas over Petri nets. The proof system is a tableau-based one (see below). At each step in the proof either the prover itself applies a proof rule, or the user does [18].

Sometimes, this type of approach leads to a hybrid system which uses both the automatic model checking algorithms described below and theorem proving approaches – this is discussed in more detail later.

2.3.2 Automatic Equivalence and Other Testing

For certain systems which can be represented as labelled transition systems (such as certain classes of CCS agents), the Concurrency Workbench has algorithms for computing different kinds of equivalences and preorders [41]. The two advantages of using equivalences such as bisimulation over language equivalence are:

- Bisimulation can distinguish behaviour which language equivalence can not.
- There are significant computational advantages. For example, deciding regular language equivalence is PSPACE-complete, while the best known algorithm for deciding bisimulation between two regular processes is $O(m \log n)$ where m is the number of transitions in the process and n is the number of states [103].

For finite state systems, CCS agents can be represented using BDDs [54], from which equivalence relations can be computed [27]. Other work in this line includes a tool which can compute equivalence of LOTOS programs [56].

Other approaches can also be applied to transition systems, see ([1, 14]).

2.3.3 Model Checking

Given a model of a system behaviour, M, and a temporal logic formula g interpreted over M, the model checking problem is to find out whether g holds of M, or whether M is a model of

g. Typically this is written as $M \models g$. Variations such as finding whether a set of states or a set of sequences of states satisfies the formula are used too.

Model checking is a difficult problem: some useful versions are undecidable [55], and the satisfiability and model checking problems for even simple modal logics are NP-hard [63, 52]. For finite state systems, whether a structure is a model of a system can be determined directly from the satisfaction relation by explicit state enumeration: however, except for small systems this is rarely feasible.

Tableau-based Methods

The tableau-based method is one of the best-known methods and a number of variations have been implemented (the best known implementation is the Concurrency Workbench [41]). Although the underlying proof method is very different to the method of symbolic trajectory evaluation, this method is of some relevance because tableau systems use rules of inference, and because there has been much work in compositional reasoning. Good introductions to the tableau method are [19, 121]. Note that tableau methods do not always require the construction of the global state space.

A tableau is a proof tree built from a root sequent of the form $S \vdash \Phi$ (this is the goal sequent). The tree is built using one of the tableau rules until all the leaves of the tree are terminals. If all the terminals are 'successful' then $S \models \Phi$. The most important and difficult part of the tableau construction is dealing with the fixed-point operators, particularly the least fixed point operator.

Stirling and Walker proposed a sound and complete tableau system for finite-state processes [122]. They showed that the tableau-construction always terminates, making this method an effective model checking scheme. They also show how the model checking algorithm of Winskel [127] can be incorporated in a tableau scheme.

Bradfield extended the tableau approach to infinite state systems [19]. Dealing with the fixed point operators is more complicated as the definition of a successful terminal takes some care. His approach is sound and complete. If $S \models \Phi$, then using the rules automatically will derive $S \vdash \Phi$, and $S \vdash \Phi$ will only be derived when $S \models \Phi$. Note, however, that if $S \not\models \Phi$, his algorithm may not terminate.

Automatic Model Checking through State Exploration

For finite state systems, it is feasible to model check some logics through state exploration methods. Although model checking expressive temporal logics such as CTL* is very expensive (the problem is PSPACE-complete), for less expressive logics there are better results (note that model checking LTL is also PSPACE-complete). The best known result is one for model checking the subset of CTL* known as CTL [36]. This algorithm works by building the state transition graph, and then using graph algorithms to label the states in the graph. The algorithm is $O(|S||\phi|)$ where $|S|$ is the number of states in the system, and $|\phi|$ is the size of the formula ϕ. Recently this work has been extended to show how a richer logic CTL^2 can be model checked with the same complexity result [12].

Although the algorithm is linear in the size of the state space, this is a significant limitation since the size of the state space in many realistic systems is extremely large (a very small circuit with only 100 state holding components can have a reachable state space of size 2^{100}).

State exploration methods can be extended to some types of infinite systems. Burkart and Steffen have developed a state exploration method for effective model checking of the alternation-free μ-calculus for context-free processes [29]. (A local model checking version based on tableaux has also been developed [85]).

Symbolic Model Checking

For finite systems symbolic model checking methods are very popular and have had success in a number of applications. The use of BDDs has revolutionised model checking by providing a compact method for implicit state representation, thereby increasing by orders of magnitude the size of the state space that can be dealt with. (Other approaches exist too [14, 46]: however BDDs seem to be most effective for a large class of problems.)

The most well-known work based on symbolic model checking and BDDs has emerged from Carnegie Mellon University. A number of model checking algorithms for the modal μ-calculus and other logics have been developed [26, 27]. The SMV verification system based on these ideas has successfully verified a range of systems [26, 98].

The basic idea of these approaches is to represent the transition relation of the system under consideration with a BDD. A set of states is also represented with a BDD. Given a formula of the temporal logic, the model checking task is to compute the set of states that satisfy the formula. The operations defined on BDDs allow the computation of operations such as existential quantification, conjunction etc. Using these BDD operations, it is possible to compute the set of reachable states and the set of states satisfying a given formula.

Although these methods have had some success, the computational complexity and cost of model checking remains a significant stumbling block. Symbolic CTL model checking is PSPACE-complete in the number of variables needed to encode the state space [98]. A number of approaches have been suggested to improve the performance of the algorithm: compositional approaches; abstraction; and improving representational methods (for example, partitioning the next state relation [26]).

That BDDs revolutionised automatic model checking indicates the importance of good and appropriate data structures, and motivates the search for new ones, and considerable work is being done on extending BDD-style structures and developing new ones [24, 34, 99]).[3]

All these approaches to improve symbolic model checking need to be pursued. Circuits with wide data paths are not suitable for verification with SMV, which itself is unable to verify circuits with arithmetic data. However, by extending the method through the use of abstraction [39] or more sophisticated data structures [35] such circuits can be verified.

There are other symbolic model checking approaches. Symbolic trajectory evaluation — a central part of this thesis — is one. It differs from other approaches in the novel way in which the state space is represented. Although the logic which it supports is limited it has been successfully used in hardware verification [8, 47]. Full details are given later. Other symbolic methods have been proposed in [16, 43, 87].

Combining Theorem Proving and Model Checking

Since combining model checking and theorem proving has considerable promise, research has been done in combining the two approaches in different technical frameworks.

Seger and Joyce linked the HOL and Voss systems. This allows the HOL theorem proving system to reason about properties of a circuit by using the model checking facilities of Voss [117]. Although there are some similarities between the prototypes presented in this thesis, and the HOL-Voss system, there are two important distinctions:

- One of the important uses of a theorem prover with the Voss system is to reason about objects that do not have concise BDD representations in all cases — for example, integer expressions. Rather than providing a general and powerful theorem prover such as HOL, simple semi-automated methods are used to provide the prototypes the ability to do this (see Section 6.2). Although not as powerful as HOL, it is much simpler.
- The prototypes provide specialised theorem provers that implement a compositional theory for STE. The use of this compositional theory increases the power of the verification

[3] Some of these approaches are applicable to other model checking approaches too.

approach significantly.

Kurshan and Lamport have combined the COSPAN model checker with the TLP theorem prover [93]. The model checker proves properties of components of the system, which are then translated into a form suitable for the theorem prover. In order to prove the overall result, a number of sub-results need to be proved. Not only is the way in which composition is handled different to the way it is in this thesis, there are also two very important practical distinctions: first, their approach is not entirely mechanised; second their approach relies on linking two quite distinct tools and using two distinct formalisms, rather than one integrated tool and verification style.

The style of the method of Hungar [84], who also links model checking and theorem proving, is closest to the method of combining model checking and theorem proving proposed in this thesis. The model is given by a Kripke structure representing the semantics of an Occam program, and the properties are expressed in a variant of CTL. The results generated by model checking are combined using the LAMBDA theorem prover. The proof system consists of rules for inferring results using an assume-guarantee style of reasoning. The inference rules used are: embedding, modus ponens, conjunction and weakening. Given an Occam program consisting of a number of processes, properties can be proven of each process using the model checker, and the properties combined.

An important distinction between the model used in [84] and the model used in this thesis is that in Hungar's framework, each process has its own model — the model for the entire program is the composition of these models. In the compositional theory proposed in this thesis, a model is given for the entire system, and it is not necessary to give a model for the components of the system.

Rajan *et al.* have combined a μ-calculus model checker with PVS by using the model checker as a decision procedure for PVS [111]. They demonstrate how such an integrated system can be used. Using the ideas of Clarke *et al.* discussed below, they create an abstraction of a circuit to be verified. Using theorem proving they show that the abstraction has the required properties. Using model checking they show that the abstraction satisfies the specification.

In an alternative approach, Dingel and Filkorn verify abstractions of a system using model checking, using certain assumptions about the system environment [49]. Theorem proving is used to prove the correctness of the abstraction and to ensure that the system environment assumptions are met.

2.4 Symbolic Trajectory Evaluation

This section briefly outlines the existing STE based approach. This is useful in the later discussion and will help illustrate some of the novel aspects of the thesis. Symbolic trajectory evaluation was first proposed in [23] and the full theory can be found in [116]. Good examples of verification using STE can be found in [8, 47]. This section is heavily based on the presentation of STE found in [77].

The model of a system is simple and general, a tuple $\mathcal{M} = (\langle \mathcal{S}, \sqsubseteq \rangle, \mathbf{Y})$, where $\langle \mathcal{S}, \sqsubseteq \rangle$ is a complete lattice ($\mathcal{S}$ being the state space and $\sqsubseteq$ a partial order on $\mathcal{S}$) and $\mathbf{Y}$ is a monotone successor function $Y\colon \mathcal{S} \to \mathcal{S}$. A sequence is a *trajectory* if and only if $\mathbf{Y}(\sigma^i) \sqsubseteq \sigma^{i+1}$ for $i \geq 0$.

2.4.1 Trajectory formulas

The key to the efficiency of trajectory evaluation is the restricted language that can be used to phrase questions about the model structure. The basic specification language used is very simple, but expressive enough to capture many of the properties we need to check.

A *predicate* over $\mathcal{S}$ is a mapping from $\mathcal{S}$ to the lattice $\{\mathtt{F}, \mathtt{T}\}$ (where $\mathtt{F} \sqsubseteq \mathtt{T}$). Informally, a predicate describes a potential state of the system: e.g., a predicate might be $(A$ **is** $x)$ which says that node A has the value x. A predicate is *simple* if it is monotone and there is a unique weakest $s \in \mathcal{S}$ for which $p(s) = \mathtt{T}$. TF, the set of *trajectory formulas* is defined recursively as:

1. **Simple predicates**: Every simple predicate over $\mathcal{S}$ is a trajectory formula. Simple predicates are used to describe simple, instantaneous properties of the model.

2. **Conjunction**: $(F_1 \wedge F_2)$ is a trajectory formula if F_1 and F_2 are trajectory formulas. Conjunction allows the combination of formulas expressing simpler properties into a formula expressing a more complex property.

3. **Domain restriction**: $(e \rightarrow F)$ is a trajectory formula if F is a trajectory formula and e is a boolean expression over a set of boolean variables, $\mathcal{V}$. Through the use of boolean variables, a large number of scalar formulas (formulas not containing variables) can be concisely encoded into one symbolic formula.

4. **Next time**: $(\mathbf{N}F)$ is a trajectory formula if F is a trajectory formula. Using the next time operator allows the expression of properties that evolve over time.

An interpretation of variables is a function, $\phi : \mathcal{V} \rightarrow \{\mathtt{F}, \mathtt{T}\}$. An interpretation of variables can be extended inductively to be an interpretation of expressions. The truth semantics of a trajectory formula is defined relative to a model structure, a trajectory, and an interpretation, ϕ.

Whether a sequence $\tilde{\sigma}$ satisfies a formula F (written as $\tilde{\sigma} \models F$) is given by the following rules.

1. $\sigma_0\tilde{\sigma} \models p$ iff $p(\sigma_0) = \mathtt{T}$.
2. $\sigma \models (F_1 \wedge F_2)$ iff $\sigma \models F_1$ and $\sigma \models F_2$
3. $\sigma \models (e \rightarrow F)$ iff $\phi(e) \Rightarrow (\sigma \models F)$, for all interpretations, ϕ.
4. $\sigma^0\tilde{\sigma} \models \mathbf{N}F$ iff $\tilde{\sigma} \models F$.

Given a formula F there is a unique *defining sequence*, δ_F, which is the weakest sequence that satisfies the formula.[4] The defining sequence can usually be computed very efficiently. From δ_F a unique *defining trajectory*, τ_F, can be computed (often efficiently). This is the weakest trajectory which satisfies the formula — all trajectories which satisfy the formula must be greater than it in terms of the partial order.

If the main verification task can be phrased in terms of 'for every trajectory σ that satisfies the trajectory formula A, verify that the trajectory also satisfies the formula C', verification can be carried out by computing the defining trajectory for the formula A and checking that the formula C holds for this trajectory. Such results are called *trajectory assertions* and we write them as $\models \langle\!\langle A \Longrightarrow\!\!\!> C \rangle\!\rangle$. The fundamental result of STE is given below.

Theorem 2.1.
Assume A and C are two trajectory formulas. Let τ_A be the defining trajectory for formula A and let δ_C be the defining sequence for formula C. Then $\models \langle\!\langle A \Longrightarrow\!\!\!> C \rangle\!\rangle$ iff $\delta_C \sqsubseteq \tau_A$.$\Box$

A key reason why STE is an efficient verification method is that the cost of performing STE is more dependent on the size of the formula being checked than the size of the system model. STE uses BDDs for manipulation of boolean expressions.

2.5 Compositional Reasoning

The main problem with model checking is the state explosion problem — the state space grows exponentially with system size. Two methods have some popularity in attacking this problem: compositional methods and abstraction. While they cannot solve the problem in general, they do offer significant improvements in performance.

Compositional reasoning is a critical aspect of program verification. The following advantages are stated in [5]:

[4] 'Weakest' is defined in terms of the partial order.

- Modularity: if a module of a system is replaced, only the module need be verified;
- In design or synthesis it is possible to have undefined parts of a system and still be able to reason about it;
- By decomposing the verification task, verification can be made simpler;
- Re-use of verification results is promoted.

The difficulty of compositional reasoning is that often it is the case that a particular component may not have a property that we desire of it when placed in a general environment. However, when placed in the context of the rest of the system, then it does display the property. See [3] for some discussion of the issues involved in this type of reasoning.

For tableau-based methods, a number of approaches have been suggested. Andersen *et al.* have proposed a proof system for determining the whether processes of a general process algebra [5] satisfy a formula. They show that a set of 39 inference rules is sound, and – for a class of finite-state systems – is complete. Although this is an important contribution, it is difficult to assess the impact of this work without substantive examples. Furthermore, to be practical I believe the proof system needs some form of mechanical assistance.

In related work, Berezine has proposed two model checking algorithms for fragments of the μ-calculus [11] (here model checking asks whether $p \models \Phi$ — does the process p satisfy Φ). Both methods can be used to verify problems of the form $p \times q \models \Phi$, where $p \times q$ represents the composition of processes p and q. The first takes the problem and constructs a formula Φ_p such that $q \models \Phi_p$ iff $p \times q \models \Phi$. The second constructs two formulas Φ_p and Φ_q such that $p \times q \models \Phi$ iff $q \models \Phi_p$ and $p \models \Phi_q$. As the work is preliminary, it is difficult to assess the applicability and effectiveness of this approach.

Compositional techniques have been proposed for symbolic model checking. Clarke *et al.* have proposed a method for systems of concurrent processes [40]. To model check $P \| E \models \phi$ may not be computationally feasible (where P represents a process of interest, and E represents the environment). They show how an *interface* process, A, can be constructed such that under certain conditions $P\|E \models \phi$ if $P\|A \models \phi$. The point of this is that the state graph of $P\|A$ can be considerably smaller that of $P\|E$. Although this method has theoretical interest and there are examples of systems for which it works, it has not been established how applicable this method is, and how easy (in terms of human and computation cost) it is to establish the conditions for correct application.

Another approach to compositional reasoning — modular verification — is based on defining a preorder relation,$\preceq$, between models [72, 95]. This preorder is based on a simulation relationship between the models and has the property that if $M_1 \preceq M_2$ and $M_2 \models \phi$ then $M_1 \models \phi$. Suppose we wish to show that a process M when placed in its environment satisfies a property ϕ. While M may not in general satisfy ϕ, it may satisfy it whenever its environment satisfies another property ψ. Given the formula ψ, there exists a 'tableau' M_ψ which is the strongest element in the preorder which satisfies ψ. If $E \preceq M_\psi$ and $M\|M_\psi \models \phi$, then by the property of the preorder, $M\|E \models \phi$. The verification therefore includes proving the simulation relation and performing model checking. Both of these steps are automatic, using symbolic algorithms. This method is only applicable to finite state systems.

Aziz *et al.* propose a compositional method dependent on the formula being checked [6]. The model is represented as a composition of state machines. Given a formula to be checked, an equivalence relation is computed for each machine which preserves the truth of the formula. Using these equivalence relations, quotient machines are constructed and the composition of these machines computed. This composition will have a smaller state space than the original composition and can be used to determined the correctness of the formula.

Other compositional approaches exist too. Some of these focus on the question of the refinement of a specification into an implementation. They tend to use hand proofs. Examples of other approaches include [3, 89, 93].

2.6 Abstraction

The idea behind abstraction is that instead of verifying property f of model M, we verify property f_A of model M_A and the answer we get helps us answer the original problem. The system M_A is an abstraction of the system M.

One possibility is for the abstraction M_A to be equivalent (e.g. bisimilar) to M. This sometimes leads to performance advantages if the state space of M_A is smaller than M, but usually this type of abstraction is used in model comparison (e.g. as in [74]).

Typically, the behaviour of an abstraction is not equivalent to the underlying model. The abstractions are *conservative* in that M_A satisfies f_A implies that M satisfies f (but not necessarily the converse). Some examples of abstraction methods are [50, 70, 83, 95].

In hardware verification, abstraction is particularly needed in dealing with the data path of circuits. A drawback of abstraction is that it takes effort to both come up with the suitable abstraction (see [37, 123]) and prove that the abstraction is conservative. For an example of this type of proof see [28].

Clarke *et al.* define abstractions and approximations [39]. They show how an approximation can be abstracted from the program text without having to construct the model of the system. They provide a number of possible abstractions: congruence modulo an integer (the use of the Chinese remainder theorem); representation by logarithm; single-bit and product abstraction; and symbolic abstraction. They show how this is used on a number of examples.

2.7 Discussion

Although equivalence checking is also attractive, this thesis explores one model checking because of its success in verifying large state spaces. Moreover, in some situations it is not appropriate or possible to have a formal model to compare an implementation against (although work such as [130] offers some ideas in how such a model could be built from a set of properties).

Theorem provers and model checkers both have strong adherents because both methods have had successes. However, they both have weaknesses. Automatic verification techniques have the advantage of being automated, but have limitations on the size of the systems that they can deal with, and theorem proving methods, while very powerful, are still computationally intensive and require a great deal of skill. Work such as [77, 84, 93, 117] among others shows that there is much to be gained from combining the approaches.

The vision adopted in this research is that symbolic model checking is used to prove low-level properties of the system which would be very tedious for the theorem prover, while the theorem prover — partly automated — is used to prove higher-level properties.

Efficient model checking is very important. Although tableau-based methods are powerful and attractive in some situations, BDD-based methods are more appropriate for finite state systems, especially VLSI circuits. Although progress has been made, much work remains to be done to improve performance by examining issues such as abstraction, composition and methods for state and transition relation representation.

There are many different criteria for evaluating verification methods, depending on application and setting (for some discussion of this, see [119]). Three criteria for evaluating the approaches discussed above are:

1. Range of application;
2. Performance;
3. Degree of automation/ease of use.

For a fuller discussion of the use of verification methods in industry, see [119].

A problem with this area is that because verification is very difficult, methods tend to be suited for particular applications. Often it is difficult to compare approaches because they solve different problems. The types of properties to be checked for and the way in which the model is represented are critical. For example, verifying multiplier circuitry modelled at the switch level where timing is critical is a very different proposition to dealing with a very high level description of an algorithm where timing is not an issue.

Furthermore, because many of these problems are so difficult (i.e. are NP-hard) analytic categorisations of different algorithms are not always very useful. Empirical results are also difficult to analyse since verifications are run with systems on different hardware architectures and written in different languages. Particularly difficult to measure is how easy the verification method is to use (how automatic is an automatic verification method) for different classes of user.

Many examples in the literature give a few examples but fail to give convincing evidence that the method will work on a larger class of problems.

All of this is exacerbated by a lack of published empirical results. Work with detailed performance figures is available (such as [26]) but important theoretical contributions such as [5, 72] come with no performance results and only small examples to illustrate the applicability of the method.

The importance of gaining more experimental results has been recognised ([26] is a good example), and the IFIP working group on hardware verification has recently established a benchmark suite to help facilitate comparative work [91]. Chapter 7 presents some experimental data in order to evaluate the methods proposed in this thesis.

Chapter 3

The Temporal Logic TL

This chapter introduces and defines the quaternary temporal logic at the core of the research. Section 3.1 describes the model over which formulas of the logic are interpreted: a complete lattice is used to represent the set of instantaneous states, and a monotonic next state function is used to represent system behaviour. This gives a way of formally describing an implementation of a system such as a VLSI design. Section 3.2 defines $\mathcal{Q}$, a quaternary logic, and proves elementary properties of this logic. Using $\mathcal{Q}$ as a base, the quaternary temporal logic TL is defined in Section 3.3. The syntax of TL formulas is given; the truth of these formulas is defined with respect to sequences of states of the model. TL gives a way of describing intended behaviour.

The primary application of the theory presented in this thesis is for circuit models. For convenience, and as these models have useful properties, it is appropriate to specialise the temporal logic for circuit models. This is discussed in Section 3.4.

A critical question is whether the model satisfies the intended behaviour. Section 3.5 is a precursor for the discussion of this question in Chapter 4 by presenting alternative semantics for TL; these semantics illustrate the idea on which the model checking approach of Chapter 4 is based.

3.1 The Model Structure

The *model structure* $(\langle \mathcal{S}, \sqsubseteq \rangle, \mathcal{R}, \mathbf{Y})$ represents the system under consideration.

- $\mathcal{S}$, a complete lattice under the information ordering $\sqsubseteq$, represents the state space. Let X be the least element in $\mathcal{S}$. (When $\mathcal{S} = \mathcal{C}^n$, then $\mathsf{X} = \mathsf{U}^n$.)

- $\mathcal{R} \subseteq \mathcal{S}$, the set of *realisable states*, represents those states which correspond to states the system could actually attain — $\mathcal{S} - \mathcal{R}$ are the 'inconsistent' states, which arise as artifacts of the verification process. Which states are realisable and which are inconsistent is entirely up to the intuition of the modeller; the entire state space could be realisable, or only part of it.

  Verification conditions will be of the form: do sequences that satisfy g also satisfy h? Distinguishing unrealisable behaviour from realisable behaviour allows the detection of cases where verification conditions are vacuously satisfied: if it is the case that no sequences with only realisable states satisfies g then the verification condition may indeed by satisfied. However, it is likely that either the specification or implementation are wrong.

  On the other hand, it may be that for all sequences of realisable states the verification conditions are satisfied, but that some sequences with unrealisable behaviour satisfy g but do not satisfy h. If we consider the set of all sequences, the verification condition will fail; if we consider only the sequences of realisable states the verification conditions succeed.

  Thus, the concept of realisability allows the modeller to deal with inconsistent information in a sensible way: detecting vacuous results and ignoring degenerate cases.

  There is a technical requirement: $\mathcal{R}$ must be downward closed, so that if $x \in \mathcal{R}$, and $y \sqsubseteq x$ then $y \in \mathcal{R}$. This makes computation much easier and has a sound intuitive basis. Intuitively, if a state is not realisable, it is because it is 'inconsistent'; any state above it in the information ordering must be even more 'inconsistent' and thus also not realisable. Conversely, if a state is 'consistent', then a state below it in the information ordering will

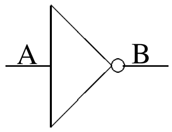


Figure 3.1: Inverter Circuit

also be 'consistent'.

- $\mathbf{Y} : \mathcal{S} \to \mathcal{S}$ is a monotonic next state function: if $s \sqsubseteq t$ then $\mathbf{Y}(s) \sqsubseteq \mathbf{Y}(t)$.

  Although the next state function is inherently deterministic, the partial-order structure of the state space can model non-determinism to some extent. A useful analogy here is that a non-deterministic finite state machine can be modelled by a deterministic one — in the deterministic machine, a state represents a set of states of the non-deterministic machine. In the same way, in our partial-order setting, a state represents all the states above it in the partial order. By embedding a flat,[1] non-deterministic model in a lattice, the model becomes deterministic. The next state function $\mathbf{Y}$ can be thought of as a representation of the next state relation

  $$\{(s,t) \in \mathcal{S} \times \mathcal{S} : \mathbf{Y}(s) \sqsubseteq t\}.$$

  Therefore, although technically we deal with a deterministic system, the deterministic system models non-deterministic behaviour.

Example

For synchronous circuit models, the most important way in which non-determinism is used is to model input non-determinism, that is the non-deterministic behaviour of inputs of a circuit. One way of modelling the fact that inputs of the circuit are controlled by the environment, not by

[1] A set without any structure.

the circuit, is to have a non-deterministic next state relation. For example, consider the simple inverter circuit of Figure 3.1.

If the model structure uses a flat state space, the state space and next state relation shown in Figure 3.2 are likely candidates for the model structure. For the next state relation, for each row there is a transition from the state in the first column to each state in the second column.

$$\{(\mathsf{L},\mathsf{L}),(\mathsf{L},\mathsf{H}),(\mathsf{H},\mathsf{L}),(\mathsf{H},\mathsf{H})\}$$

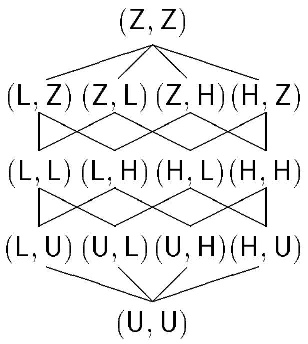
(a) State space

From	To
(L,L)	$(\mathsf{L},\mathsf{H}),(\mathsf{H},\mathsf{H})$
(L,H)	$(\mathsf{L},\mathsf{H}),(\mathsf{H},\mathsf{H})$
(H,L)	$(\mathsf{L},\mathsf{L}),(\mathsf{H},\mathsf{L})$
(H,H)	$(\mathsf{L},\mathsf{L}),(\mathsf{H},\mathsf{L})$

(b) Next state relation

Figure 3.2: Inverter Model Structure — Flat State Space

If a partial order state space is used, one way of constructing the model structure is shown in Figure 3.3. Figure 3.3(a) shows the state space, and Figure 3.3(b) gives the next state function. A c entry in the table means that this row holds for all $c \in \mathcal{C}$.



(a) State space

From	To
(Z,c)	(U,Z)
(L,c)	(U,H)
(H,c)	(U,L)
(U,c)	(U,U)

(b) Next state function

Figure 3.3: Lattice-based Model Structure

Branching time versus linear time

One of the key issues of temporal logics is whether the logic is linear time or branching time. Since the next state function of the model is deterministic, and since in practice all temporal formulas used are finite, the question of whether the logic used is linear or branching time is rather a fine point. Nevertheless, as trajectory evaluation has been described as a linear time approach [26, p. 403], and as non-determinism can be represented by the model structure, the topic should be discussed briefly.

'Logically the difference between a linear and a branching time operator resides with the possibility of path switching ... ' [120]. The model structure proposed here deals with non-determinism by merging paths where necessary. If in the flat model structure there are non-deterministic transitions from state s to states $t_1, \ldots, t_j$, in the lattice model structure there is a state t, such that $t \sqsubseteq t_i$ for $i = 1, \ldots, j$, and a single deterministic transition from s to t. Consider the non-deterministic transition diagram shown in Figure 3.4. The difference between linear time and branching time semantics is nicely illustrated here. Suppose an instantaneous property g is true in states s_1 and s_3 and false in all other states. With a linear time semantics, we can express the property that in all runs of the system, there exists a state from which time all states in the run have the property g. This cannot be expressed in a branching time semantics: for example, in the run $s_0 s_3 s_3 s_3 \ldots$, a branching time semantics always detects the possibility of path-switching and takes into account the potential of a transition from s_3 to s_2.

Using a lattice structure, instead of using the set $\mathcal{S} = \{s_0, \ldots, s_3\}$ as the state space, we use a subset of the power set of $\mathcal{S}$. The state space shown in Figure 3.4 is embedded in the lattice state space shown in Figure 3.5(a) (here, the partial order is shown by dotted lines). The next state relation of Figure 3.4 is replaced with the next state function shown in Figure 3.5(b) (note, only states reachable from s_0 are shown in this transition diagram). Note how the two non-deterministic transitions from s_0 to s_1 and s_3 in Figure 3.4 are merged into one deterministic

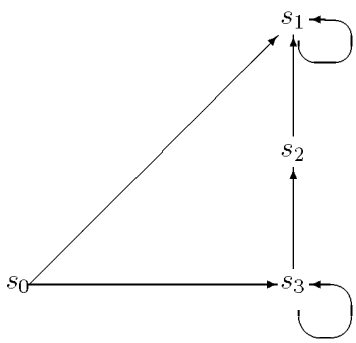
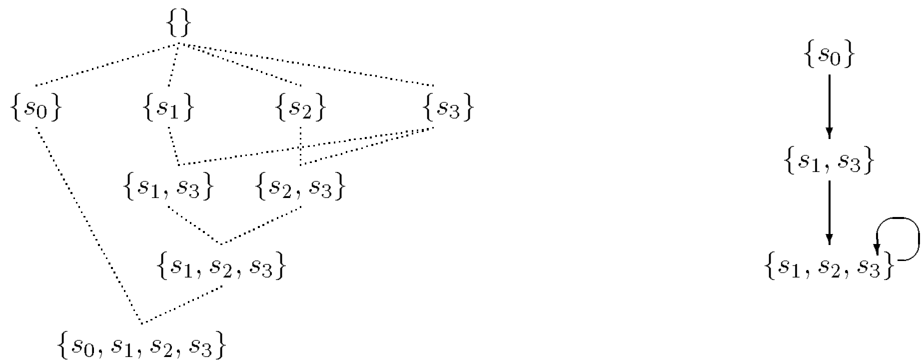


Figure 3.4: Non-deterministic state relation

transition from s_0 to s_4 shown in Figure 3.5(b).



(a) Partial order (b) Transition function

Figure 3.5: Lattice State Space and Transition Function

By using the model structure adopted here, non-deterministic paths that exist in a flat model structure are merged, losing information in the process. It is possible to ask the question whether in all runs of the system property g holds; however, the answer returned will be 'unknown.' So, it would not be accurate to characterise the logic proposed here as either linear time or branching time, since the distinction between the two is blurred. As the expressiveness of the logic and the type of non-determinism used in models is limited compared to many other verification approaches, this question of branching versus linear time semantics is not nearly as important as in other contexts.

In the inverter example above, consider the sequence $\sigma = (\mathsf{L}, \mathsf{H})(\mathsf{U}, \mathsf{H})\ldots$ in the partial order model. This represents both of the sequences

$$(\mathsf{L}, \mathsf{H})(\mathsf{H}, \mathsf{H})\ldots$$
$$(\mathsf{L}, \mathsf{H})(\mathsf{L}, \mathsf{H})\ldots.$$

in the flat model structure. Proving a property of σ will take into account the branching structure at each state in the sequence: but it does so in a trivial way by considering (at the same time) both possible values of the input node of the inverter.

3.2 The Quaternary Logic $\mathcal{Q}$

The four values of $\mathcal{Q}$, the quaternary propositional logic used as the basis of the temporal logic, represent truth, falsity, undefined (or unknown) and overdefined (or inconsistent). Such a logic was proposed by Belnap [10], and has since been elaborated upon and different application areas discussed in a number of other works [59, 125]. This section first gives some mathematical background, based on [58, 113], and then definitions are given and justified.

A *bilattice* is a set together with two partial orders, $\preceq$ and $\leq$, such that the set is a complete lattice with respect to both partial orders. A bilattice is *distributive* if for both partial orders the meet distributes over the join and vice-versa. A bilattice is *interlaced* if the meets and joins of both partial orders are monotonic with respect to the other partial order.

In our application domain, we are interested in the interlaced bilattice

$$\mathcal{Q} = \{\bot, \mathbf{f}, \mathbf{t}, \top\}$$

where the partial orders are shown in Figure 3.6. $\mathbf{f}$ and $\mathbf{t}$ represent the boolean values false and true, $\perp$ represents an unknown value, and $\top$ represents an inconsistent value. $\mathcal{B}$ denotes the set $\{\mathbf{f}, \mathbf{t}\}$ (so $\mathcal{B} \subset \mathcal{Q}$).

The partial order $\preceq$ represents an information ordering (on the truth domain), and the partial order $\leq$ represents a truth ordering. (Note, the ordering $\sqsubseteq$ is used for comparing *states* and the ordering $\preceq$ is used to compare *truth values*). It is very important to emphasise at this point that *different* lattices are used to represent truth information and state information.

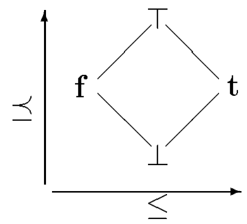


Figure 3.6: The Bilattice $\mathcal{Q}$

Informally, the information ordering indicates how much information the truth value contains: the minimal element $\perp$ contains no truth information; the mutually incommensurable elements $\mathbf{f}$ and $\mathbf{t}$ contain sufficient information to determine truth exactly; and the maximal element $\top$ contains inconsistent truth information. The truth ordering indicates how true a value is. The minimum element in the ordering is $\mathbf{f}$ (without question not true); and the maximum element is $\mathbf{t}$ (without question true). The two elements $\perp$ and $\top$ are intermediate in the ordering — in the first case, the lack of information places it between $\mathbf{f}$ and $\mathbf{t}$, and in the second case, inconsistent information does.

Formally, the partial orders $\leq$ and $\preceq$ are relations on $\mathcal{Q}$ (i.e., subsets of $\mathcal{Q} \times \mathcal{Q}$). It is useful to consider the relations as mappings from pairs of elements to a truth domain (if two elements are ordered by the relation we get a true value, if not a false value). Informally, therefore, we can consider the partial orders as mappings from $\mathcal{Q} \times \mathcal{Q}$ to $\mathcal{B}$.

For representing and operating on $\mathcal{Q}$ as a set of truth values, there are natural definitions for negation, conjunction and disjunction, namely the weak negation operation of the bilattice and the meet and join of the $\mathcal{Q}$ with respect to the truth ordering [58].

These definitions are shown in Table 3.1, and have the following pleasant properties, which makes it suitable for model-checking partially-ordered state spaces.

- The definitions are consistent with the definitions of conjunction, disjunction and negation on boolean values.
- These operations have their natural distributive laws, and also obey De Morgan's laws (so, the definition of disjunction was redundant).
- Efficiency of implementation. The quaternary logic is represented by a dual-rail encoding, i.e. a value in $\mathcal{Q}$ is represented by a pair of boolean values, where:
  - $\perp = (F, F)$,
  - $\mathbf{f} = (F, T)$,
  - $\mathbf{t} = (T, F)$,
  - $\top = (T, T)$.

  If a is represented by the pair (a_1, a_2) and b by the pair (b_1, b_2) then $a \wedge b$ is represented by the pair $(a_1 \wedge b_1, a_2 \vee b_2)$, $a \vee b$ by the pair $(a_1 \vee b_1, a_2 \wedge b_2)$ and $\neg a = (a_2, a_1)$. These operations on $\mathcal{Q}$ can be implemented as one or two boolean operations.

$\wedge$	$\perp$	$\mathbf{f}$	$\mathbf{t}$	$\top$
$\perp$	$\perp$	$\mathbf{f}$	$\perp$	$\mathbf{f}$
$\mathbf{f}$	$\mathbf{f}$	$\mathbf{f}$	$\mathbf{f}$	$\mathbf{f}$
$\mathbf{t}$	$\perp$	$\mathbf{f}$	$\mathbf{t}$	$\top$
$\top$	$\mathbf{f}$	$\mathbf{f}$	$\top$	$\top$

$\vee$	$\perp$	$\mathbf{f}$	$\mathbf{t}$	$\top$
$\perp$	$\perp$	$\perp$	$\mathbf{t}$	$\mathbf{t}$
$\mathbf{f}$	$\perp$	$\mathbf{f}$	$\mathbf{t}$	$\top$
$\mathbf{t}$	$\mathbf{t}$	$\mathbf{t}$	$\mathbf{t}$	$\mathbf{t}$
$\top$	$\mathbf{t}$	$\top$	$\mathbf{t}$	$\top$

	$\neg$
$\perp$	$\perp$
$\mathbf{f}$	$\mathbf{t}$
$\mathbf{t}$	$\mathbf{f}$
$\top$	$\top$

Table 3.1: Conjunction, Disjunction and Negation Operators for $\mathcal{Q}$

Implication, $\Rightarrow$, is defined as a derived operator $a \Rightarrow b \;\equiv\; \neg a \vee b$.

There is an intuitive explanation of the dual-rail encoding and the implementation of the operators. If q is encoded by the pair (a, b), a is evidence *for* the truth of q, and b is evidence *against* q. To compute $q_1 \wedge q_2$, we conjunct the evidence for q_1 and q_2 and take the disjunction of the evidence against. The computation of $q_1 \vee q_2$ is symmetric. And if a is the evidence for q and b the evidence against q, then b is the evidence for $\neg q$ and a is the evidence against $\neg q$.

However nice this intuition, the definition of $\mathcal{Q}$ is not without problem. In the context of a temporal logic, it is hard to justify the definition that $\top \wedge \perp = \mathbf{f}$. Similarly, since $\top \vee \perp = \mathbf{t}$, if $\mathbf{t} = q_1 \vee q_2$ it is not necessarily the case that either q_1 or q_2 is $\mathbf{t}$. Nevertheless it is the 'classical' definition, and is convenient because the dual-rail encoding is efficient. Other definitions are possible too (for example, defining the operations so that if $\top$ is an operand, the result of the operation must be $\top$ too) and might simplify some of the proofs in later sections and chapters; the particular definition adopted in this thesis is not fundamental.

The following properties of $\mathcal{Q}$ are used in subsequent proofs. The first lemma is a consequence of the property that negating a value does not increase the information available.

Lemma 3.1.

1. If $q \neq \neg q$, then $q \not\preceq \neg q$.
2. If $q_1 \preceq q_2$, then $\neg q_1 \preceq \neg q_2$.

Proof.

1. If $q \in \{\perp, \top\}$, then $q = \neg q$.

   If $q = \mathbf{t}$, then $\neg q = \mathbf{f}$: $\mathbf{t} \not\preceq \mathbf{f}$. Similarly, $\mathbf{f} \not\preceq \mathbf{t}$.
2. (a) If $q_1 = \perp$, $\neg q_1 = \perp$. $\perp \preceq q$ for all q.

   (b) If $q_1 = \mathbf{t}$, then $\neg q_1 = \mathbf{f}$ and $q_2 \in \{\mathbf{t}, \top\}$.

   Therefore, $\neg q_1 \preceq \neg q_2$

   (c) Similarly, if $q_1 = \mathbf{f}$, $\neg q_1 \preceq \neg q_2$

(d) If $q_1 = \top$, then $q_1 = \neg q_1 = q_2 = \neg q_2$.

Result follows by reflexivity of partial order.

□

The second lemma extracts some trivial properties of $\mathcal{Q}$ from Table 3.1; these are useful when trying to deduce values of sub-formulas from values of formulas.

Lemma 3.2.

1. If $\mathbf{t} \preceq q_1 \vee q_2$, then $\mathbf{t} \preceq q_i$ for at least one of $i = 1, 2$.
2. If $\mathbf{t} = q_1 \wedge q_2$, then $\mathbf{t} = q_i$ for both of $i = 1, 2$.

   If $\mathbf{t} \preceq q_1 \wedge q_2$, then $\mathbf{t} \preceq q_i$ for both of $i = 1, 2$.
3. If $\mathbf{f} \preceq q_1 \wedge q_2$, then $\mathbf{f} \preceq q_i$ for at least one of $i = 1, 2$.
4. If $\mathbf{f} = q_1 \vee q_2$, then $\mathbf{f} = q_i$ for both of $i = 1, 2$.

   If $\mathbf{f} \preceq q_1 \vee q_2$, then $\mathbf{f} \preceq q_i$ for both of $i = 1, 2$.

Proof.

Consider Table 3.1.

1. $q_1 = \mathbf{t}$ or $q_2 = \mathbf{t}$, or $q_1 = \bot$ and $q_2 = \top$, or $q_1 = \top$ and $q_2 = \bot$.
2. Only when $q_1 = q_2 = \mathbf{t}$ is $q_1 \wedge q_2 = \mathbf{t}$.

   Only for the four bottom, right entries of the table is $q_1 \wedge q_2 \preceq \mathbf{t}$.
3. $q_1 = \mathbf{f}$ or $q_2 = \mathbf{f}$; or $q_1 = \bot$ or $q_2 = \top$; or $q_1 = \top$ or $q_2 = \bot$
4. Only when $q_1 = q_2 = \mathbf{f}$ is $q_1 \vee q_2 = \mathbf{f}$.

   Only for the four upper, left entries of the table is $\mathbf{f} \preceq q_1 \vee q_2$.

□

3.3 An Extended Temporal Logic

The propositional logic $\mathcal{Q}$ is used as the base for the temporal logic TL. This section first presents the scalar version of TL, the fragment of TL not containing variables, and then presents the symbolic version of TL, which contains variables.

3.3.1 Scalar Version of TL

Given a model structure $(\langle \mathcal{S}, \sqsubseteq \rangle, \mathcal{R}, \mathbf{Y})$, a $\mathcal{Q}$-predicate over $\mathcal{S}$ is a function mapping from $\mathcal{S}$ to the bilattice $\mathcal{Q}$. A $\mathcal{Q}$-predicate, p is monotonic if $s \sqsubseteq t$ implies that $p(s) \preceq p(t)$ (monotonicity is defined with respect to the information ordering of $\mathcal{Q}$). A $\mathcal{Q}$-predicate is a generalised notion of predicate, and to simplify notation, the term 'predicate' is used in the rest of this discussion.

Example 3.1.

Take, as an example, the state space $\mathcal{S}$ given in Figure 1.1 on page 9. Define $g, h : \mathcal{S} \to \mathcal{Q}$ by:

$$g(s) = \begin{cases} \bot & \text{when } s = s_0 \\ \mathbf{f} & \text{when } s \in \{s_1, s_2, s_4, s_5, s_6\} \\ \mathbf{t} & \text{when } s \in \{s_3, s_7, s_8\} \\ \top & \text{when } s = s_9 \end{cases} \text{ and } h(s) = \begin{cases} \bot & \text{when } s \in \{s_0, s_2, s_6\} \\ \mathbf{f} & \text{when } s \in \{s_1, s_4, s_5\} \\ \mathbf{t} & \text{when } s \in \{s_3, s_7, s_8\} \\ \top & \text{when } s = s_9 \end{cases}$$

Figure 3.7 depicts these definitions graphically. g and h are $\mathcal{Q}$-predicates. The same state space and functions will be used in subsequent examples.

□

Note that in the example, s_3 is the weakest state for which $g(s) = \mathbf{t}$. In a sense, s_3 partially characterises g, and we use this idea as a building block for characterising predicates, motivating the next definition. Given a predicate p, we are interested in the pairs (s_q, q) where s_q is a weakest state for which $p(s) = q$.

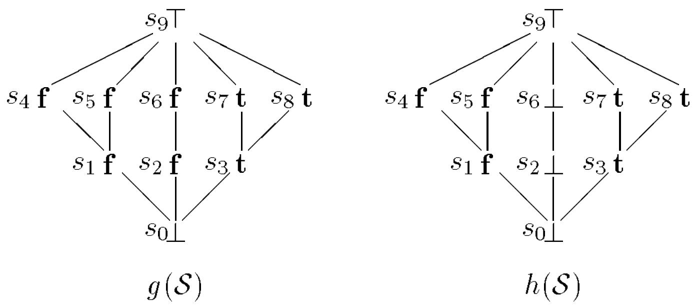


Figure 3.7: Definition of g and h

Definition 3.1.

$(s_q, q) \in \mathcal{S} \times \mathcal{Q}$ is a *defining pair* for a predicate g if $g(s_q) = q$ and $\forall s \in \mathcal{S}, g(s) = q$ implies that $s_q \sqsubseteq s$. $\square$

In Example 3.1 $(s_3, \mathbf{t})$ is a defining pair for g. If $g(s) = \mathbf{t}$ then $s_3 \sqsubseteq s$. However, there is no defining pair $(s_{\mathbf{f}}, \mathbf{f})$ for g since there is no unique weakest element in $\mathcal{S}$ for which g takes on the value $\mathbf{f}$. On the other hand $(s_1, \mathbf{f})$ is a defining pair for h.

Definition 3.2.

If $g\colon \mathcal{S} \to \mathcal{Q}$ then $D(g) = \{(s_q, q) \in \mathcal{S} \times \mathcal{Q} : (s_q, q)$ is a defining pair for $g\}$, is the *defining set* of g. $\square$

Using this definition it is easy to compute the defining sets of the functions g and h that were defined in Example 3.1.

$$D(g) = \{(s_0, \bot), (s_3, \mathbf{t}), (s_9, \top)\}$$

$$D(h) = \{(s_0, \bot), (s_1, \mathbf{f}), (s_3, \mathbf{t}), (s_9, \top)\}$$

If a monotonic predicate has a defining pair for every element in its range, then its defining set uniquely characterises it (see Lemma 3.3 below). Such monotonic predicates are called simple predicates and form the basis of our temporal logic. The following notation is used in the next definition and elsewhere in the thesis: if $g : A \to B$ is a function then $g(A) = \{g(a) : a \in A\}$ is the *range* of g.

Definition 3.3.

A monotone predicate $g\colon \mathcal{S} \to \mathcal{Q}$ is *simple* if $\forall q \in g(\mathcal{S}), \exists (s_q, q) \in D(g)$. □

In Example 3.1, h is simple since every element in the range of h has a defining pair. On the other hand, g is not simple since there is no defining pair $(s_{\mathbf{f}}, \mathbf{f})$. Informally, g is not simple since we cannot use a single element of $\mathcal{S}$ to characterise the values for which $g(s) = \mathbf{f}$.

Definition 3.4.

Some of the important simple predicates are the constant predicates. For each $q \in \mathcal{Q}$, the constant predicate $C_q(s) = q$ has defining set $D(C_q) = \{(\mathsf{X}, q)\}$ and so is simple. □

Note that simple predicates need not be surjective; the only requirement is that if q is in the range of a simple predicate, there is a unique weakest element is $\mathcal{S}$ for which the predicate attains the value q. A trivial result used a number of times here is that the bottom element of $\mathcal{S}$ must be one of the defining values for every predicate: this has the consequence that every element in $\mathcal{S}$ is ordered (by being at least as large as) with respect to one of the defining values of each monotonic predicate.

Theorem 3.3.

If $g, h\colon \mathcal{S} \to \mathcal{Q}$ are simple, then $D(g) = D(h)$ implies that $\forall s \in \mathcal{S}$, $g(s) = h(s)$.

Proof. See Section A.1 □

This result is used later to show the generality of the definitions.

Definition 3.5.

Let G be the set of simple predicates over $\mathcal{S}$. □

We now use G to construct the temporal logic.

Definition 3.6 (The Scalar Extended Logic — TL).

The set of scalar TL formulas is defined by the following abstract syntax

$$\mathrm{TL} ::= \;\; G \;\mid\; \mathrm{TL} \boldsymbol{\wedge} \mathrm{TL} \;\mid\; \boldsymbol{\neg}\mathrm{TL} \;\mid\; \texttt{Next}\,\mathrm{TL} \;\mid\; \mathrm{TL}\,\texttt{Until}\,\mathrm{TL}$$

□

The semantics of a formula is given by the satisfaction relation *Sat* (*Sat* $: \mathcal{S}^\omega \times \mathrm{TL} \rightarrow \mathcal{Q}$). Given a sequence σ and a TL formula g, *Sat* returns the degree to which σ satisfies g.

Suppose g and h are TL formulas. Informally, if g is simple, a sequence satisfies it if g holds of the initial state of the sequence. Conjunction has a natural definition. A sequence satisfies $\boldsymbol{\neg} g$ if it doesn't satisfy g. A sequence satisfies $\texttt{Next}\, g$ if the sequence obtained by removing the first element of the sequence satisfies g. A sequence satisfies $g\,\texttt{Until}\,h$ if there is a k such that the first $k-1$ suffixes of the sequence satisfy g and the k-th suffix satisfies h.[2] Note that in the definitions below, $\boldsymbol{\wedge}$ and $\boldsymbol{\neg}$ (bold face symbols) are operations on TL formulas, whereas $\wedge$ and $\neg$ are operations on $\mathcal{Q}$.

Comment on notation: Sequences are ubiquitous throughout this thesis. There is extensive need to refer to suffixes and individual elements of these sequences. Moreover, individual elements of sequences can be vectors, and on top of this, it is often useful to talk about different sequences. It is plausible to use subscripts to describe all these, but, unfortunately, there is also often a need to refer to these different concepts in close proximity to each other and so there is

[2] In the special case of g and h being simple, this is equivalent to saying that g is true of the first $k-1$ states in the sequence, and h is true of the k-th state.

great opportunity for confusion. To avoid this confusion, a slightly more cumbersome notation is used than might otherwise be desirable. This notation is summarised below.

1. Lower case Greek letters, $\sigma, \tau, \ldots$ are used to refer to sequences.
2. If $\sigma = s_0 s_1 s_2 \ldots$, then σ_i denotes s_i.
3. If $\sigma = s_0 s_1 s_2 \ldots$ is a sequence, $\sigma_{\geq i}$ refers to the sequence $s_i s_{i+1} \ldots$, which is a suffix of σ.
4. Superscripts are used to refer to different sequences, e.g. σ^1, σ^2. Although this conflicts with the usual use of superscript in mathematical text, there is little chance of confusion since 'squaring' states is not defined.
5. If s is a state which is a vector of elements, then $s[k]$ refers to the k-th component of s.

For example, $\sigma^3_{\geq i}$ refers to the suffix of the sequence σ^3 obtained by removing first i elements of σ^3. $(\sigma^3_{\geq i})_0[k] = \sigma^3_i[k]$ is the k-th component of the i-th element in the sequence σ^3.

Definition 3.7 (Semantics of TL**).**

Let $\sigma = s_0 s_1 s_2 \ldots \in \mathcal{S}^\omega$:

1. If $g \in G$ then $\mathit{Sat}(\sigma, g) \;=\; g(s_0)$.
2. $\mathit{Sat}(\sigma, g \wedge h) \;=\; \mathit{Sat}(\sigma, g) \wedge \mathit{Sat}(\sigma, h)$
3. $\mathit{Sat}(\sigma, \neg g) \;=\; \neg \mathit{Sat}(\sigma, g)$
4. $\mathit{Sat}(\sigma, \mathtt{Next}\, g) \;=\; \mathit{Sat}(\sigma_{\geq 1}, g)$
5. $\mathit{Sat}(\sigma, g\, \mathtt{Until}\, h) \;=\; \bigvee_{i=0}^{\infty} \big((\bigwedge_{j=0}^{i-1} \mathit{Sat}(\sigma_{\geq j}, g)) \;\wedge\; \mathit{Sat}(\sigma_{\geq i}, h) \big)$

□

Note that this is the strong version of the until operator: g need never hold, and h must eventually hold. The until operator is defined as an infinite disjunction of conjunctions. That this is well defined comes from $\mathcal{Q}$ being a complete lattice with respect to the truth ordering. Recall that $\wedge$

is defined as the meet of the truth ordering, and $\vee$ is defined as the join. Moreover, in a complete lattice, *all* sets have a meet and join. Therefore each conjunction is well defined, and thus the disjunction of the conjunctions is too. An intuition to support that the definition is well behaved is that the sequence $a_k = \bigvee_{i=0}^{k} ((\bigwedge_{j=0}^{i-1} \mathit{Sat}(\sigma_{\geq j}, g)) \wedge \mathit{Sat}(\sigma_{\geq i}, h))$ is an increasing sequence in $\mathcal{Q}$. As $\mathcal{Q}$ is finite and bounded above, the sequence (a_k) has a limit.

Using these operators we can define other operators as shorthand.

Definition 3.8 (Other operators).

Some that we shall use are:–

- *Disjunction:* $g \vee h = \neg((\neg g) \wedge (\neg h))$.
- *Implication:* $g \Rightarrow h = (\neg g) \vee h$.
- *Sometime:* $\texttt{Exists}\, g = \texttt{t}\, \texttt{Until}\, g$. (Some suffix of the sequence satisfies g.)
- *Always:* $\texttt{Global}\, g = \neg(\texttt{Exists}\, \neg g)$. (No suffix of the sequence does not satisfy g, hence all must satisfy g).
- *Weak until:* $g\, \texttt{UntilW}\, h = (g\, \texttt{Until}\, h) \vee (\texttt{Global}\, g)$. (This doesn't demand that h ever be satisfied.)

Using the operators defined above, other operators can be defined, including bounded versions of `Global`, `Exists`, `UntilW` and `Until` and a periodic operator `Periodic` that can be used to test the state of the system periodically. Other operators — for example, periodic versions of the until operators etc. — are possible too. Two very useful derived operators are the generalised version of `Next` and the bounded always operator.

- The generalised `Next` operator is defined by:

$$\texttt{Next}^0 g = g$$

$$\texttt{Next}^{k+1} g = \texttt{Next}\,(\texttt{Next}^k g)$$

- The bounded always operator, defined by
$$\texttt{Global}\,[(a_0, b_0), \ldots, (a_n, b_n)]\ g = \bigwedge_{j=0}^{n} \Big(\bigwedge_{k=a_j}^{b_j} \texttt{Next}^k g \Big),$$
asks whether g holds between a_j and b_j for $j = 0, \ldots, n$.

□

If $q = \mathit{Sat}(\sigma, g)$ then we say that σ satisfies g with truth value q, and if $q \preceq \mathit{Sat}(\sigma, g)$, then we say that σ satisfies g with truth value at least q.

One of the key properties of the satisfaction relation is that it is monotonic.

Lemma 3.4.

The satisfaction relation is monotonic: for all $\sigma^1, \sigma^2 \in \mathcal{S}^\omega$, if $q = \mathit{Sat}(\sigma^1, g)$ and $\sigma^1 \sqsubseteq \sigma^2$, then $q \preceq \mathit{Sat}(\sigma^2, g)$

Proof. If g is simple, this follows since g is monotonic. Since the operators of $\mathcal{Q}$ (conjunction, disjunction and negation) are all monotonic with respect to their operands, the monotonicity of TL follows by structural induction. Again, for the until operator this relies on $\mathcal{Q}$ being a complete lattice. □

Although the basis of the logic is G, the set of simple predicates, Theorem 3.5 shows that all monotonic predicates can be expressed in TL. If g is a TL formula not containing any temporal operators, then its semantics with respect to a sequence is determined solely by the value of the first element of the sequence. This implies that we can consider such a g to be a predicate from $\mathcal{S} \to \mathcal{Q}$. Formally, overloading the symbol g, we can define $g : \mathcal{S} \to \mathcal{Q}$ by $g(s) = \mathit{Sat}(s\mathsf{XX}\ldots, g)$

Theorem 3.5.

For all monotonic predicates $p : \mathcal{S} \to \mathcal{Q}$, $\exists p' \in$ TL such that $\forall s \in \mathcal{S}$, $p(s) = p'(s)$.

Proof. See Section A.1. □

Consider the functions defined in Example 3.1, and let

$$h'(s) = \begin{cases} \bot & \text{when } s \in \{s_0, s_1, s_4, s_5\} \\ \mathbf{f} & \text{when } s \in \{s_2, s_6\} \\ \mathbf{t} & \text{when } s \in \{s_3, s_7, s_8\} \\ \top & \text{when } s = s_9 \end{cases}$$

$D(h') = \{(s_0, \bot), (s_2, \mathbf{f}), (s_3, \mathbf{t}), (s_9, \top)\}$ and so h' is simple. Note that $g = h \wedge h'$. So, although g is not simple, it can be expressed as the conjunction of two simple predicates.

The *depth* of a TL formula is a measure of how far in the future it describes behaviour of sequences; it shows how deeply nested next state operators are. Formally, if g is a TL formula, its depth, $d(g)$ is defined by:

$$d(g) = 0 \text{ for } g \in G \qquad d(g_1 \wedge g_2) = \max\{g_1, g_2\} \qquad d(\neg g) = d(g)$$

$$d(\texttt{Next}\, g) = d(g) + 1 \qquad d(g_1\, \texttt{Until}\, g_2) = \infty$$

3.3.2 Some Laws of TL

This section presents some of the algebraic laws of TL. These are used extensively in proofs and are often used in practical situations. First, the equivalence of two TL formulas is defined.

Definition 3.9.

If $g, h \in$ TL, then $g \equiv h$ if $\forall \sigma \in \mathcal{S}^\omega$, *Sat*$(\sigma, g_1) =$ *Sat*(σ, g_2). □

TL obeys most of the laws of a boolean algebra ($C_\mathtt{f}$ and $C_\mathtt{t}$, two of the constant simple predicates, are identities under disjunction and conjunction respectively). However, the inverse or complementary laws do not hold (since the law of the excluded middle does not hold). Moreover, if we do consider TL as an algebra, it has a more complex structure than a boolean algebra.

Lemma 3.6 (Some algebraic laws of TL**).**

1. *Commutativity:*

$g_1 \wedge g_2 \ \equiv\ g_2 \wedge g_1,\ g_1 \vee g_2 \ \equiv\ g_2 \vee g_1.$

2. *Associativity:*

$(g_1 \vee g_2) \vee g_3 \ \equiv\ g_1 \vee (g_2 \vee g_3),\ (g_1 \wedge g_2) \wedge g_3 \ \equiv\ g_1 \wedge (g_2 \wedge g_3)$

3. *De Morgan's Law:*

$g_1 \wedge g_2 \ \equiv\ \neg(\neg g_1 \vee \neg g_2),\ g_1 \vee g_2 \ \equiv\ \neg(\neg g_1 \wedge \neg g_2).$

4. *Distributivity of* $\wedge$ *and* $\vee$:

$h \wedge (g_1 \vee g_2) \ \equiv\ (h \wedge g_1) \vee (h \wedge g_2),\ h \vee (g_1 \wedge g_2) \ \equiv\ (h \vee g_1) \wedge (h \vee g_2)$

5. *Distributivity of* `Next` *:*

$\texttt{Next}\,(g_1 \wedge g_2) \ \equiv\ (\texttt{Next}\,g_1) \wedge (\texttt{Next}\,g_2),\ \texttt{Next}\,(g_1 \vee g_2) \ \equiv\ (\texttt{Next}\,g_1) \vee (\texttt{Next}\,g_2).$

6. *Identity:*

$g \vee C_{\mathtt{f}} \ \equiv\ g,\ g \wedge C_{\mathtt{t}} \ \equiv\ g.$

7. *Double negation:*

$\neg\neg g \ \equiv\ g$

Proof. See Section A.1.3. □

3.3.3 Symbolic Version

Describing the properties of a system explicitly by a set of scalar formulas of TL would be far too tedious. Symbolic formulas allow a concise representation of a large set of scalar formulas. A symbolic formula represents the set of all possible instantiations of that symbolic formula.

TL is extended to symbolic domains by allowing boolean variables to appear in the formulas. Let $\mathcal{V}$ be a set of variable names $\{v_1, \ldots, v_n\}$. It would be possible to define the symbolic version of the logic by introducing quaternary variables. However, in practice, it is boolean variables which are needed, and introducing only boolean variables means that simpler and more efficient implementations of the logic can be accomplished. Furthermore, the effect of a quaternary variable can be created by introducing a pair of boolean variables.

Definition 3.10 (The Extended Logic — TL).

The syntax of the set of symbolic TL formulas, $\dot{\text{TL}}$, is defined by:–

$$\dot{TL} ::= \; G \mid \mathcal{V} \mid \dot{TL} \wedge \dot{TL} \mid \dot{\neg}\dot{TL} \mid \dot{\mathtt{Next}}\,\dot{TL} \mid \dot{TL}\,\dot{\mathtt{Until}}\,\dot{TL}$$

□

The derived operators are defined in a similar way to Definition 3.8. For convenience, where there is little chance of confusion, the dots on $\dot{\text{TL}}$ formulas are omitted.

The satisfaction relation is now determined by a sequence, a formula, *and* an *interpretation* of the variables. An interpretation, ϕ, is a mapping from variables to the set of constant predicates $\{\mathbf{f}, \mathbf{t}\}$, Let $\Phi = \{\phi : \;\; \phi\colon \mathcal{V} \to \{\mathbf{f}, \mathbf{t}\}\}$ be the set of all interpretations. Given an interpretation ϕ of the variables, there is a natural, inductively defined interpretation of TL formulas. For a given $\phi \in \Phi$, we extend the definition from $\mathcal{V}$ to all of TL by defining:

$$\phi(g) = g \text{ if } g \in G$$

$$\phi(\neg g) = \neg\phi(g)$$

$$\phi(g_1 \wedge g_2) = \phi(g_1) \wedge \phi(g_2)$$

$$\phi(\mathtt{Next}\, g) = \mathtt{Next}\, \phi(g)$$

$$\phi(g_1 \,\mathtt{Until}\, g_2) = \phi(g_1) \,\mathtt{Until}\, \phi(g_2)$$

This can be expressed syntactically: if $\phi(v_i) = b_i$, replace each occurrence of v_i with b_i, written as $\phi(g) = g[b_1/v_1, \ldots, b_n/v_n]$.

Given a sequence and a symbolic formula, the symbolic satisfaction relations, SAT_q, determine for which interpretations of variables the sequence satisfies the formula with which degree of truth. For example, we may be interested in the interpretations of variables for which a sequence satisfies a formula with truth value $\mathtt{t}$, or the interpretations for which a sequence satisfies a formula with truth value at least $\mathtt{t}$. By being able to determine for which interpretations a property holds with a given degree of truth, we are able to construct appropriate verification conditions. The scalar satisfaction relation, *Sat*, is used in the definition of the symbolic relations.

Definition 3.11 (Satisfaction relations for $\dot{\text{T}}$L).

A number of satisfaction relations are defined.

- For $q = \mathbf{f}, \mathbf{t}, \top$,

  $SAT_q(\sigma, g) \ = \ \{\phi \in \Phi : q = Sat(\sigma, \phi(g))\}$.

- For $q = \mathbf{f}, \mathbf{t}, \top$,

  $SAT_{q\uparrow}(\sigma, g) \ = \ \{\phi \in \Phi : q \preceq Sat(\sigma, \phi(g))\}$. $\square$

Note that if g is a (symbolic) formula and ϕ an interpretation, then $SAT_q(\sigma, g) \subseteq \Phi$, while $Sat(\sigma, \phi(g)) \in \mathcal{Q}$. Informally,

- $SAT_{\top\uparrow}(\sigma, g)$ is the set of interpretations for which g and $\neg g$ hold. Such results are undesirable and verification algorithms should detect and flag them.

  $SAT_{\top\uparrow}(\sigma, g) = SAT_{\top}(\sigma, g)$.

- $SAT_{\mathbf{t}}(\sigma, g)$ is the set of interpretations for which g is (sensibly) true.

  $SAT_{\mathbf{t}\uparrow}(\sigma, g) = SAT_{\top}(\sigma, g) \cup SAT_{\mathbf{t}}(\sigma, g)$.

- $SAT_{\mathbf{f}}(\sigma, g)$ is the set of mappings for which g is (sensibly) false.

  $SAT_{\mathbf{f}\uparrow}(\sigma, g) = SAT_{\top}(\sigma, g) \cup SAT_{\mathbf{f}}(\sigma, g)$.

Thus each satisfaction relation defines a set of interpretations for which a desired relationship holds. Sets of interpretations can be represented efficiently using BDDs, as is discussed in Chapter 6.

3.4 Circuit Models as State Spaces

In practice, the model-checking algorithms described in this thesis are applied to circuit models. The state space for such a model represents the values which the nodes in the circuit take on, and the next state function can be represented implicitly by symbolic simulation of the circuit. The nodes in a circuit take on high (H) and low (L) voltage values. It is useful, both computationally and mathematically, to allow nodes to take on unknown (U) and inconsistent or over-defined (Z) values. The set $\mathcal{C} = \{\mathsf{U}, \mathsf{L}, \mathsf{H}, \mathsf{Z}\}$ forms the lattice defined in Figure 1.2 on page 10.

The special case of the state space being a cross-product of quaternary sets need be treated no differently than the general case (when the state space is an arbitrary lattice) as all the above definitions apply. However, it is convenient to establish some additional notation. Let $\mathcal{S} = \mathcal{C}^n$ for some n. Typically in this case $\mathcal{R} = \{\mathsf{U}, \mathsf{L}, \mathsf{H}\}^n$ (node values can be unknown or have well-defined values, but cannot be in an inconsistent state).

Let G_n be the smallest set with the following predicates:–

- The constant predicates: $\mathbf{f}, \mathbf{t}, \bot, \top \in G_n$;
- $\forall i \in \{1, \ldots, n\}, [i] \in G$.

Here $[i]$ refers to the i-th component of the state space. A formula g is evaluated with respect to a state by substituting for each $[i]$ which appears in the formula the value of the i-th component of the state. Formally,

- $[i](s) = \begin{cases} \bot & \text{when } s[i] \equiv \mathsf{U} \\ \mathbf{f} & \text{when } s[i] \equiv \mathsf{L} \\ \mathbf{t} & \text{when } s[i] \equiv \mathsf{H} \\ \top & \text{when } s[i] \equiv \mathsf{Z} \end{cases}$

- $\mathbf{f}(s) = \mathbf{f}$;

- $\mathbf{t}(s) = \mathbf{t}$;

- $\bot\,(s) = \bot$;

- $\top(s) = \top$;

Note that all members of G_n are simple and hence monotonic. The definition below of the TL_n is based on that of TL, replacing G with G_n. The set of scalar TL_n formulas is defined by the following abstract syntax:

$$\mathrm{TL}_n ::= \; G_n \mid \mathrm{TL}_n \wedge \mathrm{TL}_n \mid \neg\mathrm{TL}_n \mid \texttt{Next}\,\mathrm{TL}_n \mid \mathrm{TL}_n\,\texttt{Until}\,\mathrm{TL}_n$$

The semantics of TL_n is patterned on Definition 3.7, replacing G with G_n; this is reproduced below for completeness.

Definition 3.12 (Semantics of TL_n).

The semantics of TL_n formulas is defined by the following:

1. If $g \in G_n$ then $\mathit{Sat}(\sigma, g) = g(s_0)$;
2. $\mathit{Sat}(\sigma, g \wedge h) = \; \mathit{Sat}(\sigma, g) \wedge \mathit{Sat}(\sigma, h)$;
3. $\mathit{Sat}(\sigma, \neg g) = \; \neg\mathit{Sat}(\sigma, g)$;
4. $\mathit{Sat}(\sigma, \texttt{Next}\, g) = \; \mathit{Sat}(\sigma_{\geq 1}, g)$;
5. $\mathit{Sat}(\sigma, g\,\texttt{Until}\,h) = \; \bigvee_{i=0}^{\infty} ((\bigwedge_{j=0}^{i-1} \mathit{Sat}(\sigma_{\geq j}, g)) \wedge \mathit{Sat}(\sigma_{\geq i}, h)))$.

□

These definitions are useful because in practice properties of interest are built up from the set of predicates that say things about individual state components. Lemma 3.7 shows that restricting the basis of TL_n to G_n is not a real restriction as any simple predicate can be constructed using the operators such as conjunction.

Lemma 3.7 (Power of G).
If p is a simple predicate over $\mathcal{C}^n$, then there is a predicate $g_p \in \mathrm{TL}_n$ such that $p \equiv g_p$.

Proof. See Section A.1. □

The combined impact of Theorem 3.5 and Lemma 3.7 is that the logic TL_n is powerful enough to describe all interesting (monotonic) state predicates over $\mathcal{Q}$.

The definition of the symbolic version of TL_n is exactly the same as the general definitions (Definitions 3.10 and 3.11), substituting G_n for G.

The set of TL_n formulas in which $\top$ does not syntactically appear is known as the *realisable fragment* of TL_n. If g is a formula in the realisable fragment of TL_n, then *Sat*$(\sigma, g) = \top$ only if there exists i, j such that $\sigma_i[j] = \mathsf{Z}$. Thus, if g is a formula with this restriction, and Z does not appear in σ then $\mathit{SAT}_{\mathsf{t}\dagger}(\sigma, g) = \mathit{SAT}_{\mathsf{t}}(\sigma, g)$. This result is important since we are most interested in the $\mathit{SAT}_{\mathsf{t}}$ relation. As shown in the next chapter, there is a good decision procedure for the relation $\mathit{SAT}_{\mathsf{t}\dagger}$: we check whether $\mathit{SAT}_{\mathsf{t}\dagger}(\sigma, g) = \mathit{SAT}_{\mathsf{t}}(\sigma, g)$, and thereby extend the decision procedure to formulas in the realisable fragment of TL_n to determine the $\mathit{SAT}_{\mathsf{t}}$ relation too.

Other Application Areas

Although $\mathcal{Q}$ is proposed here as the basis of a temporal logic, it may have other applications in computer science. In a widely quoted and influential logic text, the White Knight says (the quote is taken from an extract dealing with names and reference):

> 'It's long,' said the Knight, 'but it's very, *very* beautiful. Everybody that hears me sing it — either it brings *tears* into their eyes, or else —'
>
> ...
>
> 'Or else it doesn't you know ... ' — [31]

In his commentary on this, Heath says [79]: 'An essentially vacuous claim, since it merely sets forth the logical truism, p or not-p, embodied in the "law of the excluded middle."' In the light of the preceding discussion, the White Knight's claim, and particularly Heath's critique can be seen to be problematic. While it will be the case that hearing the White Knight sing the song makes everyone cry or not cry, as computer scientists, we are interested in making predictions about the behaviour of a system under study. Thus the analyses of the White Knight and Heath are somewhat simplistic, and do not take into account lack of information or inconsistent information which often occur when reasoning about the world.

A far more serious instance of the same error can be found in [51] where in *The Beryl Coronet*, Sherlock Holmes says: 'It is an old maxim of mine that when you have excluded the impossible, whatever remains, however improbable, must be the truth.' In this context, Holmes is using logic to reason about a system that inherently has partial and inconsistent information. Our knowledge about such a system must reflect this: the characterisation of propositions about the world into 'impossible' and 'truth' is, as argued earlier, an inadequate logical framework for reasoning. Given the influence of this work on an important branch of logic and deduction, it is important to show the limits of a two-valued logic. And, the notion that simple characterisations may not be appropriate was recognised in work contemporaneous with [51], in an approach that is to be preferred: 'Truth is rarely pure, and never simple' [126].

3.5 Alternative Definition of Semantics

Although in this chapter the semantics of TL formulas was given through the definition of the satisfaction relations, there are alternative ways in which the semantics could be given.[3] A method that is useful to consider here because its underlying motivation leads to an effective verification method defines the semantics by giving for each temporal logic formula the set of sequences that satisfy it. For TL the same pattern could be used, adjusting for the fact that TL is quaternary. Definition 3.13 suggests how this could be done for TL (based on [120, p. 523]).

Definition 3.13 (Alternative definition of semantics).

$$\|g\|_{\mathtt{t}} = \{\sigma : \mathtt{t} = g(s_0)\} \text{ if } g \in G.$$

$$\|g_1 \wedge g_2\|_{\mathtt{t}} = \|g_1\|_{\mathtt{t}} \cap \|g_2\|_{\mathtt{t}}.$$

$$\|\neg g\|_{\mathtt{t}} = \|g\|_{\mathtt{f}}.$$

$$\|g_1 \,\mathtt{Until}\, g_2\|_{\mathtt{t}} = \bigcup_{i=0}^{\infty}\{\sigma \in \mathcal{S}^{\mathcal{T}} : \forall j : 0 \leq j < i,\ \sigma_{\geq j} \in \|g_1\|_{\mathtt{t}} \text{ and } \sigma_{\geq i} \in \|g_2\|_{\mathtt{t}}\}.$$

The definition of $\|g\|_q$ for values of $\mathcal{Q}$ other than $\mathtt{t}$ is similar. □

If this definition were used to give the semantics, then to ask whether σ satisfies g with degree q is to ask whether $\sigma \in \|g\|_q$. Similar definitions could be given for satisfaction 'with degree at least q'. Practically speaking, this definition is not useful since these sets are so large that even if only finite subsequences were considered (which is often reasonable to do) the sets would be too large to compute and represent explicitly.

However, the partial order representation of the state space is extremely useful. Take as an example simple predicates. If g is a simple predicate, in general, the set of sequences for which *Sat*$(\sigma, g) = \mathtt{t}$ will be too large to compute. However, we have seen that the defining set of g, $D(g)$, essentially captures this information: if, for example, $(s_{\mathtt{t}}, \mathtt{t})$ and $(s_{\top}, \top)$ are defining pairs, and if σ is an arbitrary sequence, then $\sigma \in \|g\|_{\mathtt{t}}$ if $s_{\mathtt{t}} \preceq \sigma_0 \prec s_{\top}$.

[3] The semantics are the same; it is the way that the semantics is *given* that differs.

In the same way as the defining sets of a simple predicates characterise the simple predicates, there are analogous structures for other TL formulas that characterise them. And in the same way the defining sets can be considered to give semantics to simple predicates when viewed as TL formulas, these analogous structures give semantics to more complicated TL formulas, and can therefore be used to test satisfaction of sequences. This is the subject of the next chapter.

Chapter 4

Symbolic Trajectory Evaluation

This chapter develops a model checking algorithm for TL. It is based on the idea raised in the last part of Chapter 3 that formulas of TL can be characterised by the set of sequences or trajectories which satisfy them.

Initially, only the scalar version of TL is examined. Extension to the symbolic case is straightforward; however, there is enough extra notation and detail to make an exposition of the scalar case clearer, which overcomes the disadvantage of a little repetition to present the symbolic case.

Let the model structure of the system be $\mathcal{M} = (\langle \mathcal{S}, \sqsubseteq \rangle, \mathcal{R}, \mathbf{Y})$. $\mathcal{S}^\omega$ is the set of sequences of the state space. The partial order on $\mathcal{S}$ is extended point-wise to sequences. Informally, the *trajectories* are all the possible runs of the system; formally, a *trajectory*, σ, is a sequence compatible with the next state function:

$$\forall i \geq 0, \mathbf{Y}(\sigma_i) \sqsubseteq \sigma_{i+1}.$$

Let $\mathcal{S}_\mathcal{T}$ be the set of trajectories and, $\mathcal{R}_\mathcal{T} = \mathcal{R}^\omega \cap \mathcal{S}_\mathcal{T}$ is the set of *realisable trajectories*. $\mathcal{R}_\mathcal{T}$ represents those trajectories corresponding to real behaviours of a system. $\mathcal{R}_\mathcal{T}(m) = \{\sigma_0\sigma_1 \ldots \sigma_{m-1} : \sigma \in \mathcal{R}_\mathcal{T}\}$ is the set of prefixes of $\mathcal{R}_\mathcal{T}$ of length m.

Section 4.1 explores the style of verification adopted; this introduces some useful notation and definitions and guides the rest of this discussion. Section 4.2 shows that a formula of TL can be characterised by the sets of minimal trajectories that satisfy it, and furthermore shows that these sets can be used to accomplish verification. The computation of such sets is not directly possible, but Section 4.3 shows that computing approximations of the sets *is* feasible (and as later experimental evidence will show, forms a good basis for practical verification). Finally, Section 4.4 generalises the work to the full symbolic logic.

4.1 Verification with Symbolic Trajectory Evaluation

The style of verification used in symbolic trajectory evaluation (STE) is to ask questions of the form:

> Do all trajectories that satisfy g also satisfy h?

The formula g is known as the *antecedent*, and the formula h is known as the *consequent*. ‘Satisfy’ is a broad term — there are a number of satisfaction relations that can be used. Which one matches our notion of correctness? There are a number of possible ways of modelling correctness, and the key issue is how to deal with inconsistent information. How correctness is modelled depends on choices the verifier makes — although guided by technical considerations, the verifier has considerable flexibility. There are two obvious ways to formalise the notion of ‘trajectory σ (successfully) satisfies g’.

$$\mathbf{t} = \mathit{Sat}(\sigma, g) \tag{4.1}$$

$$\mathbf{t} \preceq \mathit{Sat}(\sigma, g) \tag{4.2}$$

Relation (4.1) captures a more precise notion — successful satisfaction describes a situation where inconsistency has not caused a predicate to be true of a trajectory. Intuitively, it is a better model of satisfaction than Relation (4.2). However, the latter definition has some advantages: it does capture some useful information; most importantly, as shown later, there is an efficient model checking algorithm using Relation (4.2); and it is often practical to infer the former relation from the latter one. For this reason, we concentrate, for the moment, on the second choice.

Corresponding to these two definitions, there are two ways of asserting correctness with respect to a formula.

Definition 4.1.
$g \Longrightarrow\!\!\triangleright\, h$ if and only if $\forall \sigma \in \mathcal{R}_{\mathcal{T}}, \mathbf{t} = \mathit{Sat}(\sigma, g)$ implies that $\mathbf{t} = \mathit{Sat}(\sigma, h)$.

and

Definition 4.2.
$g \Longrightarrow\!\!\gg h$ iff $\forall \sigma \in \mathcal{S}_{\mathcal{T}}, \mathbf{t} \preceq \mathit{Sat}(\sigma, \phi(g))$ implies that $\mathbf{t} \preceq \mathit{Sat}(\sigma, \phi(h))$.

The first definition takes a very precise view of realisability. First, we only consider realisable trajectories — if there are unrealisable trajectories with strange behaviour, then these are ignored. Moreover, by this definition a sequence satisfying a formula with degree of truth greater than $\mathbf{t}$ (i.e. with degree $\top$) is undesirable. In practice, the model checking algorithm will check in addition that there are some realisable trajectories which satisfy the antecedent (i.e. that the verification assertion is not satisfied vacuously). I submit that this definition best captures the notion of correctness.

The second definition takes a more relaxed view of inconsistent behaviour. We consider the behaviour of all trajectories, whether realisable or not, and treat the truth values $\mathbf{t}$ and $\top$ as satisfying the notion of correctness. Although, this definition is not as good a model of correctness as the first, it has the advantage that there is an efficient verification method for it.

Therefore, for pragmatic reasons, we concentrate at first on Definition 4.2, which will be central in the development of the next three sections. These sections show how an efficient verification methodology for correctness assertions based on this definition can be developed. The last part of Section 4.4 shows that for circuit models, this methodology can be used to infer correctness assertions based on Definition 4.1.

4.2 Minimal Sequences and Verification

This section first formalises the notion of the sets of minimal trajectories satisfying formulas, and then shows how these sets can be used in verification.

The first definition is an auxiliary one: given a subset of a partially-ordered set, it is useful to be able to determine the minimal elements of the set. If B is a subset of A, then $b \in B$ is a minimal element of B if no other element in B is smaller than b (i.e. all elements of A smaller than b do not lie in B).

Definition 4.3.

If A is a set, $B \subset A$, and $\sqsubseteq$ a partial order on A, then

$$\min B = \{b \in B \ : \ \text{if } \exists a \in A \ni a \sqsubseteq b, \ \text{either } a = b \text{ or } a \notin B\}.$$

□

Definition 4.4.

If g is a (scalar) TL formula, then $\min g$ is the set of minimal trajectories satisfying g, where $\min g$ is defined by: $\min g = \min\{\sigma \in \mathcal{S}_{\mathcal{T}} : \mathbf{t} \preceq \mathit{Sat}(\sigma, g)\}$ □

Note that if $\min g \subseteq \min h$, then every trajectory that satisfies g also satisfies h. For suppose σ satisfies g: then there must exist $\sigma' \in \min g$ such that $\mathbf{t} \preceq \mathit{Sat}(\sigma', g)$ and $\sigma' \sqsubseteq \sigma$; but since $\min g \subseteq \min h$, $\sigma' \in \min h$ and hence $\mathbf{t} \preceq \mathit{Sat}(\sigma', h)$; hence by monotonicity $\mathbf{t} \preceq \mathit{Sat}(\sigma, h)$. This gives some indication that manipulating and comparing the sets of minimal trajectories that satisfy formulas can be useful in verification.

Although we will be comparing sets of sequences, containment is too restrictive, motivating a more general method of set comparison. The statement 'every trajectory that satisfies g also satisfies h' implies that the requirements for g to hold are stricter than the requirements for h to hold. Thus, if σ is a minimal trajectory satisfying g, σ must satisfy h. Since the requirements for g are stricter than the requirements on h, σ need not be a *minimal* trajectory satisfying h, but there must be a minimal trajectory, σ', satisfying h where $\sigma' \sqsubseteq \sigma$. This is the intuition behind the following definition, which defines a relation over $\mathcal{P}(\mathcal{S})$, the power set of $\mathcal{S}$.

Definition 4.5.

If $\mathcal{S}$ is a lattice with partial order $\sqsubseteq$ and $A, B \subseteq \mathcal{S}$, then

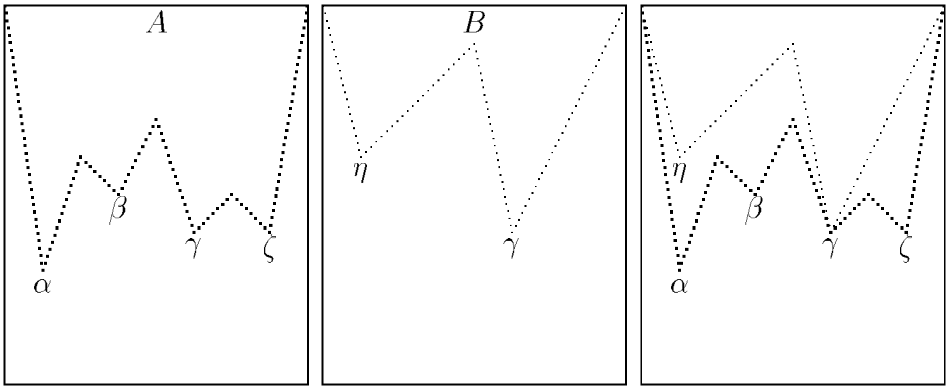
$A \sqsubseteq_{\mathcal{P}} B$ if $\forall b \in B, \exists a \in A$ such that $a \sqsubseteq b$. □

To illustrate this definition, consider the example of Figure 4.1. Assume A and B are subsets of some partially ordered set, $\mathcal{S}$. Note that in this example that both A and B are upward closed. Although the definitions given here do not require this, we will be dealing with upward closed sets.[1] Figure 4.1(a) depicts A. Let $A_m = \min A = \{\alpha, \beta, \gamma, \zeta\}$ be the set of minimal elements of A. Then A consists of all the elements above the dotted line. Similarly, Figure 4.1(b), depicts B. Let $B_m = \min B = \{\eta, \gamma\}$ be the set of minimal elements of B. Figure 4.1(c) is the superposition of Figures 4.1(a) and (b).

Note that $A_m \sqsubseteq_{\mathcal{P}} B_m$. For each element of B_m there is an element of A_m less than or equal to it: $\alpha \sqsubseteq \eta$ and $\gamma \sqsubseteq \gamma$.

Suppose A is the set of elements with property g, and that B is the set of elements with property h. Then $\min g = A_m$ and $\min h = B_m$. By examining the figure it is easy to see that all elements of $\mathcal{S}$ that have property h also have property g (h implies g). But, note that $\min h \not\subseteq \min g$. However, it is the case that $\min g \sqsubseteq_{\mathcal{P}} \min h$, which motivates exploring the $\sqsubseteq_{\mathcal{P}}$ relation further.

[1] We will be manipulating sets of trajectories and sequences that satisfy formulas; that these are upward closed follows from the monotonicity of the satisfaction relation (Lemma 3.4).



(a) A (b) B (c) A and B

Figure 4.1: The Preorder $\sqsubseteq_{\mathcal{P}}$

Lemma 4.1.

If $\mathcal{S}$ is a lattice with partial order $\sqsubseteq$, then $\sqsubseteq_{\mathcal{P}}$ is a preorder (i.e., it is reflexive and transitive).

Proof.

Reflexivity follows directly from the reflexivity of $\sqsubseteq$.

Suppose that $A \sqsubseteq_{\mathcal{P}} B$ and $B \sqsubseteq_{\mathcal{P}} C$, and let $c \in C$.

(1)	$\exists b \in B \ni b \sqsubseteq c$	$B \sqsubseteq_{\mathcal{P}} C$: $\forall c \in C, \exists b \in B \ni b \sqsubseteq c$
(2)	$\exists a \in A \ni a \sqsubseteq b$	$A \sqsubseteq_{\mathcal{P}} B$: $\forall b \in B, \exists a \in A \ni a \sqsubseteq b$
(3)	$a \sqsubseteq c$	$\sqsubseteq$ is transitive
(4)	$A \sqsubseteq_{\mathcal{P}} C$	Since c was arbitrary.

□

Note that if $B \subseteq A$, then $A \sqsubseteq_{\mathcal{P}} B$. The following theorem shows the importance of the definition of $\sqsubseteq_{\mathcal{P}}$.

Theorem 4.2.

If g and h are TL formulas, then $g \Longrightarrow\!\!\!> h$ if and only if $\min h \sqsubseteq_{\mathcal{P}} \min g$.

Proof.

By the definition of minimal sets, if $\mathbf{t} \preceq Sat(\sigma, g)$, there exists $\sigma' \in \min g$ with $\sigma' \sqsubseteq \sigma$.

$$\begin{aligned} g \Longrightarrow\!\!\!> h \quad & \Longleftrightarrow \forall \sigma \in \mathcal{S}_{\mathcal{T}}, \mathbf{t} \preceq Sat(\sigma, g) \text{ implies that } \mathbf{t} \preceq Sat(\sigma, h) \\ & \Longleftrightarrow \forall \sigma' \in \min g \text{ implies that } \mathbf{t} \preceq Sat(\sigma', h) \\ & \Longleftrightarrow \forall \sigma' \in \min g \text{ implies } \exists \sigma'' \in \min h, \text{ with } \sigma'' \sqsubseteq \sigma \\ & \Longleftrightarrow \min h \sqsubseteq_{\mathcal{P}} \min g \end{aligned}$$

$\square$

Although computing the minimal sets directly is often not practical, it is possible to find approximations of the minimal sets (they are approximations because they may contain some redundant sequences). The next section shows how to construct two types of approximations to the minimal sets. $\Delta^{\mathbf{t}}(h)$ is an approximation of the set of minimal *sequences* that satisfy h, and $T^{\mathbf{t}}(g)$ is an approximation of $\min g$. The importance of these approximations are that (i) $\Delta^{\mathbf{t}}(h) \sqsubseteq_{\mathcal{P}} T^{\mathbf{t}}(g)$ exactly when $g \Longrightarrow\!\!\!> h$ (an analogue of Theorem 4.2), and (ii) there is an efficient method for computing these approximations, which we now turn to.

4.3 Scalar Trajectory Evaluation

The method of computing the approximations to the minimal sets of formulas is based on symbolic trajectory evaluation (STE), a model checking algorithm for checking partially-ordered state spaces. The original version of STE was first presented in [25] and a full description of STE can be found in [116]. In these presentations, the algorithm is applied only to trajectory formulas, a restricted, two-valued temporal logic. This chapter generalises earlier work in two important respects.

1. It presents the theory for applying STE to the quaternary logic.
2. It presents the theory for the full class of TL. In particular it deals with disjunction and negation.

This section examines the scalar version of TL and shows how given a TL formula, a set of sequences characterising the formula can be constructed. Recall the definition of defining pair and defining set from Section 3.3.1. The defining set of a simple predicate characterises that predicate; this can be used as a building block to find a characterisation of all temporal predicates. By using the partial order representation, an approximation of the minimal sequences that satisfy a formula can be used to characterise a formula. These sets are called defining sequence sets. Practical experience with verification using STE has shown that there are many formulas that have small defining sequence sets.

This section shows how to construct defining sequence sets using the defining pairs of simple predicates as the starting point. The defining sequence sets of a formula are a pair of sets where the first set of the pair contains those sequences, σ, for which $\mathrm{t} \preceq \mathit{Sat}(\sigma, g)$, and the second set contains those sequences for which $\mathrm{f} \preceq \mathit{Sat}(\sigma, g)$. These sets are constructed using the syntactic structure of TL formulas. If a formula is simple its defining sequence sets are constructed directly from the defining set of the formula. For compound formulas, these sets are constructed by performing set manipulation described below.

As manipulating sets of sequences is very important, first we build up some notation for manipulating and referring to such sets.

Definition 4.6 (Notation).

If A and B are subsets of a lattice $\mathcal{L}$ on which a partial order $\sqsubseteq$ is defined, then $A \amalg B = \{a \sqcup b \ : \ a \in A, b \in B\}$. If $g\colon \mathcal{L} \to \mathcal{L}$, $g(A)$ continues to represent the range of g, and similarly, $g(\langle A, B\rangle) = \langle g(A), g(B)\rangle$. $\square$

Note that we write $A \amalg B$ rather than $A \sqcup B$ since although $A \amalg B$ is a least upper bound (with respect to $\sqsubseteq_{\mathcal{P}}$) of A and B it is not *the* least upper-bound (this reflects the fact that $\sqsubseteq_{\mathcal{P}}$ is a preorder not a partial order).

The two fundamental operations used are join and union, and it is worth discussing how they are used. First, if we know how to characterise sequences that satisfy g_1 and those that satisfy g_2, how do we characterise sequences that satisfy $g_1 \wedge g_2$? Let $q \in \mathcal{Q}$ and suppose that σ^1 and σ^2 are the weakest sequences such that $q \preceq \mathit{Sat}(\sigma^i, g_i)$. Let $\sigma^J = \sigma^1 \sqcup \sigma^2$. Clearly, $q \preceq \mathit{Sat}(\sigma^J, g_1 \wedge g_2)$. Moreover, suppose $q \preceq \mathit{Sat}(\sigma', g_1 \wedge g_2)$, then it must be that $q \preceq \mathit{Sat}(\sigma', g_1)$ and $q \preceq \mathit{Sat}(\sigma', g_2)$. Thus $\sigma^1 \sqsubseteq \sigma'$ and $\sigma^2 \sqsubseteq \sigma'$ since the σ^i are the weakest sequences such that $q \preceq \mathit{Sat}(\sigma^i, g_i)$. But, since $\sigma^J = \sigma^1 \sqcup \sigma^2$, $\sigma^J \sqsubseteq \sigma'$. Thus σ^J is the weakest sequence satisfying $g_1 \wedge g_2$.

What about characterising sequences that satisfy $g_1 \vee g_2$? At first it may seem that this is analogous, and we should just use meet instead of join. However, this is not symmetric: since we are characterising a predicate by the *weakest* sequences that satisfy it, taking the meets will lose information. While it will be the case that if $q \preceq \mathit{Sat}(\sigma', g_1 \vee g_2)$ then $\sigma^1 \sqcap \sigma^2 \preceq \sigma'$, the converse does not hold in general. This means that to characterise $g_1 \vee g_2$ we need to use both σ^1 and σ^2.

Since the law of the excluded middle does not hold in the quaternary logic, we need to characterise both the sequences that satisfy a predicate with value at least $\mathbf{t}$ and those that satisfy a predicate with value at least $\mathbf{f}$.

Definition 4.7 (Defining sequence set).

Let $g \in$ TL. Define the *defining sequence sets* of g as $\Delta(g) = \langle \Delta^{\mathbf{t}}(g), \Delta^{\mathbf{f}}(g) \rangle$, where the $\Delta^q(g)$ are defined recursively by:

1. If g is simple, $\Delta^q(g) = \{s\mathsf{XX}\ldots : (s, q) \in D_g$, or $(s, \top) \in D_g\}$. This says that provided a sequence has as its first element a value at least as big as s then it will satisfy g with truth value at least q. Note that $\Delta^q(g)$ could be empty.

2. $\Delta(g_1 \vee g_2) = \langle \Delta^{\mathbf{t}}(g_1) \cup \Delta^{\mathbf{t}}(g_2), \Delta^{\mathbf{f}}(g_1) \amalg \Delta^{\mathbf{f}}(g_2) \rangle$

Informally, if a sequence satisfies $g \vee h$ with a truth value at least $\mathtt{t}$ then it must satisfy either g or h with truth value at least $\mathtt{t}$. Similarly if it satisfies $g \vee h$ with a truth value at least $\mathtt{f}$ then it must satisfy both g and h with a truth value at least $\mathtt{f}$.

3. $\Delta(g_1 \wedge g_2) = \langle \Delta^{\mathbf{t}}(g_1) \amalg \Delta^{\mathbf{t}}(g_2), \Delta^{\mathbf{f}}(g_1) \cup \Delta^{\mathbf{f}}(g_2) \rangle$

   This case is symmetric to the preceding one.

4. $\Delta(\neg g) = \langle \Delta^{\mathbf{f}}(g), \Delta^{\mathbf{t}}(g) \rangle$

   This is motivated by the fact that for $q = \mathtt{f}, \mathtt{t}$, σ satisfies g with truth value at least q if and only if it satisfies $\neg g$ with truth value at least $\neg q$.

5. $\Delta(\mathtt{Next}\, g) = \mathit{shift}\Delta(g)$, where $\mathit{shift}(s_0 s_1 \ldots) = \mathsf{X} s_0 s_1 \ldots$

   $s_0 s_1 s_2 \ldots$ satisfies $\mathtt{Next}\, g$ with truth value at least q if and only if $s_1 s_2 \ldots$ satisfies g with at least value q.

6. $\Delta(g_1\, \mathtt{Until}\, g_2) = \langle \Delta^{\mathbf{t}}(g_1\, \mathtt{Until}\, g_2), \Delta^{\mathbf{f}}(g_1\, \mathtt{Until}\, g_2) \rangle$, where

   - $\Delta^{\mathbf{t}}(g_1\, \mathtt{Until}\, g_2) = \cup_{i=0}^{\infty}(\Delta^{\mathbf{t}}(\mathtt{Next}^0 g_1) \amalg \ldots \amalg \Delta^{\mathbf{t}}(\mathtt{Next}^{(i-1)} g_1) \amalg \Delta^{\mathbf{t}}(\mathtt{Next}^i g_2))$
   - $\Delta^{\mathbf{f}}(g_1\, \mathtt{Until}\, g_2) = \amalg_{i=0}^{\infty}(\Delta^{\mathbf{f}}(\mathtt{Next}^0 g_1) \cup \ldots \cup \Delta^{\mathbf{f}}(\mathtt{Next}^{(i-1)} g_1) \cup \Delta^{\mathbf{f}}(\mathtt{Next}^i g_2))$

   Recall that $\mathtt{Next}^k g = g$ if $k = 0$ and $\mathtt{Next}^k g = \mathtt{Next}\,\mathtt{Next}^{k-1} g$ otherwise. Here we consider the until operator as a series of disjunctions and conjunctions and apply the motivation above when constructing the defining sequence sets.

Note that it may be that $\delta^1, \delta^2 \in \Delta^q(g)$ where $\delta^1 \sqsubseteq \delta^2$. As a practical matter it would be preferable for only δ^1 to be a member of $\Delta^q(g)$. However, this redundancy does not affect what is presented below.

An important consequence of Definition 4.7 is that for each formula g of TL, $\Delta(g)$ characterises g: all sequences that satisfy g must be greater than one of the sequences in $\Delta^{\mathbf{t}}(g)$. The lemma below formalises this (the proof is in Section A.2).

Lemma 4.3.

Let $g \in$ TL, and let $\sigma \in \mathcal{S}^{\omega}$. For $q = \mathbf{t}, \mathbf{f}$, $q \preceq Sat(\sigma, g)$ iff $\exists \delta^{g} \in \Delta^{q}(g)$ with $\delta^{g} \sqsubseteq \sigma$. □

4.3.1 Examples

The constant predicates have very simple defining sequence sets.

$$\Delta(\mathbf{t}) = \langle \{\mathsf{XX}\ldots\}, \emptyset \rangle \qquad \Delta(\bot) = \langle \emptyset, \emptyset \rangle$$

$$\Delta(\mathbf{f}) = \langle \emptyset, \{\mathsf{XX}\ldots\} \rangle \qquad \Delta(\top) = \langle \{\mathsf{XX}\ldots\}, \{\mathsf{XX}\ldots\} \rangle$$

Every sequence satisfies the predicate $\mathbf{t}$ with truth value $\mathbf{t}$, and no sequence satisfies the predicate $\mathbf{t}$ with truth value $\mathbf{f}$ or $\top$. Similarly, no sequence satisfies $\mathbf{f}$ with truth value at least $\mathbf{t}$, while all sequences satisfy $\mathbf{f}$ with truth value $\mathbf{f}$. Note that $\Delta(\mathbf{t}) = \Delta(\neg\mathbf{f})$ (indeed, it would be disconcerting if this were not the case).

Example 4.1.

Suppose that $\Delta(g) = \langle A_g, B_g \rangle$ and $\Delta(h) = \langle A_h, B_h \rangle$. Then,

$\Delta(g \vee h) = \Delta(\neg(\neg g \wedge \neg h))$. To facilitate the proof, let $rev\langle A, B \rangle = \langle B, A \rangle$.

$$\begin{aligned}
\Delta(\neg(\neg g \wedge \neg h)) &= rev\, \Delta(\neg g \wedge \neg h) \\
&= rev\langle \Delta^{\mathbf{t}}(\neg g) \amalg \Delta^{\mathbf{t}}(\neg h), \Delta^{\mathbf{f}}(\neg g) \cup \Delta^{\mathbf{f}}(\neg h) \rangle \\
&= rev\langle \Delta^{\mathbf{f}}(g) \amalg \Delta^{\mathbf{f}}(h), \Delta^{\mathbf{t}}(g) \cup \Delta^{\mathbf{t}}(h) \rangle \\
&= \langle \Delta^{\mathbf{t}}(g) \cup \Delta^{\mathbf{t}}(h), \Delta^{\mathbf{f}}(g) \amalg \Delta^{\mathbf{f}}(h) \rangle \\
&= \Delta(g \vee h)
\end{aligned}$$

Example 4.2.

$x = y$ is short for $(x \wedge y) \vee ((\neg x) \wedge (\neg y))$.

$$\begin{aligned}
\Delta(x = y) &= \langle \Delta^{\mathbf{t}}(x \wedge y \ \vee \ (\neg x \wedge \neg y)), \\
&\quad \Delta^{\mathbf{f}}(x \wedge y \ \vee \ (\neg x \wedge \neg y)) \rangle \\
&= \langle \Delta^{\mathbf{t}}(x \wedge y) \cup \Delta^{\mathbf{t}}((\neg x \wedge \neg y)), \\
&\quad \Delta^{\mathbf{f}}(x \wedge y) \amalg \Delta^{\mathbf{f}}(\neg x \wedge \neg y) \rangle \\
&= \langle (\Delta^{\mathbf{t}}(x) \amalg \Delta^{\mathbf{t}}(y)) \cup (\Delta^{\mathbf{t}}(\neg x) \amalg \Delta^{\mathbf{t}}(\neg y)), \\
&\quad (\Delta^{\mathbf{f}}(x) \cup \Delta^{\mathbf{f}}(y)) \amalg (\Delta^{\mathbf{f}}(\neg x) \cup \Delta^{\mathbf{f}}(\neg y)) \rangle \\
&= \langle (\Delta^{\mathbf{t}}(x) \amalg \Delta^{\mathbf{t}}(y)) \cup (\Delta^{\mathbf{f}}(x) \amalg \Delta^{\mathbf{f}}(y)), \\
&\quad (\Delta^{\mathbf{f}}(x) \cup \Delta^{\mathbf{f}}(y)) \amalg (\Delta^{\mathbf{t}}(x) \cup \Delta^{\mathbf{t}}(y)) \rangle
\end{aligned}$$

□

Example 4.3.

Let $\mathcal{S} = \mathcal{C}^3$; this models a circuit with three state holding components. The formula $g = (([1] \vee [2]) = \mathbf{f})$ asks whether it is true that neither component 1 nor component 2 has the H value.

$$\begin{aligned}
\Delta(\mathbf{f}) &= \langle \emptyset, \{(\mathsf{U},\mathsf{U},\mathsf{U})\ldots\} \rangle \\
\Delta([1]) &= \langle \{(\mathsf{H},\mathsf{U},\mathsf{U})(\mathsf{U},\mathsf{U},\mathsf{U})\ldots\}, \{(\mathsf{L},\mathsf{U},\mathsf{U})(\mathsf{U},\mathsf{U},\mathsf{U})\ldots\} \rangle \\
\Delta([2]) &= \langle \{(\mathsf{U},\mathsf{H},\mathsf{U})(\mathsf{U},\mathsf{U},\mathsf{U})\ldots\}, \{(\mathsf{U},\mathsf{L},\mathsf{U})(\mathsf{U},\mathsf{U},\mathsf{U})\ldots\} \rangle \\
\Delta([1] \vee [2]) &= \langle \{(\mathsf{H},\mathsf{U},\mathsf{U})(\mathsf{U},\mathsf{U},\mathsf{U})\ldots, (\mathsf{U},\mathsf{H},\mathsf{U})(\mathsf{U},\mathsf{U},\mathsf{U})\ldots\}, \\
&\quad \{(\mathsf{L},\mathsf{L},\mathsf{U})(\mathsf{U},\mathsf{U},\mathsf{U})\ldots\} \rangle \\
\Delta^{\mathbf{t}}([1] \vee [2]) = \mathbf{f}) &= \Delta^{\mathbf{t}}([1] \vee [2]) \amalg \emptyset \ \cup \ \Delta^{\mathbf{f}}([1] \vee [2]) \amalg \{(\mathsf{U},\mathsf{U},\mathsf{U})\ldots\} \\
&= \Delta^{\mathbf{f}}([1] \vee [2]) \\
&= \{(\mathsf{L},\mathsf{L},\mathsf{U})(\mathsf{U},\mathsf{U},\mathsf{U})\ldots\}
\end{aligned}$$

$$\begin{aligned}
\Delta^{\mathbf{f}}([1] \vee [2]) = \mathsf{f}) &= (\Delta^{\mathbf{f}}([1] \vee [2]) \cup \{(\mathsf{U},\mathsf{U},\mathsf{U})\ldots\}) \ \amalg\ (\Delta^{\mathbf{t}}([1] \vee [2]) \cup \emptyset) \\
&= (\Delta^{\mathbf{f}}([1] \vee [2]) \cup \{(\mathsf{U},\mathsf{U},\mathsf{U})\ldots\}) \ \amalg\ (\Delta^{\mathbf{t}}([1] \vee [2])) \\
&= \ (\{(\mathsf{L},\mathsf{L},\mathsf{U})(\mathsf{U},\mathsf{U},\mathsf{U})\ldots\} \cup \{(\mathsf{U},\mathsf{U},\mathsf{U})\ldots\}) \\
&\qquad \amalg\ \{(\mathsf{H},\mathsf{U},\mathsf{U})(\mathsf{U},\mathsf{U},\mathsf{U})\ldots, (\mathsf{U},\mathsf{H},\mathsf{U})(\mathsf{U},\mathsf{U},\mathsf{U})\ldots\} \\
&= \ \{(\mathsf{L},\mathsf{L},\mathsf{U})(\mathsf{U},\mathsf{U},\mathsf{U})\ldots, (\mathsf{U},\mathsf{U},\mathsf{U})\ldots\} \\
&\qquad \amalg\ \{(\mathsf{H},\mathsf{U},\mathsf{U})(\mathsf{U},\mathsf{U},\mathsf{U})\ldots, (\mathsf{U},\mathsf{H},\mathsf{U})(\mathsf{U},\mathsf{U},\mathsf{U})\ldots\} \\
&= \ \{(\mathsf{Z},\mathsf{L},\mathsf{U})(\mathsf{U},\mathsf{U},\mathsf{U})\ldots, (\mathsf{L},\mathsf{Z},\mathsf{U})(\mathsf{U},\mathsf{U},\mathsf{U})\ldots, \\
&\qquad (\mathsf{H},\mathsf{U},\mathsf{U})(\mathsf{U},\mathsf{U},\mathsf{U})\ldots, (\mathsf{U},\mathsf{H},\mathsf{U})(\mathsf{U},\mathsf{U},\mathsf{U})\ldots\}
\end{aligned}$$

Note that $\Delta^{\mathbf{t}}([1] \vee [2]) \sqsubseteq_{\mathcal{P}} (\Delta^{\mathbf{f}}([1] \vee [2]) \cup \{(\mathsf{U},\mathsf{U},\mathsf{U})\ldots\}) \ \amalg\ (\Delta^{\mathbf{t}}([1] \vee [2]))$ showing the redundancy in defining sequence sets. □

4.3.2 Defining Trajectory Sets

The defining sequence sets contain the set of the minimal sequences that satisfy the formula. It is possible to find the analogous structures for trajectories — we can find an approximation of the set of minimal trajectories that satisfy a formula. This section first shows how, given an arbitrary sequence, to find the weakest trajectory larger than it. Using this, the *defining trajectory sets* of a formula are defined. Finally, Theorem 4.5 is presented, which provides the basis for using defining sequence sets and defining trajectory sets to accomplish verification based on Definition 4.2.

Definition 4.8.

Let $\sigma = s_0 s_1 s_2 \ldots$. Let $\tau(\sigma) = t_0 t_1 t_2 \ldots$ where:

$$t_i = \begin{cases} s_0 & \text{when } i = 0 \\ \mathbf{Y}(t_{i-1}) \sqcup s_i & \text{otherwise} \end{cases}$$

$t_0 t_1 t_2 \ldots$ is the smallest trajectory larger than σ. s_0 is a possible starting point of a trajectory, so $t_0 = s_0$. Any run of the machine that starts in s_0 must be in a state at least as large as $\mathbf{Y}(s_0)$ after one time unit. So t_1 must be the smallest state larger than both s_1 and $\mathbf{Y}(s_0)$. By definition of join, $t_1 = \mathbf{Y}(s_0) \sqcup s_1 = \mathbf{Y}(t_0) \sqcup s_1$. This can be generalised to $t_i = \mathbf{Y}(t_{i-1}) \sqcup s_i$.

□

In the same way that there is a set of minimal sequences that satisfy a formula, there is a set of minimal trajectories that satisfy a formula. A set that contains this set of minimal trajectories can be computed from the defining sequence sets. The defining trajectory sets are computed by finding for each sequence in the defining sequence sets the smallest trajectory bigger than the sequence.

Definition 4.9 (Defining trajectory set).
$T(g) = \langle T^{\mathbf{t}}(g), T^{\mathbf{f}}(g) \rangle$, where $T^q(g) = \{\tau(\sigma) : \sigma \in \Delta^q(g)\}$. □

Note that by construction, if $\tau^g \in T^q(g)$ then there is a $\delta^g \in \Delta^q(g)$ with $\delta^g \sqsubseteq \tau^g$. $T(g)$ characterises g by characterising the trajectories that satisfy g. This is formalised in the following lemma which is proved in Section A.2.

Lemma 4.4.
Let $g \in$ TL, and let σ be a trajectory. For $q = \mathtt{t}, \mathtt{f}$, $q \preceq \mathit{Sat}(\sigma, g)$ if and only if $\exists \tau^g \in T^q(g)$ with $\tau^g \sqsubseteq \sigma$. □

The existence of defining sequence sets and defining trajectory sets provides a potentially efficient method for verification of assertions such as $g \Longrightarrow\!\!\!\!> h$. The formula g, the *antecedent*, can be used to describe initial conditions or 'input' to the system. The *consequent*, h, describes the 'output'. This method is particularly efficient when the cardinalities of the defining sets are small. This verification approach is formalised in Theorem 4.5 (which is proved in Section A.2). Section 4.1 showed how this result is used in practice. Recall that these antecedent, consequent pairs are called assertions.

Theorem 4.5.

If g and h are TL formulas, then $\Delta^{\mathtt{t}}(h) \sqsubseteq_{\mathcal{P}} T^{\mathtt{t}}(g)$ if and only if $g \Longrightarrow\!\!\!> h$. $\square$

Some formulas have small defining sequence sets with simple structure.

Definition 4.10.

If $g \in$ TL, and $\exists \delta^g \in \Delta^{\mathtt{t}}(g)$ such that $\forall \delta \in \Delta^{\mathtt{t}}(g), \delta^g \sqsubseteq \delta$, then δ^g is known as the *defining sequence* of g. If the δ^g is the defining sequence of g, then $\tau^g = \tau(\delta^g)$ is known as the *defining trajectory* of g. $\square$

Finite formulas with defining sequences are known as trajectory formulas. Seger and Bryant characterised these syntactically (see Section 2.4).

Two useful special cases of Theorem 4.5 should be noted. First, if A is a formula of TL with a well-defined defining sequence δ^A, and $h \in$ TL, then $\forall \delta \in \Delta^{\mathtt{t}}(h), \delta \sqsubseteq \tau^A$ if and only if, for every trajectory σ for which t $\preceq$ *Sat*(σ, A) it is the case that t $\preceq$ *Sat*(σ, h).

Second, let A and C be formulas of TL with well-defined defining sequences δ^A and δ^C. Then $\delta^C \sqsubseteq \tau^A$ if and only if, for every trajectory σ for which $q \preceq$ *Sat*(σ, A) it is the case that $q \preceq$ *Sat*(σ, C). This is essentially the result of Seger and Bryant generalised to the four valued logic.

4.4 Symbolic Trajectory Evaluation

The results of Section 4.3 can easily be generalised to the symbolic version of TL. The constructs used in the previous section such as defining set and so on all have symbolic extensions. Each symbolic TL formula is a concise encoding of a number of scalar formulas; each interpretation of the variables yields a (possibly) different scalar formula. To extend the theory of trajectory evaluation, symbolic sets are introduced; these can be considered as concise encodings of a number of scalar sets. Symbolic sets can be manipulated in an analogous way to scalar sets. Using this approach, the key results presented above extend to the symbolic case.

This section first presents some preliminary mathematical definitions and generalisations and then presents symbolic trajectory evaluation.

The verification conditions are extended to the symbolic case. Given two symbolic formulas g and h we are interested in for which interpretations, ϕ, it is the case that for all trajectories, σ, if σ satisfies g, then σ also satisfies h. Again, both the $\Longrightarrow\!\!\!\!>$ and $=\!\!\!\!\rhd$ relations are considered.

4.4.1 Preliminaries

Definition 4.11.

$\langle\!| \, g =\!\!\rhd h \, |\!\rangle = \{\phi \in \Phi : \forall \sigma \in \mathcal{R}_{\mathcal{T}}, \mathbf{t} = \mathit{Sat}(\sigma, \phi(g))$ implies that $\mathbf{t} = \mathit{Sat}(\sigma, \phi(h))\}$. □

Ideally such verification *assertions* should hold for all interpretations of variables.

Definition 4.12.

$\models \langle\!| \, g =\!\!\rhd h \, |\!\rangle$ denotes $\langle\!| \, g =\!\!\rhd h \, |\!\rangle = \Phi$. □

Note that $\models \langle\!| \, g =\!\!\rhd h \, |\!\rangle$ if and only if $\forall \sigma \in \mathcal{R}_{\mathcal{T}}, \mathit{SAT}_{\mathbf{t}}(\sigma, g) \subseteq \mathit{SAT}_{\mathbf{t}}(\sigma, h)$. An alternative approach is to treat inconsistency more robustly (which is what happens in STE defined on a two-valued logic). We could use these definitions.

Definition 4.13.

$\langle\!| \, g \Longrightarrow\!\!\!> h \, |\!\rangle = \{\phi \in \Phi : \forall \sigma \in \mathcal{S}_{\mathcal{T}}, \mathbf{t} \preceq \mathit{Sat}(\sigma, \phi(g))$ implies that $\mathbf{t} \preceq \mathit{Sat}(\sigma, \phi(h))\}$ □

and

Definition 4.14.

$\models \langle\!| g \Longrightarrow\!\!\!> h |\!\rangle$ denotes $\langle\!| g \Longrightarrow\!\!\!> h |\!\rangle = \Phi$. □

Note that $\models \langle\!| g \Longrightarrow\!\!\!> h |\!\rangle$ if and only if $\forall \sigma \in \mathcal{S}_\mathcal{T}, SAT_{\mathfrak{t}\uparrow}(\sigma, g) \subseteq SAT_{\mathfrak{t}\uparrow}(\sigma, h))$.

Symbolic sets are now introduced.

Definition 4.15.

Define $\mathcal{E}$, the boolean subset of TL, by

$$\mathcal{E} ::= \mathbf{t} \mid \mathbf{f} \mid \mathcal{V} \mid \mathcal{E} \wedge \mathcal{E} \mid \neg\mathcal{E}$$

This definition is used in this chapter and Chapter 5.

□

Recall that an interpretation of variables can be considered as a function mapping a symbolic TL formula to a scalar one. In particular, if ϕ is an interpretation of variables, and $a \in \mathcal{E}$, $\phi(a) \in \{\mathbf{f}, \mathbf{t}\}$.

In what follows, let $\mathcal{S}$ be a lattice over a partial order $\sqsubseteq$; this induces a lattice structure on $\mathcal{S}^\omega$; in turn, $\sqsubseteq_\mathcal{P}$ is the induced preorder on $\mathcal{P}(\mathcal{S}^\omega)$ defined earlier.

Definition 4.16.

A symbolic set over a domain $\mathcal{P}(\mathcal{S}^\omega)$ is one of

1. $A \in \mathcal{P}(\mathcal{S}^\omega)$;
2. $a \to \dot{A}$ where $a \in \mathcal{E}$;
3. $\dot{A}_1 \dot{\cup} \dot{A}_2$, where $\dot{A}_1, \dot{A}_2$ are symbolic sets;
4. $\dot{A}_1 \dot{\cap} \dot{A}_2$, where $\dot{A}_1, \dot{A}_2$ are symbolic sets; or
5. $\dot{A}_1 \dot{\amalg} \dot{A}_2$, where $\dot{A}_1, \dot{A}_2$ are symbolic sets.

□

Each symbolic set represents a number of sets. Each interpretation of variables, yields a scalar set contained in $\mathcal{P}(\mathcal{S}^\omega)$. Given an interpretation of variables, there is a natural interpretation of symbolic sets, given below.

Definition 4.17.

Let $\phi \in \Phi$ be given.

1. $\phi(A) = A$ for all $A \in \mathcal{P}(\mathcal{S}^\omega)$;
2. $\phi(a \to \dot{A}) = \{x \in \dot{A} : \phi(a) = \mathtt{t}\}$. Thus if a evaluates to $\mathtt{t}$, then $a \to \dot{A}$ is the set $\dot{A}$, otherwise it is the empty set.
3. $\dot{A} \,\dot{\amalg}\, \dot{B}$ is defined by $\phi(\dot{A} \,\dot{\amalg}\, \dot{B}) = \phi(\dot{A}) \amalg \phi(\dot{B})$.
4. If $f\colon \mathcal{P}(\mathcal{S}^\omega)^m \to \mathcal{P}(\mathcal{S}^\omega)$, then the symbolic version of f is defined by
   $$\phi(\dot{f}(\dot{A}_1, \ldots, \dot{A}_n)) = f(\phi(\dot{A}_1), \ldots, \phi(\dot{A}_n)).$$
   These definitions can be used in extending set operations such as set union, as well as for more general functions, for example in extending Definition 4.9 to give a definition of symbolic defining trajectory sets.
5. $\dot{A} \,\dot{\sqsubseteq}_{\mathcal{P}}\, \dot{B} = \{\phi \in \Phi : \phi(\dot{A}) \sqsubseteq_{\mathcal{P}} \phi(\dot{B})\}$.

□

The following lemma shows that these definitions are sensible.

Lemma 4.6.

Let $\dot{A}, \dot{B}, \dot{C}$ be symbolic sets over domain $\mathcal{P}(\mathcal{S}^\omega)$.

1. $(\dot{A} \,\dot{\sqsubseteq}\, \dot{A} \dot{\amalg} \dot{B}) = \Phi$.
2. If $(\dot{A} \,\dot{\sqsubseteq}\, \dot{C}) = \Phi$ and $(\dot{B} \,\dot{\sqsubseteq}\, \dot{C}) = \Phi$, then $(\dot{A} \dot{\amalg} \dot{B} \,\dot{\sqsubseteq}\, \dot{C}) = \Phi$.

Proof.

Let ϕ be arbitrary.

(1) $\phi(\dot{A}) \sqsubseteq \phi(\dot{A}) \amalg \phi(\dot{B})$ Property of $\amalg$.

(2) $\phi(\dot{A}) \sqsubseteq \phi(\dot{A} \dot{\amalg} \dot{B})$ Definition.

Since ϕ is arbitrary part 1 follows.

(3) $\phi(\dot{A}), \phi(\dot{B}) \sqsubseteq \phi(\dot{C})$ Hypothesis 2.

(4) $\phi(\dot{A}) \amalg \phi(\dot{B}) \sqsubseteq \phi(\dot{C})$ From (3), by property of join.

(5) $\phi(\dot{A} \dot{\amalg} \dot{B}) \sqsubseteq \phi(\dot{C})$ Definition.

Since ϕ is arbitrary, part 2 follows.

□

4.4.2 Symbolic Defining Sequence Sets

Given this mathematical machinery, symbolic defining sequence sets can now be defined. The definition of defining sequence sets (Definition 4.7) must be extended by using the symbolic versions of set union and join etc. In addition, one more part must be added to the definition to take into account formulas of TL containing variables. For completeness, this definition is given below.

Definition 4.18 (Extension to Definition 4.7).

Let $\dot{g} \in$ TL. Define the *symbolic defining sequence sets* of $\dot{g}$ as $\dot{\Delta}(\dot{g}) = \langle \dot{\Delta}^{\mathbf{t}}(\dot{g}), \dot{\Delta}^{\mathbf{f}}(\dot{g}) \rangle$, where the $\dot{\Delta}^{q}(\dot{g})$ are defined recursively by:

1. If $\dot{g}$ is simple, $\dot{\Delta}^{q}(\dot{g}) = \{s\mathsf{XX}\ldots : (s, q) \in D_{\dot{g}}$, or $(s, \top) \in D_{\dot{g}}\}$.
2. $\dot{\Delta}(\dot{g}_1 \vee \dot{g}_2) = \langle \dot{\Delta}^{\mathbf{t}}(\dot{g}_1) \dot{\cup} \dot{\Delta}^{\mathbf{t}}(\dot{g}_2), \dot{\Delta}^{\mathbf{f}}(\dot{g}_1) \dot{\amalg} \dot{\Delta}^{\mathbf{f}}(\dot{g}_2) \rangle$
3. $\dot{\Delta}(\dot{g}_1 \wedge \dot{g}_2) = \langle \dot{\Delta}^{\mathbf{t}}(\dot{g}_1) \dot{\amalg} \dot{\Delta}^{\mathbf{t}}(\dot{g}_2), \dot{\Delta}^{\mathbf{f}}(\dot{g}_1) \dot{\cup} \dot{\Delta}^{\mathbf{f}}(\dot{g}_2) \rangle$
4. $\dot{\Delta}(\neg \dot{g}) = \langle \dot{\Delta}^{\mathbf{f}}(\dot{g}), \dot{\Delta}^{\mathbf{t}}(\dot{g}) \rangle$
5. $\dot{\Delta}(\texttt{Next}\, \dot{g}) = \dot{shift}\, \dot{\Delta}(\dot{g})$
6. $\dot{\Delta}(\dot{g}_1\, \texttt{Until}\, \dot{g}_2) = \langle \dot{\Delta}^{\mathbf{t}}(\dot{g}_1\, \texttt{Until}\, \dot{g}_2), \dot{\Delta}^{\mathbf{f}}(\dot{g}_1\, \texttt{Until}\, \dot{g}_2) \rangle$, where

- $\dot{\Delta}^{\mathbf{t}}(\dot{g}_1\,\texttt{Until}\,\dot{g}_2) = \dot{\cup}_{i=0}^{\infty}(\dot{\Delta}^{\mathbf{t}}(\texttt{Next}^0\dot{g}_1)\dot{\amalg}\ldots\dot{\amalg}\dot{\Delta}^{\mathbf{t}}(\texttt{Next}^{(i-1)}\dot{g}_1)\dot{\amalg}\dot{\Delta}^{\mathbf{t}}(\texttt{Next}^{i}\dot{g}_2))$
- $\dot{\Delta}^{\mathbf{f}}(\dot{g}_1\,\texttt{Until}\,\dot{g}_2) = \dot{\amalg}_{i=0}^{\infty}(\dot{\Delta}^{\mathbf{f}}(\texttt{Next}^0\dot{g}_1)\dot{\cup}\ldots\dot{\cup}\dot{\Delta}^{\mathbf{f}}(\texttt{Next}^{(i-1)}\dot{g}_1)\dot{\cup}\dot{\Delta}^{\mathbf{f}}(\texttt{Next}^{i}\dot{g}_2))$

7. If $v \in \mathcal{V}$, $\dot{\Delta}(v) = \langle v \to \{\mathsf{XX}\ldots\}, \neg v \to \{\mathsf{XX}\ldots\}\rangle$.

   If $\phi(v) = \mathbf{t}$, then $\phi(\dot{\Delta}(v)) = \Delta(\mathbf{t})$, and if $\phi(v) = \mathbf{f}$, then $\phi(\dot{\Delta}(v)) = \Delta(\mathbf{f})$.

The extension of the definition of defining trajectory set is straightforward.

Definition 4.19 (Symbolic defining trajectory set).
$\dot{T}^q(\dot{g}) = \dot{\tau}(\dot{\Delta}(\dot{g}))$.

The main result of symbolic trajectory evaluation is based on Theorem 4.5. It says that the set of interpretations $\langle\!|\, \dot{g} \Longrightarrow\!\!\!> \dot{h} \,|\!\rangle$ (those interpretations, ϕ, such that $\phi(g) \Longrightarrow\!\!\!> \phi(h)$. is exactly the same as the set of interpretations for which $\dot{\Delta}^{\mathbf{t}}(\dot{h}) \;\dot{\sqsubseteq}_{\mathcal{P}}\; \dot{T}^{\mathbf{t}}(\dot{g})$. So, if we can compute one, then we can compute the other.

Theorem 4.7.
Let $\dot{g}, \dot{h}$ be TL formulas. $\langle\!|\, \dot{g} \Longrightarrow\!\!\!> \dot{h} \,|\!\rangle = (\dot{\Delta}^{\mathbf{t}}(\dot{h}) \;\dot{\sqsubseteq}_{\mathcal{P}}\; \dot{T}^{\mathbf{t}}(\dot{g}))$

Proof.

$$\begin{aligned}
\langle\!|\, \dot{g} \Longrightarrow\!\!\!> \dot{h} \,|\!\rangle &= \{\phi \in \Phi : \forall \sigma \in \mathcal{S}_{\mathcal{T}}, \mathbf{t} \preceq \mathit{Sat}(\sigma, \phi(\dot{g})) \text{ implies that } \mathbf{t} \preceq \mathit{Sat}(\sigma, \phi(\dot{h}))\} \\
&= \{\phi \in \Phi : \Delta^{\mathbf{t}}(\phi(\dot{h})) \sqsubseteq_{\mathcal{P}} T^{\mathbf{t}}(\phi(\dot{g}))\} \quad \text{(Theorem 4.5)} \\
&= \dot{\Delta}^{\mathbf{t}}(\dot{h}) \;\dot{\sqsubseteq}_{\mathcal{P}}\; \dot{T}^{\mathbf{t}}(\dot{g}) \quad \text{(By definition.)}
\end{aligned}$$

This is an important result, because efficient methods of computing these symbolic sets and performing the trajectory evaluation exist, and have been implemented in the Voss tool discussed in Chapter 6. This forms the basis of the verification presented here.

4.4.3 Circuit Models

When $\mathcal{S} = \mathcal{C}^n$ and $\mathcal{R} = \{\mathsf{U}, \mathsf{L}, \mathsf{H}\}^n$, and only the realisable fragment of TL_n (those TL_n formulas not syntactically containing $\top$) are considered, computing these verification results is simplified. In the rest of this section we only consider the realisable fragment of TL_n. Two important properties of the realisable fragment of TL_n are given in the next lemma.

Lemma 4.8.

1. $\sigma \in \mathcal{R}_{\mathcal{T}}$ if and only if Z does not appear in σ (for all i, j, $\sigma_i[j] \neq \mathsf{Z}$).
2. If $\sigma \in \mathcal{R}_{\mathcal{T}}$ and g is in the realisable fragment of TL_n,

$$\mathit{SAT}_{\top}(\sigma, g) = \emptyset \text{ and } \mathit{SAT}_{\mathbf{t}}(\sigma, g) = \mathit{SAT}_{\mathbf{t}\uparrow}(\sigma, g)$$

Proof.

The proof of (1) comes from the definition of $\mathcal{R}$. For (2), recall from Section 3.4 that *Sat*$(\sigma, g) = \top$ only if g contains a subformula $g' \in G_n$ which is either the constant predicate $\top$ or if Z appears in σ. $\square$

We compute $\models \langle\!|\, g \Longrightarrow\!\!\!\triangleright\, h \,|\!\rangle$ as follows. First, compute $T^{\mathbf{t}}(g)$. It is easy to determine whether $T^{\mathbf{t}}(g) \subseteq \mathcal{R}_{\mathcal{T}}$ using Lemma 4.8(1). If not, then there are inconsistencies in the antecedent which should be flagged for the user to deal with before verification continues. Thus we may assume that $T^{\mathbf{t}}(g) \subseteq \mathcal{R}_{\mathcal{T}}$.

$$\begin{aligned}
&\models \langle\!|\, g \Longrightarrow\!\!\!> h \,|\!\rangle = \\
&\qquad \forall \sigma \in \mathcal{S}_{\mathcal{T}}, (\mathit{SAT}_{\mathbf{t}\uparrow}(\sigma, g) \subseteq \mathit{SAT}_{\mathbf{t}\uparrow}(\sigma, h)) \quad \text{(By definition)} \\
&\qquad \Longrightarrow \forall \sigma \in \mathcal{R}_{\mathcal{T}}, (\mathit{SAT}_{\mathbf{t}\uparrow}(\sigma, g) \subseteq \mathit{SAT}_{\mathbf{t}\uparrow}(\sigma, h)) \quad \text{(since } \mathcal{R}_{\mathcal{T}} \subseteq \mathcal{S}_{\mathcal{T}}) \\
&\qquad \Longrightarrow \forall \sigma \in \mathcal{R}_{\mathcal{T}}, (\mathit{SAT}_{\mathbf{t}}(\sigma, g) \subseteq \mathit{SAT}_{\mathbf{t}}(\sigma, h)) \quad \text{(By Lemma 4.8(2).)} \\
&\qquad = \models \langle\!|\, g \Longrightarrow\!\!\!\triangleright\, h \,|\!\rangle
\end{aligned}$$

This result is useful because in this important special case, efficient STE-based algorithms can be used. The rest of this thesis uses this result implicitly. The main computational task is to determine $\models \langle\!| \, g \Longrightarrow\!\!\!> h \, |\!\rangle$. By placing sensible restrictions on the logic used and checking for inconsistency in the defining trajectory set of the antecedent, we can then deduce $\models \langle\!| \, g \Longrightarrow\!\!\!\rhd h \, |\!\rangle$ from $\models \langle\!| \, g \Longrightarrow\!\!\!> h \, |\!\rangle$.

Chapter 5

A Compositional Theory for TL

5.1 Motivation

Although STE is an efficient method of model checking, it suffers from the same inherent performance problems that other model checking algorithms do. Enriching the logic that STE supports, as is proposed in previous chapters, potentially exacerbates the problem. A primary thesis of this research is that a compositional theory for TL can overcome performance limitations of automatic model checking. Compositionality provides a method of divide-and-conquer: the problem can be broken into smaller sub-problems, the sub-problems solved using automatic model checking, and the overall result proved using the compositionality theory. This chapter presents the compositional theory for trajectory evaluation, which is a set of sound inference rules for deducing the correctness of verification assertions. Chapter 6 discusses the development of a practical tool that can use this compositional theory — this allows the use of an integrated theorem prover/model checker with useful practical implementations.

As discussed in Chapter 1, the focus of this theory is property composition: Section 5.2 presents compositional rules for TL; Section 5.3 presents additional compositional rules for TL_n; and practical considerations are presented in Section 5.4. Structural composition is briefly discussed in Section B.1.

5.2 Compositional Rules for the Logic

This section presents the main compositional theory with each rule being presented and proved in turn. The compositional theory is developed for the $\Longrightarrow\!\!\!\!>$ relation. In general, the theory does not apply to the $=\!\!\!\triangleright$ relation since the disjunction, consequence, transitivity and until rules do not hold for this relation. However, the other composition rules do apply for $=\!\!\!\triangleright$ (in the related theorems below, replacing $\Longrightarrow\!\!\!\!>$ with $=\!\!\!\triangleright$ and $\preceq$ with $=$, and considering only trajectories in $\mathcal{R}_{\mathcal{T}}$ will yield the desired result). Moreover, as shown in Section 5.3, the full compositional theory does hold for the $=\!\!\!\triangleright$ relation for the important realisable class of TL_n

The circuit shown in Figure 5.1 will be used in the rest of this section to illustrate the use of the inference rules. The circuit is very simple and can easily be dealt with directly by STE, but the smallness of the circuit helps the clarity of the example. A unit-delay model is used for inverter and gate delays. *Notation:* $[B]$ is the simple predicate which evaluates to $\top$ when the state component B has the value Z, $\mathtt{t}$ when B has the value H, $\mathtt{f}$ when B has the value L, $\bot$ when B has the value U.

Note that except for the Specialisation Rule, all of the proofs are for the scalar case only as this simplifies the proof. However, as a symbolic formula is merely shorthand for a set of scalar formulas, the rules for the symbolic case follow directly in all cases.

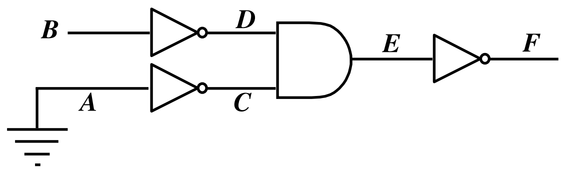


Figure 5.1: Example

5.2.1 Identity Rule

This rule is a trivial technical rule. However, it turns out to be useful in practice where a sequence of inference rules will be used to perform a verification. In the practical system which implements this theory, the proofs are written as a program script, and this rule is useful to initialise the process. Its advantage is that it makes the program slightly more elegant.

Theorem 5.1.
For all $g \in$ TL, $g \Longrightarrow\!\!\!\!> g$.

Proof. Let $\mathrm{t} \preceq$ *Sat*(σ, g). Clearly then $\mathrm{t} \preceq$ *Sat*(σ, g). Hence $g \Longrightarrow\!\!\!\!> g$. □

5.2.2 Time-shift Rule

The time-shift rule is important because it allows abstraction from the exact times things happen at. This may reduce the amount of detail that the human verifier will have to deal with, and more importantly, allows verification results to be reused a number of times. In practice this is very important in making verification efficient.

Lemma 5.2.
Suppose $g \Longrightarrow\!\!\!\!> h$. Then $\texttt{Next}\, g \Longrightarrow\!\!\!\!> \texttt{Next}\, h$

Proof. Let $\sigma = s_0 s_1 s_2 \ldots$ be a sequence such that $\mathrm{t} \preceq$ *Sat*$(\sigma, \texttt{Next}\, g)$.

(1)	$\mathrm{t} \preceq$ *Sat*$(\sigma_{\geq 1}, g)$	By definition of the satisfaction relation.
(2)	$\mathrm{t} \preceq$ *Sat*$(\sigma_{\geq 1}, h)$	Since $g \Longrightarrow\!\!\!\!> h$.
(3)	$\mathrm{t} \preceq$ *Sat*$(\sigma_{\geq 0}, \texttt{Next}\, h)$	Definition of satisfaction of `Next` .
(4)	Thus $\texttt{Next}\, g \Longrightarrow\!\!\!\!> \texttt{Next}\, h$.	

□

Theorem 5.3 follows directly from Lemma 5.2 by induction.

Theorem 5.3.

Suppose $g \Longrightarrow\!\!\!> h$. Then $\forall t \geq 0,\ \texttt{Next}^t g \Longrightarrow\!\!\!> \texttt{Next}^t h$.

Example 5.1.

In the circuit of Figure 5.1, using STE, it can be shown that $[B] \Longrightarrow\!\!\!> \texttt{Next}\,(\neg[D])$. Using Theorem 5.3, we can deduce that $\forall t \geq 0$, $\texttt{Next}^t[B] \Longrightarrow\!\!\!> \texttt{Next}^{(t+1)}\,(\neg[D])$.

The requirement of Theorem 5.3 that $t \geq 0$ is necessary: in general it does not hold when $t < 0$. For example, in our circuit we can prove that $\texttt{Next}^1[D] \Longrightarrow\!\!\!> \texttt{Next}^2[E]$. However, it is not the case that $\texttt{Next}^0[D] \Longrightarrow\!\!\!> \texttt{Next}^1[E]$. In the former case, the node C has the value H at time 1 because A is connected to ground; at time 0 we know nothing of the value of C.

5.2.3 Conjunction Rule

Conjunction and disjunction allow the combination of separately proved results. This is particularly useful where properties of different parts of the system being verified have been proved and need to be combined. Given two results $g_1 \Longrightarrow\!\!\!> h_1$ and $g_2 \Longrightarrow\!\!\!> h_2$, the two antecedents are combined into one antecedent and the two consequents are combined into one consequent. Using the conjunction rule, combination is done using the $\land$ operator, and using the disjunction rule, combination is done using the $\lor$ operator. There is no need for the g_i and h_i to be 'independent', i.e. they can share common sub-formulas.

Theorem 5.4.

Suppose $g_1 \Longrightarrow\!\!\!> h_1$ and $g_2 \Longrightarrow\!\!\!> h_2$.

Then $g_1 \land g_2 \Longrightarrow\!\!\!> h_1 \land h_2$.

Proof. Let $\sigma \in \mathcal{S}^\omega$ and suppose $\mathrm{t} \preceq \mathit{Sat}(\sigma, g_1 \wedge g_2)$.

(1)	$\mathrm{t} \preceq \mathit{Sat}(\sigma, g_1) \wedge \mathit{Sat}(\sigma, g_2)$	Definition of $\mathit{Sat}(\sigma, g_1 \wedge g_2)$.
(2)	$\mathrm{t} \preceq \mathit{Sat}(\sigma, g_i),\ i = 1, 2$	Lemma 3.2(2).
(3)	$\mathrm{t} \preceq \mathit{Sat}(\sigma, h_i),\ i = 1, 2$	Since $g_i \Longrightarrow\!\!> h_i,\ i = 1, 2$.
(4)	$\mathrm{t} \preceq \mathit{Sat}(\sigma, h_1) \wedge \mathit{Sat}(\sigma, h_2)$	Lemma 3.2(2).
(5)	$\mathrm{t} \preceq \mathit{Sat}(\sigma, h_1 \wedge h_2)$	Definition of $\mathit{Sat}(\sigma, h_1 \wedge h_2)$.

As σ is arbitrary, $g_1 \wedge g_2 \Longrightarrow\!\!> h_1 \wedge h_2$. □

Example 5.2.

In the circuit shown in Figure 5.1, we can show using STE that

$$\neg[B] \Longrightarrow\!\!> \texttt{Next}\ [D]$$

$$\neg[A] \Longrightarrow\!\!> \texttt{Next}\ [C].$$

Using Theorem 5.4 we have that:

$$\neg[A] \wedge \neg[B] \Longrightarrow\!\!> \texttt{Next}\ [C] \wedge \texttt{Next}\ [D].$$

5.2.4 Disjunction Rule

Theorem 5.5.

Suppose $g_1 \Longrightarrow\!\!> h_1$ and $g_2 \Longrightarrow\!\!> h_2$.

Then $g_1 \vee g_2 \Longrightarrow\!\!> h_1 \vee h_2$.

Proof. Let $\sigma \in \mathcal{S}^\omega$ and suppose $\mathrm{t} \preceq \mathit{Sat}(\sigma, g_1 \vee g_2)$.

(1)	$\mathrm{t} \preceq \mathit{Sat}(\sigma, g_1) \vee \mathit{Sat}(\sigma, g_2)$	Definition of $\mathit{Sat}(\sigma, g_1 \vee g_2)$.
(2)	$\mathrm{t} \preceq \mathit{Sat}(\sigma, g_i)$, for $i = 1$ or $i = 2$	Lemma 3.2(1).
(3)	$\mathrm{t} \preceq \mathit{Sat}(\sigma, h_i)$, for $i = 1$ or $i = 2$	Since $g_i \Longrightarrow\!\!> h_i,\ i = 1, 2$.
(4)	$\mathrm{t} \preceq \mathit{Sat}(\sigma, h_1) \vee \mathit{Sat}(\sigma, h_2)$	Lemma 3.2(1).
(4)	$\mathrm{t} \preceq \mathit{Sat}(\sigma, h_1 \vee h_2)$	Definition of $\mathit{Sat}(\sigma, h_1 \vee h_2)$.

As σ is arbitrary, $g_1 \vee g_2 \Longrightarrow\!\!> h_1 \vee h_2$. □

Example 5.3.

In the circuit in Figure 5.1, we can use STE to show that

$$\neg[D] \Longrightarrow\!\!\!> \texttt{Next}\,\neg[E]$$

$$\neg[C] \Longrightarrow\!\!\!> \texttt{Next}\,\neg[E].$$

Using Theorem 5.5, we have that

$$(\neg[D] \ \vee\ \neg[C]) \Longrightarrow\!\!\!> \texttt{Next}\,\neg[E].$$

Although the consequents of both premisses used here in the disjunct rule are the same, in general they may be different.

5.2.5 Rules of Consequence

The rules of consequence have two main purposes:

- Rewriting antecedents and consequents into syntactically different but semantically equivalent forms (see Example 5.4);
- Removing information which is not needed for subsequent steps in the proof so as to reduce clutter (see Example 5.5).

The next lemma is an auxiliary result. Informally it says that if the defining sequence sets of g and h are ordered with respect to each other, then every sequence that satisfies h also satisfies g.

Lemma 5.6.

Suppose $\Delta^{\mathbf{t}}(g) \sqsubseteq_{\mathcal{P}} \Delta^{\mathbf{t}}(h)$ and $\mathbf{t} \preceq \mathit{Sat}(\sigma, h)$.

Then $\mathbf{t} \preceq \mathit{Sat}(\sigma, g)$.

Proof.

(1)	$\exists \delta \in \Delta^{\mathbf{t}}(h) \ni \delta \sqsubseteq \sigma$ and $\mathbf{t} \preceq \mathit{Sat}(\delta, h)$	Lemma 4.3.
(2)	$\exists \delta' \in \Delta^{\mathbf{t}}(g) \ni \delta' \sqsubseteq \delta$	Definition of $\sqsubseteq_{\mathcal{P}}$.
(3)	$\mathbf{t} \preceq \mathit{Sat}(\delta', g)$	Lemma 4.3.
(4)	$\delta' \sqsubseteq \sigma$	Transitivity of (1) and (2).
(5)	$\mathbf{t} \preceq \mathit{Sat}(\sigma, g)$	From (3) and (4) by Lemma 4.3.

□

The intuition behind this is that if $\Delta^{\mathbf{t}}(g) \sqsubseteq_{\mathcal{P}} \Delta^{\mathbf{t}}(h)$, then any sequence that satisfies h will also satisfy g. Given this result, the rules of consequence are easy to prove.

Theorem 5.7.

Suppose $g \Longrightarrow\!\!\!> h$ and $\Delta^{\mathbf{t}}(g) \sqsubseteq_{\mathcal{P}} \Delta^{\mathbf{t}}(g_1)$ and $\Delta^{\mathbf{t}}(h_1) \sqsubseteq_{\mathcal{P}} \Delta^{\mathbf{t}}(h)$.

Then $g_1 \Longrightarrow\!\!\!> h_1$.

Proof. Suppose σ is a trajectory such that $\mathbf{t} \preceq \mathit{Sat}(\sigma, g_1)$.

(1)	$\mathbf{t} \preceq \mathit{Sat}(\sigma, g)$	Lemma 5.6.
(2)	$\mathbf{t} \preceq \mathit{Sat}(\sigma, h)$	$g \Longrightarrow\!\!\!> h$.
(3)	$\mathbf{t} \preceq \mathit{Sat}(\sigma, h_1)$	Lemma 5.6.
(4)	$g_1 \Longrightarrow\!\!\!> h_1$	Since σ is arbitrary.

□

Example 5.4.

Using this theorem to rewrite one assertion into a semantically equivalent one can be illustrated by examining the result of Example 5.2:

$$(\neg[A] \wedge \neg[B]) \Longrightarrow\!\!\!> (\mathtt{Next}\,[C] \wedge \mathtt{Next}\,[D]).$$

Since conjunction can be distributed over the next-time operator, as $\mathtt{Next}\,[C] \wedge \mathtt{Next}\,[D] \equiv \mathtt{Next}\,([C] \wedge [D])$, this can be rewritten as:

$$(\neg[A] \wedge \neg[B]) \Longrightarrow\!\!\!> \mathtt{Next}\,([C] \wedge [D]).$$

Example 5.5.

In the circuit of Figure 5.1, we can show using STE that

$$([B] \wedge \texttt{Next}\,[B]) \mathrel{\Longrightarrow\!\!\!\!\gg} \texttt{Next}^1(\neg D) \wedge \texttt{Next}^2(\neg D).$$

Using Theorem 5.7, we can refine this to

$$([B] \wedge \texttt{Next}\,[B]) \mathrel{\Longrightarrow\!\!\!\!\gg} \texttt{Next}^1(\neg D).$$

5.2.6 Transitivity

The rule of transitivity is an analogue of the transitivity rule of logic: it gives the condition for deducing from $g_1 \mathrel{\Longrightarrow\!\!\!\!\gg} h_1$ and $g_2 \mathrel{\Longrightarrow\!\!\!\!\gg} h_2$ that $g_1 \mathrel{\Longrightarrow\!\!\!\!\gg} h_2$. This condition is that $\Delta^{\mathtt{t}}(g_2) \sqsubseteq_{\mathcal{P}} \Delta^{\mathtt{t}}(g_1) \amalg \Delta^{\mathtt{t}}(h_1)$. Note that this is a weaker condition than showing that $\Delta^{\mathtt{t}}(g_2) \sqsubseteq_{\mathcal{P}} \Delta^{\mathtt{t}}(h_1)$.

Theorem 5.8.

Suppose $g_1 \mathrel{\Longrightarrow\!\!\!\!\gg} h_1$ and $g_2 \mathrel{\Longrightarrow\!\!\!\!\gg} h_2$ and that $\Delta^{\mathtt{t}}(g_2) \sqsubseteq_{\mathcal{P}} \Delta^{\mathtt{t}}(g_1) \sqcup \Delta^{\mathtt{t}}(h_1)$.

Then $g_1 \mathrel{\Longrightarrow\!\!\!\!\gg} h_2$.

Proof. Suppose σ is a trajectory such that $\mathtt{t} \preceq \mathit{Sat}(\sigma, g_1)$.

(1)	$\mathtt{t} \preceq \mathit{Sat}(\sigma, h_1)$	$g_1 \mathrel{\Longrightarrow\!\!\!\!\gg} h_1$
(2)	$\mathtt{t} \preceq \mathit{Sat}(\sigma, g_1 \wedge h_1)$	Definition of $\mathit{Sat}(\sigma, g_1 \wedge h_1)$.
(3)	$\exists \delta \in \Delta^{\mathtt{t}}(g_1 \wedge h_1) \ni \delta \sqsubseteq \sigma$	Lemma 4.3.
(4)	$\Delta^{\mathtt{t}}(g_1 \wedge h_1) = \Delta^{\mathtt{t}}(g_1) \amalg \Delta^{\mathtt{t}}(h_1)$	By definition of $\Delta^{\mathtt{t}}$.
(5)	$\exists \delta' \in \Delta^{\mathtt{t}}(g_2) \ni \delta' \sqsubseteq \delta$	$\Delta^{\mathtt{t}}(g_2) \sqsubseteq_{\mathcal{P}} \Delta^{\mathtt{t}}(g_1) \amalg \Delta^{\mathtt{t}}(h_1)$.
(6)	$\delta' \sqsubseteq \sigma$	Applying transitivity to (3) and (5).
(7)	$\mathtt{t} \preceq \mathit{Sat}(\delta', g_2)$	From (5) by Lemma 4.3.
(8)	$\mathtt{t} \preceq \mathit{Sat}(\sigma, g_2)$	From (6), (7) by monotonicity.

(9) $\quad \mathtt{t} \preceq \mathit{Sat}(\sigma, h_2) \quad g_2 \Longrightarrow\!\!\!\!> h_2.$

(10) $\quad g_1 \Longrightarrow\!\!\!\!> h_2 \quad$ Since σ was arbitrary.

□

Example 5.6.

Using STE, we can prove the following about the circuit of Figure 5.1:

- $\neg[B] \Longrightarrow\!\!\!\!> \mathtt{Next}^2[E]$
- $\mathtt{Next}^2[E] \Longrightarrow\!\!\!\!> \mathtt{Next}^3(\neg[F])$

Then, using Theorem 5.8, we can deduce $\neg[B] \Longrightarrow\!\!\!\!> \mathtt{Next}^3(\neg[F])$.

5.2.7 Specialisation

Specialisation is one of the key inference rules. By using specialisation it is possible to generate a large number of specific results from one general result. With STE, it is often cheaper to prove a more general result than a more specialised result. Thus in some cases, it may be cheaper to generate a more general result than needed and then to specialise this general result than to use STE to obtain the result directly. Specialisation also promotes the re-use of results. It is often used together with transitivity: before applying transitivity to combine two assertions, one or both of the assertions are first specialised.

For example, a general proof of the correctness of an adder is straightforward to obtain using trajectory evaluation, even for large bit widths. Such a proof may show that if bit vectors representing the numbers a and b are given as inputs to the circuit, then a few time steps later the bit-vector representing $a + b$ emerges as output. There are two reasons why one might want to specialise such a proof:

- If the adder is part of a large circuit the actual inputs may be bit-vectors representing complex mathematical expressions. Since STE relies on representing bit-vectors with BDDs, if the BDDs needed to represent these mathematical expressions are very large, it may not be possible to use STE to prove that the adder works correctly for the particular inputs. The solution is to prove that the adder works correctly for the general case, and then to specialise the result appropriately.

- The adder may be used a number of times in a computation, each with different input values. Instead of proving the correctness of the circuit for each set of inputs, the proof can be done once and then the specific results needed can be obtained by specialisation (and, probably, time-shifting).

Recall the definition of the boolean subset of TL presented as Definition 4.15.

Definition 5.1.

Define $\mathcal{E}$, a subset of TL, by

$$\mathcal{E} ::= \mathbf{t} \mid \mathbf{f} \mid \mathcal{V} \mid \mathcal{E} \wedge \mathcal{E} \mid \neg \mathcal{E}$$

Definition 5.2.

1. $\xi \colon \mathcal{V} \to \mathcal{E}$ is a *substitution*.
2. A substitution $\xi \colon \mathcal{V} \to \mathcal{E}$ can be extended to map from TL to TL:
   - $\xi(g_1 \wedge g_2) = \xi(g_1) \wedge \xi(g_2)$
   - $\xi(\neg g) = \neg \xi(g)$
   - $\xi(\texttt{Next}\, g) = \texttt{Next}\,(\xi(g))$
   - $\xi(g\,\texttt{Until}\,h) = \xi(g)\,\texttt{Until}\,\xi(h)$
   - Otherwise, if g is not a variable, $\xi(g) = g$

If T is the assertion $\models \langle\!| \, g \Longrightarrow\!\!> h \, |\!\rangle$ then $\xi(T)$ is the assertion $\models \langle\!| \, \xi(g) \Longrightarrow\!\!> \xi(h) \, |\!\rangle$.

Lemma 5.9 (Substitution Lemma).

Suppose $\models \langle\!\langle\, g \Longrightarrow\!\!\!> h \,\rangle\!\rangle$ and let ξ be a substitution.

Then $\models \langle\!\langle\, \xi(g) \Longrightarrow\!\!\!> \xi(h) \,\rangle\!\rangle$.

Proof. Let ϕ by an arbitrary interpretation of variables and σ be an arbitrary trajectory such that $\mathrm{t} \preceq \mathit{Sat}(\sigma, \phi(\xi(g)))$.

(1)	Let $\phi' = \phi \circ \xi$	
(2)	$\mathrm{t} \preceq \mathit{Sat}(\sigma, \phi'(g))$	Rewriting supposition.
(3)	ϕ' is an interpretation of variables	By construction.
(4)	$\mathrm{t} \preceq \mathit{Sat}(\sigma, \phi'(h))$	$\models \langle\!\langle\, g \Longrightarrow\!\!\!> h \,\rangle\!\rangle$.
(5)	$\mathrm{t} \preceq \mathit{Sat}(\sigma, \phi(\xi(h)))$	Rewriting (4).
(6)	$\models \langle\!\langle\, \xi(g) \Longrightarrow\!\!\!> \xi(h) \,\rangle\!\rangle$	ϕ and σ were arbitrary.

□

Example 5.7.

Suppose that part of a circuit multiplies two 64-bit numbers together and then compares the result to some 128-bit number. Let c be the boolean expression that this part of the circuit computes — in general it will not be possible to represent c efficiently since the BDD needed to represent c will be extremely large. Now suppose that the next step in the circuit is to invert c. We may wish to prove that

$T_1 = \;\models \langle\!\langle\, [B] = c \Longrightarrow\!\!\!> \texttt{Next}\,([D] = \neg c) \,\rangle\!\rangle$ is true.

Given that c is so large, it will not be possible to use STE directly to do this. But, let

$T_2 = \;\models \langle\!\langle\, [B] = a \Longrightarrow\!\!\!> \texttt{Next}\,([D] = \neg a) \,\rangle\!\rangle$

where a is a variable (an element of $\mathcal{V}$).

Proving that T_2 holds using STE is trivial. Having proved T_2, we can easily prove T_1 using

Lemma 5.9. Let

$$\xi(v) = \begin{cases} c & \text{when } v = a \\ v & \text{otherwise} \end{cases}$$

be a substitution. Note that $T_1 = \xi(T_2)$. As T_2 holds, and as ξ is a substitution, by Lemma 5.9, T_1 holds too.

Although substitution is useful, in practice sometimes a more sophisticated transformation is also desirable. Lemma 5.10 shows that it is possible to perform a type of conditional substitution. A *specialisation* is a conjunction of conditional substitutions which allows us to perform different substitutions in different circumstances. An example of the use of specialisation is given in Chapter 7.

Definition 5.3.
$\Xi = [(e_1, \xi_1), \ldots, (e_n, \xi_n)]$ where each ξ is a substitution and each $e_i \in \mathcal{E}$, is a specialisation. If $g \in$ TL, then $\Xi(g) = \bigwedge_{i=1}^{n}(e_i \Rightarrow \xi_i(g))$.

Lemma 5.10 (Guard lemma).
Suppose $e \in \mathcal{E}$ and $\models \langle\!\!|\, g \Longrightarrow\!\!\!> h \,|\!\!\rangle$.
Then $\models \langle\!\!|\, (e \Rightarrow g) \Longrightarrow\!\!\!> (e \Rightarrow h) \,|\!\!\rangle$.

Proof.
Suppose $\mathrm{t} \preceq$ *Sat*$(\sigma, e \Rightarrow g)$. By the definition of the satisfaction relation, either:

(i) $\mathrm{t} \preceq$ *Sat*$(\sigma, \neg e)$. In this case, by the definition of the satisfaction relation, $\mathrm{t} \preceq$ *Sat*$(\sigma, e \Rightarrow h)$.

(ii) $\mathrm{t} \preceq$ *Sat*(σ, g). In this case, by assumption *Sat*(σ, h). Thus, by definition of the satisfaction relation, $\mathrm{t} \preceq$ *Sat*$(\sigma, e \Rightarrow h)$.

As σ was arbitrary the result follows. □

Theorem 5.11 (Specialisation Theorem).

Let $\Xi = [(e_1, \xi_1), \ldots, (e_n, \xi_n)]$ be a specialisation, and suppose that $\models \langle\!\langle\, g \Longrightarrow\!\!> h \,\rangle\!\rangle$.

Then $\models \langle\!\langle\, \Xi(g) \Longrightarrow\!\!> \Xi(h) \,\rangle\!\rangle$.

Proof.

(1)	For $i = 1, \ldots, n$, $\models \langle\!\langle\, \xi_i(g) \Longrightarrow\!\!> \xi_i(h) \,\rangle\!\rangle$	By Lemma 5.9.
(2)	$\models \langle\!\langle\, (e_i \Rightarrow \xi_i(g)) \Longrightarrow\!\!> (e_i \Rightarrow \xi_i(h)) \,\rangle\!\rangle$	By Lemma 5.10.
(3)	$\models \langle\!\langle\, \bigwedge_{i=1}^{n}(e_i \Rightarrow \xi_i(g)) \Longrightarrow\!\!> \bigwedge_{i=1}^{n}(e_i \Rightarrow \xi_i(h)) \,\rangle\!\rangle$	Repeated application of Theorem 5.4.
(4)	$\models \langle\!\langle\, \Xi(g) \Longrightarrow\!\!> \Xi(h) \,\rangle\!\rangle$	By definition.

□

5.2.8 Until Rule

Theorem 5.12.

Suppose $g_1 \Longrightarrow\!\!> h_1$ and $g_2 \Longrightarrow\!\!> h_2$. Then $g_1 \, \texttt{Until} \, g_2 \Longrightarrow\!\!> h_1 \, \texttt{Until} \, h_2$.

Proof. Let $\sigma = s_0 s_1 s_2 \ldots$ be a trajectory such that $\mathbf{t} \preceq \mathit{Sat}(\sigma, g_1 \, \texttt{Until} \, g_2)$.

(1)	$\exists i \ni$	
	$\mathbf{t} \preceq \bigwedge_{j=0}^{i-1} \mathit{Sat}(\sigma, \texttt{Next}^j g_1) \wedge \mathit{Sat}(\sigma, \texttt{Next}^i g_2)$	By definition of *Sat*.
(2)	$\mathbf{t} \preceq \mathit{Sat}(\sigma, \texttt{Next}^i g_2)$ and	
	$\mathbf{t} \preceq \mathit{Sat}(j\sigma, \texttt{Next}^j g_1),\ j = 0, \ldots, i-1$	Definition of conjunction.
(3)	$\mathbf{t} \preceq \mathit{Sat}(\sigma, \texttt{Next}^i h_2)$ and	□
	$\mathbf{t} \preceq \mathit{Sat}(\sigma, \texttt{Next}^j h_1),\ j = 0, \ldots, i-1$	Assumptions and Theorem 5.3.
(4)	$\mathbf{t} \preceq \bigwedge_{j=0}^{i-1} \mathit{Sat}(\sigma, \texttt{Next}^j h_1) \wedge \mathit{Sat}(\sigma, \texttt{Next}^i h_2)$	Definition of *Sat*.
(5)	$\mathbf{t} \preceq \mathit{Sat}(\sigma, h_1 \, \texttt{Until} \, h_2)$	Definition of *Sat*.
(6)	$g_1 \, \texttt{Until} \, g_2 \Longrightarrow\!\!> h_1 \, \texttt{Until} \, h_2$	Since σ was arbitrary.

Corollary 5.13.

Suppose $g \Longrightarrow\!\!> h$: then $\texttt{Exists} \, g \Longrightarrow\!\!> \texttt{Exists} \, h$ and $\texttt{Global} \, g \Longrightarrow\!\!> \texttt{Global} \, h$.

Proof.

The first result follows directly from the theorem using the definition of the `Exists` operator and the fact that $\mathtt{t} \Longrightarrow\!\!> \mathtt{t}$.

Let σ be an arbitrary trajectory such that $\mathbf{t} \preceq \mathit{Sat}(\sigma, \mathtt{Global}\, g)$.

(1)	$\mathbf{f} \preceq \mathit{Sat}(\sigma, \mathtt{Exists}\,(\neg g))$	Expanding the definition of `Global`.
(2)	$\forall i, \mathbf{f} \preceq \mathit{Sat}(\sigma_{\geq i}, \neg g)$	Definition of *Sat* for `Exists`.
(3)	$\forall i, \mathbf{t} \preceq \mathit{Sat}(\sigma_{\geq i}, g)$	Definition of *Sat* for $\neg$, Lemma 3.1.
(4)	$\forall i, \mathbf{t} \preceq \mathit{Sat}(\sigma_{\geq i}, h)$	$g \Longrightarrow\!\!> h$
(5)	$\forall i, \mathbf{f} \preceq \mathit{Sat}(\sigma_{\geq i}, \neg h)$	Definition of *Sat* for $\neg$.
(6)	$\mathbf{f} \preceq \mathit{Sat}(\sigma, \mathtt{Exists}\,(\neg h))$	Definition of *Sat* for `Exists`.
(7)	$\mathbf{t} \preceq \mathit{Sat}(\sigma, \mathtt{Global}\, g)$	Definition of *Sat* for `Global`.

Which concludes the proof since σ was arbitrary. □

Example 5.8.

Consider again the circuit in Figure 5.1. Using STE, it is easy to prove

$$[B] \Longrightarrow\!\!> \mathtt{Next}\,(\neg[D]).$$

Using Corollary 5.13, we can deduce that $\mathtt{Global}\,[B] \Longrightarrow\!\!> \mathtt{Global}\,(\mathtt{Next}\,(\neg[D]))$.

5.3 Compositional Rules for TL$_n$

For the realisable fragment of TL$_n$, the compositional theory above applies to the $\Longrightarrow\!\!\rhd$ relation as well as the $\Longrightarrow\!\!>$ relation. A key result used is Lemma 4.8. The remainder of the section assumes that we are dealing solely with the realisable fragment of TL$_n$.

Only the statements of theorems are given as the proofs are very similar to or use the proofs in the previous section so the proofs are deferred to Section A.3. Table 5.2 summarises the rules.

Theorem 5.14 (Identity).

For all $g \in \mathrm{TL}_n$, $g \Longrightarrow\!\!\!\triangleright\, g$.

Theorem 5.15 (Time-shift).

Suppose $g \Longrightarrow\!\!\!\triangleright\, h$. Then $\forall t \geq 0$, $\mathtt{Next}^t g \Longrightarrow\!\!\!\triangleright\, \mathtt{Next}^t h$.

Theorem 5.16 (Conjunction).

Suppose $g_1 \Longrightarrow\!\!\!\triangleright\, h_1$ and $g_2 \Longrightarrow\!\!\!\triangleright\, h_2$. Then $g_1 \wedge g_2 \Longrightarrow\!\!\!\triangleright\, h_1 \wedge h_2$.

Theorem 5.17 (Disjunction).

Suppose $g_1 \Longrightarrow\!\!\!\triangleright\, h_1$ and $g_2 \Longrightarrow\!\!\!\triangleright\, h_2$. Then $g_1 \vee g_2 \Longrightarrow\!\!\!\triangleright\, h_1 \vee h_2$.

Theorem 5.18 (Consequence).

Suppose $g \Longrightarrow\!\!\!\triangleright\, h$ and $\Delta^{\mathsf{t}}(g) \sqsubseteq_{\mathcal{P}} \Delta^{\mathsf{t}}(g_1)$ and $\Delta^{\mathsf{t}}(h_1) \sqsubseteq_{\mathcal{P}} \Delta^{\mathsf{t}}(h)$. Then $g_1 \Longrightarrow\!\!\!\triangleright\, h_1$.

Theorem 5.19 (Transitivity).

Suppose $g_1 \Longrightarrow\!\!\!\triangleright\, h_1$ and $g_2 \Longrightarrow\!\!\!\triangleright\, h_2$ and that $\Delta^{\mathsf{t}}(g_2) \sqsubseteq_{\mathcal{P}} \Delta^{\mathsf{t}}(g_1) \amalg \Delta^{\mathsf{t}}(h_1)$.

Then $g_1 \Longrightarrow\!\!\!\triangleright\, h_2$.

Theorem 5.20 (Specialisation).

Let $\Xi = [(e_1, \xi_1), \ldots, (e_n, \xi_n)]$ be specialisation, and suppose that $\models \langle\!\langle\, g \Longrightarrow\!\!\!\triangleright\, h \,\rangle\!\rangle$. Then $\models \langle\!\langle\, \Xi(g) \Longrightarrow\!\!\!\triangleright\, \Xi(h) \,\rangle\!\rangle$.

Theorem 5.21 (Until).

Suppose $g_1 \Longrightarrow\!\!\!\triangleright\, h_1$ and $g_2 \Longrightarrow\!\!\!\triangleright\, h_2$. Then $g_1\,\mathtt{Until}\,g_2 \Longrightarrow\!\!\!\triangleright\, h_1\,\mathtt{Until}\,h_2$.

Other rules, like Corollary 5.13 are possible too. To illustrate this, and because the result is useful, a finite version of Corollary 5.13 follows.

Lemma 5.22.

If $g \Longrightarrow\!\!\!\triangleright\, h$, then for all t, then $\mathtt{Global}\,[(0,t)]\ g \Longrightarrow\!\!\!\triangleright\, \mathtt{Global}\,[(0,t)]\ h$.

Proof. (By induction on t)

(1)	$\texttt{Global}\,[(0,0)]\; g \Longrightarrow\!\!\triangleright \texttt{Global}\,[(0,0)]\; h$	By hypothesis.
(2)	Assume as induction hypothesis: $\texttt{Global}\,[(0,t-1)]\; g \Longrightarrow\!\!\triangleright \texttt{Global}\,[(0,t-1)]\; h$	
(3)	$\texttt{Next}^t g \Longrightarrow\!\!\triangleright \texttt{Next}^t h$	Time shift of hypothesis.
(4)	$\texttt{Global}\,[(0,t)]\; g \Longrightarrow\!\!\triangleright \texttt{Global}\,[(0,t)]\; h$	Conjunction of (2) and (3)

This concludes the induction. □

Corollary 5.23.

If $g \Longrightarrow\!\!\triangleright h$, then for all $s, t, t \geq s$, $\texttt{Global}\,[(s,t)]\; g \Longrightarrow\!\!\triangleright \texttt{Global}\,[(s,t)]\; h$.

Proof.

(1)	$\texttt{Global}\,[(0,t-s)]\; g \Longrightarrow\!\!\triangleright \texttt{Global}\,[(0,t-s)]\; h$	Lemma 5.22
(2)	$\texttt{Global}\,[(s,t)]\; g \Longrightarrow\!\!\triangleright \texttt{Global}\,[(s,t)]\; h$	Time-shift (1).

□

5.4 Practical Considerations

5.4.1 Determining the Ordering Relation: is $\Delta^{\mathsf{t}}(g) \sqsubseteq_{\mathcal{P}} \Delta^{\mathsf{t}}(h)$?

To apply the rules of consequence and transitivity, it is necessary to answer questions such as $\Delta^{\mathsf{t}}(g) \sqsubseteq_{\mathcal{P}} \Delta^{\mathsf{t}}(h)$? One way of testing this is to compute the sets and perform the comparison directly. However, for practical reasons we often wish to avoid the computation of the sets, and to use syntactic and other semantic information to determine the set ordering. Typically formulas like g and h share common sub-formulas and even some structure, which makes the tests explored in this section practical.

Lemma 5.24 is the starting point of these tests, and although very simple, it is important in

Name	**Rule**	**Side condition**
Identity	$\dfrac{}{g \Longrightarrow\!\!\triangleright\, g}$	
Time-shift	$\dfrac{g \Longrightarrow\!\!\triangleright\, h}{\texttt{Next}^t g \Longrightarrow\!\!\triangleright\, \texttt{Next}^t h}$	$t > 0$
Conjunction	$\dfrac{g_1 \Longrightarrow\!\!\triangleright\, h_1 \quad g_2 \Longrightarrow\!\!\triangleright\, h_2}{g_1 \wedge g_2 \Longrightarrow\!\!\triangleright\, h_1 \wedge h_2}$	
Disjunction	$\dfrac{g_1 \Longrightarrow\!\!\triangleright\, h_1 \quad g_2 \Longrightarrow\!\!\triangleright\, h_2}{g_1 \vee g_2 \Longrightarrow\!\!\triangleright\, h_1 \vee h_2}$	
Consequence	$\dfrac{g \Longrightarrow\!\!\triangleright\, h}{g_1 \Longrightarrow\!\!\triangleright\, h_1}$	$\Delta^{\mathsf{t}}(g) \sqsubseteq_{\mathcal{P}} \Delta^{\mathsf{t}}(g_1), \Delta^{\mathsf{t}}(h_1) \sqsubseteq_{\mathcal{P}} \Delta^{\mathsf{t}}(h)$
Transitivity	$\dfrac{g_1 \Longrightarrow\!\!\triangleright\, h_1 \quad g_2 \Longrightarrow\!\!\triangleright\, h_2}{g_1 \Longrightarrow\!\!\triangleright\, h_2}$	$\Delta^{\mathsf{t}}(g_2) \sqsubseteq_{\mathcal{P}} \Delta^{\mathsf{t}}(g_1) \amalg \Delta^{\mathsf{t}}(h_1)$
Specialisation	$\dfrac{\models \langle\!\langle\, g \Longrightarrow\!\!\triangleright\, h \,\rangle\!\rangle}{\models \langle\!\langle\, \Xi(g) \Longrightarrow\!\!\triangleright\, \Xi(h) \,\rangle\!\rangle}$	Ξ a specialisation.
Until	$\dfrac{g_1 \Longrightarrow\!\!\triangleright\, h_1 \qquad g_2 \Longrightarrow\!\!\triangleright\, h_2}{g_1\, \texttt{Until}\, g_2 \Longrightarrow\!\!\triangleright\, h_1\, \texttt{Until}\, h_2}$	

Table 5.2: Summary of TL_n Inference Rules

practice. One effect of this lemma is that if two formulas are syntactically different but semantically equivalent, then they are interchangeable in formulas.

Lemma 5.24.

If g and h are simple then the question whether $\Delta^{\mathsf{t}}(g) \sqsubseteq_{\mathcal{P}} \Delta^{\mathsf{t}}(h)$ is whether for $\forall(s, q) \in D_h$ with $q = \mathsf{t}$ or $q = \top$, $\exists(s', q) \in D_g$ with $s' \sqsubseteq s$.

Proof. This is a restatement of the definition of Δ^{t}.

□

Given this starting point, the ordering relation can be determined by examining the structure of formulas and applying the following lemmas.

Lemma 5.25.

If $g = g_1 \vee g_2$, then $\Delta^{\mathsf{t}}(g) \sqsubseteq_{\mathcal{P}} \Delta^{\mathsf{t}}(g_1)$.

Proof.

(1)	Let $\delta \in \Delta^{\mathsf{t}}(g_1)$	
(2)	$\Delta^{\mathsf{t}}(g) = \Delta^{\mathsf{t}}(g_1) \cup \Delta^{\mathsf{t}}(g_2)$	By definition of Δ^{t}
(3)	$\delta \in \Delta^{\mathsf{t}}(g)$	Set theory
(4)	$\delta \sqsubseteq \delta$	Reflexivity of partial order
(5)	$\Delta^{\mathsf{t}}(g) \sqsubseteq_{\mathcal{P}} \Delta^{\mathsf{t}}(g_1)$	Definition of $\sqsubseteq_{\mathcal{P}}$

□

Corollary 5.26.

For $e \in \mathcal{E}$, $\Delta^{\mathsf{t}}(e \Rightarrow g) \sqsubseteq_{\mathcal{P}} \Delta^{\mathsf{t}}(g)$.

Proof. Straight from the definition of implication. □

Lemma 5.27.

If $g = g_1 \wedge g_2$, then $\Delta^{\mathsf{t}}(g_1) \sqsubseteq_{\mathcal{P}} \Delta^{\mathsf{t}}(g)$.

Proof.

(1)	Let $\delta \in \Delta^{\mathsf{t}}(g)$	
(2)	$\exists \delta^1 \in \Delta^{\mathsf{t}}(g_1), \delta^2 \in \Delta^{\mathsf{t}}(g_2) \ni \delta = \delta^1 \sqcup \delta^2$	Definition of Δ^{t}.
(3)	$\delta^1 \sqsubseteq \delta$	Definition of join.
(4)	$\Delta^{\mathsf{t}}(g_1) \sqsubseteq_{\mathcal{P}} \Delta^{\mathsf{t}}(g)$	Definition of $\sqsubseteq_{\mathcal{P}}$

□

Lemma 5.28.

Suppose $\Delta^{\mathsf{t}}(g) \sqsubseteq_{\mathcal{P}} \Delta^{\mathsf{t}}(h)$: then $\forall i \geq 0, \Delta^{\mathsf{t}}(\mathtt{Next}^i g) \sqsubseteq_{\mathcal{P}} \Delta^{\mathsf{t}}(\mathtt{Next}^i h)$.

Proof. By induction on i. The base case of $i = 0$ follows directly from the assumption.

Suppose $\Delta^{\mathsf{t}}(\mathtt{Next}^i g) \sqsubseteq_{\mathcal{P}} \Delta^{\mathsf{t}}(\mathtt{Next}^i h)$.

Then *shift*$(\Delta^{\mathtt{t}}(\mathtt{Next}^i g)) \sqsubseteq_{\mathcal{P}}$ *shift*$(\Delta^{\mathtt{t}}(\mathtt{Next}^i h))$[1].

Thus $\Delta^{\mathtt{t}}(\mathtt{Next}^{(i+1)} g) \sqsubseteq_{\mathcal{P}} \Delta^{\mathtt{t}}(\mathtt{Next}^{(i+1)} h)$. □

Lemma 5.29.

Suppose $\Delta^{\mathtt{t}}(g_1) \sqsubseteq_{\mathcal{P}} \Delta^{\mathtt{t}}(h_1)$ and $\Delta^{\mathtt{t}}(g_2) \sqsubseteq_{\mathcal{P}} \Delta^{\mathtt{t}}(h_2)$.

Then $\Delta^{\mathtt{t}}(g_1 \,\mathtt{Until}\, g_2) \sqsubseteq_{\mathcal{P}} \Delta^{\mathtt{t}}(h_1 \,\mathtt{Until}\, h_2)$.

Proof.

(1)	$\forall i \geq 0, \Delta^{\mathtt{t}}(\mathtt{Next}^i g_1) \sqsubseteq_{\mathcal{P}} \Delta^{\mathtt{t}}(\mathtt{Next}^i h_1)$	By Lemma 5.28
(2)	$\forall i \geq 0, \Delta^{\mathtt{t}}(\mathtt{Next}^i g_2) \sqsubseteq_{\mathcal{P}} \Delta^{\mathtt{t}}(\mathtt{Next}^i h_2)$	By Lemma 5.28
(3)	$\Delta^{\mathtt{t}}(g_1 \,\mathtt{Until}\, g_2) = \cup_{i=0}^{\infty}(\Delta^{\mathtt{t}}(\mathtt{Next}^0 g_1) \amalg \ldots \amalg \Delta^{\mathtt{t}}(\mathtt{Next}^{(i-1)} g_1) \amalg \Delta^{\mathtt{t}}(\mathtt{Next}^i g_2))$	By definition
(4)	$\sqsubseteq_{\mathcal{P}} \cup_{i=0}^{\infty} (\Delta^{\mathtt{t}}(\mathtt{Next}^0 h_1) \amalg \ldots \amalg \Delta^{\mathtt{t}}(\mathtt{Next}^{(i-1)} h_1) \amalg \Delta^{\mathtt{t}}(\mathtt{Next}^i h_2))$	From (1) and (2)
(5)	$= \Delta^{\mathtt{t}}(h_1 \,\mathtt{Until}\, h_2)$	By definition

□

Lemma 5.30.

For all i, $\Delta^{\mathtt{t}}(\mathtt{Next}^i g) \sqsubseteq_{\mathcal{P}} \Delta^{\mathtt{t}}(\mathtt{Global}\, g)$.

Proof. Let $\delta \in \Delta^{\mathtt{t}}(\mathtt{Global}\, g)$.

(1)	$\mathtt{t} \preceq$ *Sat*$(\delta, \mathtt{Global}\, g)$	Lemma A.6
(2)	$= \neg$*Sat*$(\delta, \mathtt{Exists} \neg g)$	Definition of `Global`
(3)	$= \neg$*Sat*$(\delta, \mathtt{t}\,\mathtt{Until} \neg g)$	Definition of `Exists`
(4)	$= \neg \vee_{i=0}^{\infty}$ (*Sat*$(\delta_{\geq i}, \neg \mathtt{Next}^i g))$	Definition of satisfaction
(5)	$= \neg \vee_{i=0}^{\infty} \neg($*Sat*$(\delta_{\geq i}, \mathtt{Next}^i g))$	Definition of satisfaction
(6)	$= \wedge_{i=0}^{\infty}$*Sat*$(\delta_{\geq i}, \mathtt{Next}^i g)$	De Morgan's law
(7)	$\forall i, \mathtt{t} \preceq$ *Sat*$(\delta_{\geq i}, \mathtt{Next}^i g)$	Definition of conjunction

[1] Recall that shift$(s_0 s_1 s_2 \ldots) = \mathsf{X} s_0 s_1 s_2 \ldots$

(8) $\forall i, \exists \delta' \in \Delta^{\mathbf{t}}(\texttt{Next}^i g) \ni \delta' \sqsubseteq \delta$ Lemma 4.3

(9) $\forall i, \Delta^{\mathbf{t}}(\texttt{Next}^i g) \sqsubseteq_{\mathcal{P}} \Delta^{\mathbf{t}}(\texttt{Global}\, g)$ Definition of $\sqsubseteq_{\mathcal{P}}$

□

Similar rules can be developed for $\Delta^{\mathbf{f}}$; the two sets of rules are tied together by the definition of the satisfaction of negation. These tests seem very simple and obvious, but in practice they allow the development of efficient algorithms to test whether $\Delta^{\mathbf{t}}(g) \sqsubseteq_{\mathcal{P}} \Delta^{\mathbf{t}}(h)$.

5.4.2 Restriction to TL_n

The restrictions on the logic TL_n make it much easier reason about. Recall that the basis of the logic is the set of predicates G_n. In practice, many TL_n formulas are of the form

$$\bigwedge_{i=0}^{r} \texttt{Next}^i (\bigwedge_{j=0}^{s_i} [n_{i,j}] = e_{i,j})$$

where the $e_{i,j} \in \mathcal{E}$. Given

$$\begin{aligned} g &= \textstyle\bigwedge_{i=0}^{r'} \texttt{Next}^i (\bigwedge_{j=0}^{s'_i} [n'_{i,j}] = e'_{i,j}) \\ h &= \textstyle\bigwedge_{k=0}^{r} \texttt{Next}^k (\bigwedge_{l=0}^{s_k} [n_{k,l}] = e_{k,l}), \end{aligned}$$

from Lemma 5.27 it follows that to determine whether $\Delta^{\mathbf{t}}(g) \sqsubseteq_{\mathcal{P}} \Delta^{\mathbf{t}}(h)$, we need to check whether

$$\forall i, j; \exists l \ni n'_{i,j} = n_{i,l} \wedge e'_{i,j} = e_{i,l}.$$

This can largely be done syntactically. Depending on the representation used, testing whether $e_{i,l} = e'_{i,j}$ may either be done syntactically or using other semantic information. Particularly when the level of abstraction is raised, it is often the case that other semantic information must be used.

Of course, there are important cases where formulas are not of this form, and we need to have other ways of reasoning about them. A more general and typical case is verifying an assertion of the form $\langle\!\langle\, g \Longrightarrow h \,\rangle\!\rangle$, where g is an arbitrary TL_n formulas and $h = \texttt{Next}^j (\bigwedge_{i=0}^{k} ([n_i] = e_i))$.

Definition 5.4.

Strict dependence: Informally, $g \in \mathrm{TL}_n$ is strictly dependent on the state components $R = \{r_1, \ldots, r_l\}$ at time t if g being true at time t implies that the components $r_1, \ldots, r_l$ have defined values, and g is not dependent on any other state components. The formal condition for strict dependence is: g is strictly dependent on the state components R if

$$\forall \delta \in \Delta^{\mathtt{t}}(g) : \; \forall r \in R, \; \mathsf{U} \sqsubset \delta_t[r]; \; \forall s \notin R, \; \delta_t[s] = \mathsf{U}.$$

In practice, strict dependence can often be checked syntactically. For example $[B] = e$ where $e \in \mathcal{E}$ is strictly dependent on B. This comes from the property of exclusive-or — if $a \oplus b \in \mathcal{B}$, where exclusive-or, $a \oplus b$, is defined as $a \wedge (\neg b) \vee (\neg a) \wedge b$ — then $a, b \in \mathcal{B}$). Moreover, strict dependence can be checked relatively efficiently (as will be seen later).

Theorem 5.31 (Generalised Transitivity).

Let A_1 be a trajectory formula such that $\tau = \tau^{A_1} \in \mathcal{R}_{\mathcal{T}}(t)$, and let $h_1 = \mathtt{Next}^t h$ be a TL_n formula strictly dependent on state components $\{r_1, \ldots, r_l\}$ at time t where h contains no temporal operators. Let $A_2 = \mathtt{Next}^t(\bigwedge_{j=1}^{l}[r_j] = v_j)$ where the $v_j \in \mathcal{V}$.

Suppose $\models \langle\!|\, A_1 \Longrightarrow\!\!\!\!\triangleright\, h_1 \,|\!\rangle$ and $\models \langle\!|\, A_2 \Longrightarrow\!\!\!\!\triangleright\, h_2 \,|\!\rangle$. Then,

(1) There is a substitution ξ such that $\models \langle\!|\, A_1 \ggg \xi(h_2) \,|\!\rangle$; and

(2) $h(\tau_t^{A_1}) = \mathtt{t}$.

Proof.

(1) For $i = 1, \ldots, l, \; \exists e_i \in \mathcal{E} \ni e_i = \tau^{A_1}[r_i]$ $\quad\models \langle\!|\, A_1 \Longrightarrow\!\!\!\!\triangleright\, h_1 \,|\!\rangle$, strict dependence of h_1.

(2) Let $C = \mathtt{Next}^t \bigwedge_{j=1}^{l}[r_i] = e_i$

(3) $\models \langle\!|\, A_1 \Longrightarrow\!\!\!\!\triangleright\, C \,|\!\rangle$ — By construction of C.

(4) $\models \langle\!|\, A_1 \Longrightarrow\!\!\!\!\triangleright\, h_1 \wedge C \,|\!\rangle$ — Conjunction.

(5) For $v \in \{v_1, \ldots, v_l\}$ let $\xi(v_j) = e_j$

For $v \notin \{v_1, \ldots, v_l\}$ let $\xi(v) = v$

(6) $\models \langle\!|\, A_2 \Longrightarrow\!\!\!\!\triangleright\, h_2 \,|\!\rangle$ — Given.

(7) $\models \langle\!\langle\, \xi(A_2) \Longrightarrow\!\triangleright\, \xi(h_2) \,\rangle\!\rangle$ Substitution (Lemma A.18).

(8) $\models \langle\!\langle\, h_1 \wedge \xi(A_2) \Longrightarrow\!\triangleright\, \xi(h_2) \,\rangle\!\rangle$ Rule of consequence.

(9) $C = \xi(A_2)$ By construction.

(10) $\models \langle\!\langle\, A_1 \Longrightarrow\!\triangleright\, \xi(h_2) \,\rangle\!\rangle$ From (4), (8) by transitivity.

This concludes the proof of (1)

(11) $h(\tau_t^{A_1}) = \mathbf{t}$ $\models \langle\!\langle\, A_1 \Longrightarrow\!\triangleright\, \mathtt{Next}^t h \,\rangle\!\rangle$ and $\tau^{A_1} \in \mathcal{R}_{\mathcal{T}}(t)$

This concludes the proof of (2)

□

Although the proof of this theorem is relatively complex, the theorem itself is not, and very importantly many important side-conditions can be checked automatically. The seeming complexity of the theorem comes from having to relate A_1 to A_2. But, this turns out to be the virtue of the theorem. The difficulty with trying to use transitivity between two results $\models \langle\!\langle\, A_1 \Longrightarrow\!\triangleright\, h_1 \,\rangle\!\rangle$ and $\models \langle\!\langle\, A_2 \Longrightarrow\!\triangleright\, h_2 \,\rangle\!\rangle$ is to find the appropriate specialisation for the latter result. This theorem provides a method for doing this: the first part of the theorem says that a specialisation exists, and the second part helps find it. The example below illustrates the use of the theorem.

Example 5.9.

Figure 5.2 shows two cascaded carry-save adders (CSAs). There are four inputs to the entire circuit, and two outputs. Three of the inputs get fed into one of the CSAs; the other CSA gets the fourth of the inputs and the two outputs of the first CSA. Assuming each CSA takes one unit of time to compute its results, if four values get entered at J, K, L and M, two units of time later, the sum of these four values will be the same as the sum on nodes P and Q.

Let

$$\begin{aligned}
A_1 &= ([J] = j) \wedge ([K] = k) \wedge ([L] = l) \wedge (\mathtt{Next}\ [M] = m)\\
h_1 &= \mathtt{Next}\,([N] + [O] = j + k + l \wedge [M] = m)\\
A_2 &= \mathtt{Next}\,([M] = m) \wedge ([N] = n) \wedge ([O] = o)\\
h_2 &= \mathtt{Next}^2([P] + [Q] = m + n + o)
\end{aligned}$$

Then

$\models \langle\!\langle A_1 \Longrightarrow h_1 \rangle\!\rangle$

$\models \langle\!\langle A_2 \Longrightarrow h_2 \rangle\!\rangle$.

These two results can be proved using trajectory evaluation. The process of performing trajectory evaluation also checks that $\tau^{A_1} \in \mathcal{R}_{\mathcal{T}}(d(h_1))$. A_2 and h_1 are of the correct form for Theorem 5.31. Furthermore the strict dependence of h_1 on components M, N and O can be checked syntactically. By the theorem we have that there is a specialisation ξ such that

$$\models \langle\!\langle A_1 \Longrightarrow \texttt{Next}^2([P] + [Q] = \xi(m + n + o)) \rangle\!\rangle$$

and

$$([N] + [O] = j + k + l \wedge [M] = m)(\tau_2^{A_1}) \ = \mathbf{t}$$

which means that

$$(\tau_2^{A_1}[N] + \tau_2^{A_1}[O] = j + k + l) = \mathbf{t}.$$

But, by the structure of h_1 we know that

$$\tau_2^{A_1}[N] = \xi(n) \text{ and } \tau_2^{A_1}[O] = \xi(o) \text{ and } \tau_2^{A_1}[M] = \xi(m)$$

and so, as by the properties of substitution $\xi(x + y) = \xi(x) + \xi(y)$,

$$(\xi(n + o) = j + k + l \ \wedge \ \xi(m) = m) = \mathbf{t}.$$

This result has given us sufficient information about ξ. Thus,

$$\models \langle\!\langle A_1 \Longrightarrow \texttt{Next}^2([P] + [Q] = j + k + l + m) \rangle\!\rangle.$$

5.5 Summary

This chapter presented a compositional theory for TL; this theory is very important in overcoming the computational bottlenecks of automatic model checking. The focus of the compositional theory is property composition, which is particularly suitable for STE-based model checking. The general compositional theory for TL was presented, followed by additional rules for TL$_n$.

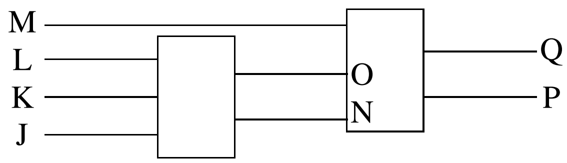


Figure 5.2: Two Cascaded Carry-Save Adders

Section 5.4 discussed some issues that are important in the practical implementation of the theory.

Chapter 6 shows how the compositional theory can be implemented in a practical tool.

Chapter 6

Developing a Practical Tool

This chapter discusses how to put the ideas presented in the previous chapters into practice. A number of prototype verification systems using these ideas have been implemented to test how effectively the verification methodology can be used. Although these prototypes have been used to verify substantial circuits, they are *prototypes*, and the purpose of the chapter is to show how a practical verification system using TL can be developed, rather than to describe a particular system.

Section 6.1 discusses the Voss system, developed to support the restricted form of trajectory evaluation. Voss is important because the algorithms that it implements form the core of the prototype verification systems. This section also discusses the important issues of how boolean expressions, sets of interpretations, and sets of states are represented. Section 6.2 examines higher-level representational issues, in particular efficient ways of representing TL formulas so that they can be efficiently stored and manipulated. It is important that appropriate representational schemes be used since different methods are appropriate at different stages. By converting (automatically) from one scheme to another, the strengths of the different methods can be combined. Section 6.3 shows how trajectory evaluation and theorem proving can be combined into one, integrated system. The motivation for this is to provide the user with a tool which provides the appropriate proof methods at the right level of abstraction — model checking at the low level, theorem proving at the high level where human insight is most productively used. The theorem prover component is the implementation of the compositional theory, which is critical for the practicality of the approach. One of the key issues here is how to provide as much assistance to the human verifier as possible. The final section, Section 6.4 extends existing trajectory evaluation algorithms so they can be used to support a richer logic.

6.1 The Voss System

In order to verify realistic systems, any theory of verification needs a good tool to support it. Seger developed the Voss system [115] as a formal verification system (primarily for hardware verification) that uses symbolic trajectory evaluation extensively.

There are two core parts of Voss. The user interface to Voss is through the functional language FL, a lazy, strongly typed language, which can be considered a dialect of ML [107]. One of the key features of FL is that BDDs are built into the language, as boolean objects are by default represented by BDDs. Since BDDs are an efficient method of representing boolean functions, data structures based on BDDs, such as bit vectors representing integers, are conveniently and efficiently manipulated. Of course, as previously discussed, the limitations of BDDs mean that there are limitations on what can be represented and manipulated efficiently; how these limitations are overcome is an important topic of this chapter.

The second component of Voss is a symbolic simulation engine with comprehensive delay modelling capabilities. This simulation engine, which is invoked by an FL command, provides the underlying trajectory evaluation mechanism for trajectory formulas.

Trajectory formulas are converted into an internal representations (the 'quintuple lists') and passed to the simulation engine; these quintuple lists essentially are representations of the defining sequences of the antecedent and consequent formulas comprising the assertions. The antecedent formula is used to initialise the circuit model, and the simulation engine then computes the defining trajectory for the antecedent. As this evaluation proceeds, Voss will flag any *antecedent failures*, *viz.* a Z appearing in the antecedent, and compares the defining trajectory with the defining sequence of the consequent. Two types of errors are reported if the comparison fails: a weak consequent failure occurs if Us appearing in the defining trajectory are the cause of the failure[1]; a strong consequent failure is reported if the defining trajectory of the antecedent is not commensurable with the defining sequence of the consequent.[2] Using the terminology of TL and $\mathcal{Q}$, a weak failure corresponds to the satisfaction relation returning $\perp$, a strong failure corresponds to a $\mathtt{f}$.

Circuit models can be described in a number of formats. Interacting through FL, the user sees models as abstract data types (ADTs) of type fsm. FL provides a library (called the EXE library) which allows the user to construct gate level descriptions of circuits. Once a circuit is constructed as an EXE object, the model can be converted into an fsm model. There are also tools provided for converting other formats (both gate level and switch level models) into fsm objects; among others, VHDL and SILOS circuit descriptions can be accepted.

Representing Sets of Interpretations

Since STE-based verification computes the sets of interpretations of variables for which a given relation holds, efficient methods for representing and manipulating these sets is important. Voss represents a set of interpretations by a boolean expression (i.e., by a BDD). If φ is a boolean expression, then φ represents the set $\{\phi \in \Phi : \phi(\varphi) = \mathtt{t}\}$. This representation relies on the power of BDDs, so although usually a good method, it breaks down sometimes. One advantage of this representation is that set manipulation can easily be accomplished as boolean operations. If φ_1 represents the set of mappings Φ_1 and φ_2 the set of mappings Φ_2, then $\varphi_1 \vee \varphi_2$ is the representation of $\Phi_1 \cup \Phi_2$, $\varphi_1 \wedge \varphi_2$ the representation of $\Phi_1 \cap \Phi_2$, $\neg\varphi_1$ is the representation of $\Phi \backslash \Phi_1$, and set containment can be tested by computing for logical implication.

[1] This happens if the defining trajectory of the antecedent is less than the defining sequence of the consequent; the verification might succeed with a stronger antecedent.

[2] A stronger antecedent will only make things worse.

A more general point about representing interpretations also needs to be made. Suppose that $b \in \mathcal{E}$ is a boolean expression (and so represented as a BDD). Let $v_1, \ldots, v_m$ be the variables appearing in b.[3] To ask whether there is an interpretation ϕ such that $\phi(b) = \mathtt{t}$ is the same as asking whether $\exists v_1, \ldots, v_m.b$ (are there boolean values that can replace the variables so that the expression evaluates to true?). Since existential quantification is a standard BDD operation, this can be computed in FL through the construction and manipulation of BDDs.

Representing Sets of States

Voss manipulates and analyses circuit models, *viz.* models where the state space is naturally represented by $\mathcal{C}^n$ for some n. A state in $\mathcal{C}^n$ is thus a vector $\langle c_1, \ldots, c_n \rangle$, where each $c_i \in \mathcal{C}$. Voss uses a variant of the dual-rail encoding system discussed in Section 3.2 for representing elements of $\mathcal{C}^n$, which means that each state in Voss is represented by a vector $\langle (a_1, b_1), \ldots, (a_n, b_n) \rangle$, where the $a_i, b_i \in \mathcal{B}$.

The use of boolean variables allows one symbolic state to represent a large number of states. The vector

$$s = \langle (a_1, b_1), \ldots, (a_n, b_n) \rangle$$

(where the $a_i, b_i \in \mathcal{E}$) represents the set of states $\{\phi(s) : \phi \in \Phi\}$ where

$$\phi(s) = \langle (\phi(a_1), \phi(b_1)), \ldots, (\phi(a_n), \phi(b_n)) \rangle.$$

Note that the a_i and b_i need not contain any variables. This idea can be extended so that sets of sequences can be represented by symbolic sequences.

This type of representation is known as a *parametric representation*. The alternative representation is the *characteristic representation*. (For discussion of these representations, see [43,

[3] This is a simple syntactic test; since quantification does not exist in $\mathcal{E}$, we do not have to ask whether variables are free or bound.

87].) $\chi : \mathcal{C}^n \to \mathcal{B}$ represents the set

$$\{s : \chi(s) = \mathbf{t}\}.$$

Note the similarity to the way in which interpretations are represented. This indicates that χ can be represented as a BDD — the mechanics of this are presented now. Let $s = \langle s_1, \ldots, s_n \rangle$ be a symbolic state representing the set of states S, and let $v_1, \ldots, v_m$ be the variables appearing in s. Let $r = \langle r_1, \ldots, r_n \rangle$, where each r_i is a pair of boolean variables $(r_{i,1}, r_{i,2})$ not appearing in $\{v_1, \ldots, v_m\}$.

$$\begin{aligned}
\chi(r) = \mathbf{t} \quad & \Longleftrightarrow \exists \phi \in \Phi \ni r = \phi(s) \\
& \Longleftrightarrow \exists \phi \in \Phi \ni \wedge_{i=1}^{n}(r_i = \phi(s_i)) \\
& \Longleftrightarrow \exists v_1, \ldots, v_n \ni \wedge_{i=1}^{n}(r_i = s_i)
\end{aligned}$$

Thus $\chi(r)$ can be represented a boolean expression (BDD) containing the $r_{i,j}$ variables only. To determine whether a particular state is in the set, the r_i are instantiated in $\chi(r)$; the value obtained is $\mathbf{t}$ if and only if the state is the set.

The advantage of the characteristic representation is that it is convenient to perform union and intersection operations on sets of states. Moreover, as each set is represented by one BDD, set representations are canonical, which is extremely useful. However, this monolithic representation of sets of states can be very expensive.

The primary advantage of the parametric representation is that it is very compact. n independent BDDs represent a set of states, which increases the size of the state space that can be manipulated. Moreover, this representation is particularly suitable for the symbolic simulation of the state space, and for the computation of defining sequences. It is the representation method on which STE is based.

6.2 Data Representation

Although exploiting the power of BDDs to implement the underlying trajectory evaluation efficiently is essential, there needs to be complementary ways of representing and manipulating data.

One way of doing this is to represent TL formulas and associated data structures symbolically. This is best explained by the following example. Suppose the circuit being verified is an integer adder. Formally, the circuit model represents the integers as bit vectors of appropriate size, and addition of integers is formally represented as bit vector manipulation. The TL formulas used to specify correctness will formally describe the behaviour of the circuit at the bit level.

In our prototype tools, integer types like this are represented and manipulated in the following way.

- An abstract data type representing integers is declared. See Figure 6.1 which gives an example declaration: integers are constants, variables, or the addition, subtraction, multiplication, division, or exponentiation of two integers; integer predicates are the comparison of two integers.
- A routine which converts integer objects into bit vectors, and integer predicates into an equivalent predicate over bit vectors is written. For convenience, this routine is referred to as bv. Typically, the bit vectors are finite and can be represented in standard ways (e.g. twos complement). However, it is also possible to have representations of infinite bit vectors (the lazy semantics of FL is useful here).
- A set of bit vector operations giving the formal semantics of the objects is implemented. Addition, for example, is modelled by operations on two bit vectors.
- A set of ADT operations corresponding to the vector operations is implemented. This means that the FL program can manipulate integer-related objects without converting them

into the associated bit vectors.

```
lettype N =  // Natural number expressions
        Nvar string
      | Nconst int
      | Nadd N N
      | Nsub N N
      | Nmult N N
      | Ndiv N N
      | Npow N N
```

Figure 6.1: An FL Data Type Representing Integers

Although the formal semantics of integer objects is given by bit vectors and the operations on bit vectors, the higher-level representation is useful for two reasons. First, it has the effect of raising the level of abstraction, which makes the verification task for the user easier since it enables the user to deal with higher-level, composite objects. Second, and more importantly, it has significant performance advantages; BDDs can be used where appropriate and other methods where BDDs fail.

This situation is depicted in Figure 6.2. The FL program stores the object d; by applying the conversion routine, bv, the bit vector which represents d can be computed. Applying the operation f_{adt} to d is the same as applying the operation f_{bdd} to d_{bdd}, which is illustrated by the commutative diagram in Figure 6.2.

$$\begin{tikzcd} d \arrow[r, leftrightarrow, "f_{adt}"] \arrow[d, "bv"] & d' \arrow[d, "bv"] \\ d_{bdd} \arrow[r, leftrightarrow, "f_{bdd}"] & d'_{bdd} \end{tikzcd}$$

Figure 6.2: Data Representation

Thus, even if d or d' cannot be represented efficiently as BDDs, there is an effective way of representing and manipulating them through the abstract data type representation. As will be seen, using this method of representation is an effective way of going beyond the limits of BDDs.

It is critical that both the conversion routine bv and the ADT operations are implemented correctly so that the diagram in Figure 6.2 *is* commutative. In the HOL-Voss system correctness was formally proved [90]. Although I did not go through this exercise in the prototype implementations, this is a critical step in the production of a tool. However, one should note that there may be a trade-off between degree of rigour and performance. For example, in an interesting paper showing how BDDs can be implemented as a HOL derived rule [75], Harrison reports that a HOL implementation of BDDs as being fifty times slower than a Standard ML implementation. Although this work is cited as being 'superior to any existing tautology-checkers implemented in HOL', Harrison points out that other approaches to ensuring correctness can be adopted.

The ADT routines that implement the operations on the data objects constitute *domain knowledge*, representing the verification system's higher level semantic knowledge of what bit-level operations mean. There are different ways in which domain knowledge can be provided. One method is to have a canonical representation for data objects, or to have a set of decision procedures for the type (for example, to tell whether two syntactically different objects have the same semantics — whether if they are both converted into BDD structures, the structures will be the same). There is a limit to how far this can go; for example, with the integer representations used, no canonical forms exist, and decision procedures have limitations and can be expensive. However, as will be seen this can be effective and, since it is automatically implemented, user-friendly, reducing the load of the human verifier.

Another method — which can be implemented as an alternative or as a complement to the decision procedure method — is to provide an interface to an external source of knowledge. One likely such source is a trusted theorem prover such as HOL, allowing the verifier to prove results in HOL, and then to import these results into the verification system. Although approach is very flexible and very powerful, it increases the level of expertise needed by the verifier considerably.

6.3 Combining STE and Theorem Proving

The practical importance of the compositional theory presented in Chapter 5 is that it provides a powerful way of combining STE and theorem proving. The inference rules of the compositional theory are implemented as proof rules of a theorem prover. The combination of theorem proving and STE creates a tool which provides the appropriate proof mechanism at the appropriate level. For a human to reason at the individual gate level, while conceptually simple and straightforward, is often too onerous and tedious. A single trajectory evaluation can often deal with the behaviour of hundreds or thousands of gates, depending on the application. The theorem prover allows the human verifier to use insight into the problem to combine lower-level results using the compositional theory. By using the representation method discussed above, and the compositional theory, the computational bottle-neck of automatic model checking algorithms can be widened considerably.

The prototype verification systems built implement proof systems based on STE and the compositional theory for STE. The object of verification is to prove properties of the form $\models \langle\!\langle\, g \Longrightarrow\!\!\triangleright\, h \,\rangle\!\rangle$. The proof system does this by using STE as a primitive rule for proving assertions; the compositional theory is implemented as set of proof rules that can be used to infer other results.

From a practical point of view, the Voss system provides a good basis for this. The user interacts with the proof system using FL. By using the appropriate FL library routines, trajectory evaluation and the compositional theory can be used. There are different ways in which this could be done and packaged.

For example, in the first prototype tool, the verification library consisted of the following rules, implemented as FL functions, each function either invoking trajectory evaluation or one of the compositional inference rules.

- VOSS: This performs trajectory evaluation.
- IDENTITY implements the identity rule.
- CONJUNCT implements conjunction.
- SHIFT allows assertions to be time-shifted.
- PRESTRONG implements part of the rule of consequence, allowing the antecedent of an assertion to be strengthened.

  POSTWEAK implements the other part of the rule of consequence, allowing the consequent of an assertion to be weakened. Both of these rules use domain knowledge to check the correctness of the use of the rule.
- TRANS takes two assertions, checks whether transitivity can be applied between the first and second (i.e., the correct relationship holds between the two assertions), and if it can be, applies the rule of transitivity.
- SPECIAL allows the user to specialise an assertion.
- SPTRANS takes two assertions, T_1 and T_2, and attempts to find a specialisation Ξ such that transitivity can be applied between T_1 and $\Xi(T_2)$. The heuristic used to find the specialisation (discussed later) does not compromise the safety of the verification since if it fails and no specialisation is found, no result is deduced. Moreover, if a putative specialisation is found, the correctness of the specialisation is checked by testing whether the conditions for transitivity to apply do hold once the specialisation is applied.
- AUTOTIME takes two assertions, T_1 and T_2, and attempts to find an appropriate time-shift t for one of the assertions so that transitivity can be applied. Recall that the time-shift rule

only applies if $t \geq 0$. If $t < 0$ is found, then the verification system shifts T_1 forward by $-t$ time steps and attempts to apply transitivity between the shifted T_1 and T_2; if $t \geq 0$, then T_2 is shifted forward by t time units, and then the verification system attempts to apply transitivity between T_1 and the shifted T_2.

- ALIGNSUB combines the ideas of the above two rules. Given two assertions it attempts to find a time-shift and specialisation so that when both are applied, transitivity can be used.
- PRETEND allows a desired result to be assumed without proof. When deciding whether an overall proof structure is correct, it may be useful to assume some of the sub-results and then see whether combining the sub-results will obtain the overall goal, before putting effort into proving the sub-results. Furthermore, in a long proof built up over some period of time it may be desirable at different stages in proof development to replace some calls of VOSS with PRETEND. Having proved a property of the circuit using STE it may take too much time in proof development to always perform all STE verifications when the proof script is run. Although at the end, the entire verification script should be run completely, it is not necessary to always perform all trajectory evaluations in proof development.

An important part of implementing this verification methodology was to integrate the trajectory evaluation and theorem proving aspects into one tool. Not only does this make the methodology easier for the user (since the quirks of only one system have to be learned and only one conceptual framework and set of notations have to be learned), the practical soundness of the system is maintained (the user does not have to translate from one formalism to another).

Moreover, using FL as the interface is very beneficial. Although this requires the user to be familiar with FL, once learned it provides a flexible and powerful proof tool. Using the basic verification library provided by the tool, the user can package the routines in different ways.

The proof is written as an FL program that invokes the proof rules appropriately. This allows the proof to be built up in parts and combined. The use of a fully programmable proof script language — FL — removes much drudgery and tedium.

A critical factor in trajectory evaluation, affecting both the performance and the automatic nature of STE, is the choice of the BDD variable orderings used in the trajectory evaluation. A poor choice of variable ordering can make trajectory evaluation impossible or slow [44]. Although the use of dynamic variable ordering techniques (one of which has been implemented in Voss) ameliorates the situation, the compositional method means that dynamic variable ordering is not a panacea. In many cases, there is simply not *one* variable ordering that can be used. The strength of the compositional theory is that it allows different variable orderings to be used for different trajectory evaluations. If different variable orderings must be used for each of many trajectory evaluations (for some proofs hundreds of trajectory evaluations could be done), using dynamic variable ordering alone might significantly degrade performance.

On other hand, in many applications, good heuristics exist for choosing variable ordering automatically based on the structure of the TL formulas. One of the advantages of representing data at a high level (an integer ADT) is that knowledge of the type and operations on the type can be used to determine appropriate variable orderings. A useful technique is to provide as part of the FL library implementing a particular ADT, a function which takes an expression of the type and produces a 'good' variable ordering.

This particular example illustrates the advantages of incorporating heuristics into a system to aid the user. Other examples of heuristics which proved to be useful are the heuristics which takes two assertions and try to find appropriate specialisations and time-shifts so that transitivity can be used between the two assertions. The algorithms that implement these heuristics are straightforward. Although there are a number of possible heuristics and algorithms that could be used, experience showed that simple implementations are quite flexible.

Finding a time-shift: This algorithm takes the consequent of one assertion and the antecedent of another and determines whether if one of the formulas is time-shifted, the two formulas are related to each other (in that their defining sequences are ordered by the information ordering). String matching is the core of the algorithm, and although the extremely large 'alphabet' restricts the sophistication of the string matching algorithms that can be applied, in practice the simple structure of the formulas means that simple string matching algorithms are quite adequate.

Finding a specialisation: This heuristic performs a restricted unification between two formulas to discover whether if one of the formulas is specialised, it is implied by the other (in that the defining sequence of one is ordered with respect to that of the specialised formula). A similar approach is used in implementing Generalised Transitivity (Theorem 5.31). Since semantic information must be used as well as syntactic (two syntactically different expressions may be semantically equivalent), the effectiveness of the algorithm is limited by the power of the domain knowledge incorporated into the tool. However, the simple structure of most antecedents means that a simple heuristic works well.

It is also possible to incorporate both heuristics into one heuristic so that candidate time-shifts and specialisations are sought at the same time. To implement this completely is much more difficult since there may be a number of different time-shift and specialisation combinations that can be applied. It can also be computationally more expensive, since for each possible time-shift it may be necessary to use different domain knowledge. However, in practice, since formulas tend not be very large, this can be useful and practical. Here, the representation of data at the ADT level is very important practically since high-level information can be used to find whether time-shifts and specialisations will be appropriate; if a lower level representation were used, much more work would need to be done.

In all cases, once a transformation is found, it is automatically applied and *checked*; this also shows that heuristics can be incorporated without compromising the soundness of the proof system. The core inference rules are always used for deducing results; the heuristics provided by the user or the verification system as FL functions are there to automate the proof (at least partially) in determining how the inference rules are to be used, and at no stage is the safety of the result compromised. Moreover, if a transformation cannot be found, suitable error messages can be printed indicating why such a transformation could not be found, helping the user to determine whether the attempted use of the rule was wrong (e.g. because the desired result is false), whether more information is needed (e.g. perhaps the rule of consequence must be applied to one of the assertions first), or more domain knowledge must be provided by the user.

6.4 Extending Trajectory Evaluation Algorithms

The core of the practical tool proposed here is the ability to perform trajectory evaluation to check assertions of the form $\models \langle\!|\, g \Longrightarrow\!\triangleright\, h \,|\!\rangle$, where g and h are TL formulas (actually TL_n formulas since we are dealing with circuit models). The basis of these algorithms is the trajectory evaluation facility of Voss, which can compute results of the form $\models \langle\!|\, A \Longrightarrow\!\!\!> C \,|\!\rangle$, where A and C are trajectory formulas.

There is a trade-off between how efficiently trajectory evaluation can be done, and the class of assertions that can be checked. This section first describes and justifies the restrictions placed on assertions, and then outlines three possible algorithms that can be used to extend Voss's STE facility. (The advantages and disadvantages of these algorithms are discussed in Section 7.6 after the presentation of experimental evidence.)

6.4.1 Restrictions

What are the problems in determining whether $\models \langle\!|\, g \Longrightarrow\!\triangleright\, h \,|\!\rangle$?

First, STE computes the $\Longrightarrow\!\!\!>$ relation, rather than the $\Longrightarrow\!\!\!\rhd$ relation. However, as shown in Section 4.4.3, if only the realisable fragment of TL_n is used, there is an efficient way to deduce the $\Longrightarrow\!\!\!\rhd$ relation from the $\Longrightarrow\!\!\!>$ relation. The limitation to the realisable fragment means that users cannot explicitly check whether a component of the circuit takes on an overconstrained value. But, the nature of the circuit model means that this is checked for implicitly in any trajectory evaluation. The underlying trajectory evaluation engine can easily check for antecedent failures by testing whether a Z appears in the defining trajectory of the antecedent. Thus, I argue that this limitation is not a severe restriction, and worth the price.

Second, allowing a general TL_n formula in the antecedent can be very costly since it may require numerous trajectory evaluations to be done. Recall from Chapter 4 that computing the defining sequence sets of a disjunction is done by taking the union of the defining sequence sets of the disjuncts. Thus, the cardinality of the defining sequence sets is proportional to the number of disjuncts. At first sight, it may seem that in practice that the structure of formulas is such that this will not be a real problem. For circuit verification, how many formulas have more than a dozen disjuncts (a number of sequences that could probably easily be dealt with)? However, this is misleading since while disjuncts may not appear explicitly in a formula, they may actually be there, particularly when dealing with non-boolean data types. For example, a predicate on an integer data type such as $[I] + [J] = k + l + m$ can translate into a very large number of disjuncts, even for moderate sized bit-widths.

Thus, for performance reasons, one restriction placed on formulas is that trajectory evaluation is only done for assertions that have trajectory formulas as antecedents, i.e. for formulas g such that the cardinality of $\Delta^{\mathsf{t}}(g)$ is one. Besides the performance justification, experience with STE verification has shown that the main need for enriching the logic is to enrich the consequents rather than the antecedents. Moreover, the use of the compositional theory allows the enriching of antecedents indirectly (for example through the disjunction, until, and general transitivity rules). Nevertheless, even though experience so far with STE has not shown this to be a significant restriction, this is an undesirable restriction, and more work needs to be done here.

The final restriction is made with respect to the infinite operators such as `Global`. Since the state space being modelled is finite, all trajectories must have repeated states; thus, in principle, it is only necessary to investigate a prefix of a trajectory. However, this requires knowing when a state in a trajectory has been repeated. Since, in the tool, symbolic sequences represent a number of sequences or trajectories, given a symbolic sequence we have to know for which element in the sequence it is the case that for all interpretations of variables there have been repeated states. The parametric representation of state is unsuitable for this computation, which requires the characteristic representation to be used. However, if the characteristic representation is to be used, then the advantages of STE over other model checking approaches is reduced. If infinite formulas must be tested, other approaches may well be more suitable. Moreover, for hardware verification, infinite operators are less important than in more general situations since timing becomes more critical. We are not interested that after a given stimulus, output happens some time in the future; we want to know that output happens within x ns of input. Trajectory evaluation's good model of time and its ability to support verifications where precise timing is important is one of its great advantages. Finally, in the same way the compositional rules can be used to enrich the antecedent, they can be used to allow the infinite operators to be expressed usefully (the until rule and its corollaries are good examples here).

In summary, the STE-based algorithms proposed here check assertions of the form $\models \langle\!\langle\, A \Longrightarrow\!\!\rhd\, h \,\rangle\!\rangle$, where

1. A and h are in the realisable fragment of TL_n;
2. A is a trajectory formula; and
3. h does not contain any infinite operators.

Through the use of the compositional rules, the limitations of (2) and (3) can be partially overcome.

The rest of this chapter examines how Voss's ability to check formulas $\models \langle\!\langle\, A \Longrightarrow\!\!\!> C \,\rangle\!\rangle$ can be used effectively. Three algorithms are presented.

6.4.2 Direct Method

If A and C are trajectory formulas, the standard use of STE for model checking trajectory assertion of the form $\langle\!\langle\, A \Longrightarrow\!\!\triangleright C \,\rangle\!\rangle$ is straight-forward since the cardinality of $\Delta^{\mathbf{t}}(A)$ (and hence $T^{\mathbf{t}}(A)$) and $\Delta^{\mathbf{t}}(C)$ are one. δ^A and δ^C are constructed, and τ^A is computed from δ^A. The last part of the verification is to check whether $\delta^C \sqsubseteq \tau^A$.

Where we choose the consequent to be a general formula g of TL_n, we need to consider the entire set $\Delta^{\mathbf{t}}(g)$. However, the basic idea is the same: construct δ^A and compute τ^A, and then check whether $\forall \delta \in \Delta^{\mathbf{t}}(g), \delta \sqsubseteq \tau^A$. How this is done is sketched in the pseudo-code below:

Compute(g, j)=
case g of

$[i]$	:	$\tau_j^A[i] = \mathsf{H}$
$g_0 \wedge g_1$	:	Compute(g_0, j) $\wedge$ Compute(g_1, j)
`Next` g	:	Compute(g, $j+1$)
$\neg g$	:	$\neg$ Compute(g,j)
$\mathbf{t}$	:	$\mathbf{t}$
$\mathbf{f}$	:	$\mathbf{f}$

This algorithm is simple and straight-forward, although care must be taken in implementations to ensure efficiency, particularly when dealing with ADTs such as vectors and integers, and derived operators such as the bounded versions of `Global`. First, only necessary information must be extracted from τ^A. Second, a very important optimisation in the Voss tool is event scheduling — usually from one time step to the next only a few state holding elements change their values. By detecting that components are stable for long periods of time, much work can be saved. Any modifications to the STE algorithm must not interfere with this.

The way this was implemented in the prototype tool was to (i) determine from the consequent which state components are important, (ii) use Voss's trace facility to obtain the values of those components at relevant times, and (iii) compute whether the necessary relationship holds. All of this can be done in FL on top of Voss, obviating the need to make any changes to the underlying trajectory evaluation algorithm. Not only did this choice of implementation make developing the prototype much easier, but more fundamentally it means that the event scheduling capacity of Voss is not impaired in any way.

As a side note, modifications to this approach to deal with the infinite operators is, in principle, straightforward. At each step in the trajectory evaluation the set of reachable states is added to. Once a fix-point is reached, the trajectory evaluation can stop. The use of partial information might improve the performance of the modification (in some cases — but not all — once a state s has been explored, we need not visit any state above s in the information ordering). Provided we are prepared to pay the cost of computing the characteristic representation of the state space, this is feasible, although care must be taken not to conflict with the event scheduling feature of Voss.

6.4.3 Using Testing Machines

An alternative way of extending STE is through the use of testing machines. The goal is to determine, given a model $\mathcal{M}$, whether $\models_{\mathcal{M}} \langle\!\langle A \Longrightarrow\!\!> g \rangle\!\rangle$. The idea behind testing machines is to answer this question by constructing a model $\mathcal{M}'$ and a trajectory formula C_g such that $\models_{\mathcal{M}'} \langle\!\langle A \Longrightarrow\!\!> C_g \rangle\!\rangle$ if and only if $\models_{\mathcal{M}} \langle\!\langle A \Longrightarrow\!\!> g \rangle\!\rangle$. An analogous approach is the one adopted by Clarke *et al.* in extending their CTL model checking tool by using tableaux so that LTL formulas can be checked [38]. Other verification techniques also use this idea of using 'satellite' or observer processes to capture properties of systems [9].

As an example, using only trajectory formulas STE can check whether a node always takes on a certain value, y, say; it cannot check that a node *never* takes on the value y since the corresponding predicate is not simple and thus the question cannot be phrased as a trajectory formula. However, suppose the circuit were to have added to it comparator circuitry that compares the value on the node to y and sets a new node N with H if the node doesn't have the value y and L if it does. To check whether the node takes on the value can now be phrased as a trajectory formula. This section gives a brief outline of this, and detail can be found in Appendix B.

As presented, model checking takes a model and a formula and then performs some computation to check whether the model satisfies the formula. The basic motivation behind testing machines is that some of the computation task can be simplified by moving the computation within the model itself. In essence what we do is construct a circuit that performs the model checking, compose this circuit with the existing circuit and then do straightforward model checking on the new circuit. This task is simplified by the close relationship between $\mathcal{Q}$ and $\mathcal{C}$.

There are thus two steps in the model checking algorithm. The first is to take the formula and to construct the testing machine; the second is to compose the testing machine with the original circuit and to perform model checking.

The construction of the testing machine is done recursively based on the structure of the formula. An important part of the algorithm is constructing the testing machines for the basic predicates. For predicates dealing with boolean nodes, the construction is straightforward, essentially doing a type conversion. For other types — especially integers — it is somewhat more complex; for example, integer comparator and arithmetic circuits are needed. Given the testing machines for the basic predicates, there is a suite of standard ways in which these testing machines are composed, depending on the structure of the formula.

One of the complexities is dealing with timing information. For example, in the formula $g \vee h$, g and h may be referring to instants in time far apart. This will mean that the testing machines that compute g and h will probably produce their results in time instants far apart, which in turn means that some sort of memory may be necessary. If the formula $g \vee h$ is nested within a temporal operator such as the bounded always operator, it may be necessary to compute the values of $g \vee h$ for many instants in time, which means that a large number of results may need to be stored temporarily. This will affect the computation and memory costs of model checking.

The testing circuitry does not deal with the unbounded operators `Global` and `Exists`. The method of Section 6.4.2 could deal with these operators by recording the set of states already examined and other information. Testing circuitry could be built that duplicates that. As the state space is finite, we know that at some finite time all states will have been examined, but since the operators are unbounded it cannot be determined *a priori* at which instant in time all states will have been examined. Thus the scheme for examining the testing circuit at a particular moment fails. It seems that this can only be dealt with by modifying the STE algorithm.

6.4.4 Using Mapping Information

Suppose that A and C are trajectory formulas which have the property that no boolean variable in C appears in A. Let $\Phi_1 = \langle\!\langle A \Longrightarrow\!\!\triangleright C \rangle\!\rangle$. This is the set of assignments of boolean variables to values for which $A \Longrightarrow\!\!\triangleright C$. In particular, it describes the relationship between the variables in the antecedent formula and the variables in the consequent formula which must hold for the trajectory evaluation to succeed.

C essentially extracts relevant components of τ^A, so by making C general enough, enough useful mapping information can be used to make model checking TL_n feasible. Provided enough information is extracted, we can use Φ_1 to determine whether $\models \langle\!\langle A \Longrightarrow\!\!\triangleright g \rangle\!\rangle$ holds: if for all interpretations in Φ_1, g holds (this is formalised later), and provided some side conditions hold, then so does $\models \langle\!\langle A \Longrightarrow\!\!\triangleright g \rangle\!\rangle$.

An example will illustrate the method.

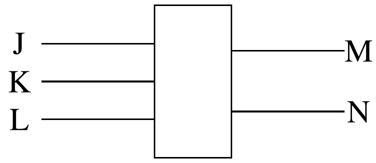


Figure 6.3: A CSA Adder

The Carry-Save adder (CSA) shown in Figure 6.3 is used in a number of arithmetic circuits. An n-bit CSA adder consists of n independent single-bit full adders. For simplicity in the example, we consider a one-bit adder. Suppose that at time 0, the state of the circuit is such that node J has the value j, node K has the value k, and L has the value l. Then at time 1, the state of the circuit should be such that M has the value $j \oplus k \oplus l$ ($\oplus$ representing exclusive or) and N has the value $j \wedge k \vee j \wedge l \vee k \vee l$.

This is easy enough to verify using a trajectory formula. However, there are verifications in which what we are interested in is not what the particular values of nodes M and N are, but that the sum of the values of nodes J, K and L at time 0 is equal to the sum of the nodes M and N at time 1. (This is exactly what we are interested in when verifying a Wallace-tree multiplier). In STE, this property can not be verified directly.

Define the trajectory formulas A and C by:

$A = (J = j) \wedge (K = k) \wedge (L = l)$

$C = \texttt{Next}\,(M = m) \wedge (N = n)$

and then compute $\Phi_1 = \langle\!|\, A \Longrightarrow\!\!\triangleright C \,|\!\rangle$.

Φ_1 gives the constraints which must hold for the trajectory evaluation to hold. In particular it gives the constraints relating j, k and l with m and n. Suppose $\forall \phi \in \Phi_1, \phi(g) = \mathtt{t}$ where $g = (j + k + l = m + n)$ (assuming here two-bit addition). If this is the case then we know that for each mapping of boolean variables to values for which the STE holds, $(j + k + l = m + n)$. Or putting this in terms of an expression in TL_n, that $h = \texttt{Next}\,(j + k + l = [M] + [N])$ holds.

Essentially g is h where we substitute the variables m and n into h to act as place-holders for the state components M and N. C is a way of extracting appropriate values out of τ_A. So, if $\forall\phi \in \Phi_1, \phi(g) = \mathtt{t}$ and $\Phi_1 = \langle\!\langle A \Longrightarrow C \rangle\!\rangle$ then $\langle\!\langle A \Longrightarrow h \rangle\!\rangle$.

One important check needs to be made – we must ensure that the above condition is not satisfied vacuously by Φ_1 being empty or only containing very few interpretations. What we want to ensure is that Φ_1 covers all the interesting cases: that for every possible assignment of values to the boolean variables j, k and l, there is an assignment to the boolean variables m and n such that the trajectory evaluation holds. This is formalised now.

Definition 6.1.

Let U be a set of variables, and $\phi\colon \mathcal{V} \to \mathcal{B}$ be an interpretation of variables. The set of extensions of ϕ with respect to U is $Ext(\phi, U) = \{\psi \in \Phi : \forall v \in \mathcal{V} - U, \phi(v) = \psi(v)\}$ □

The condition that Φ_1 is non-trivial can be expressed as: *for every interpretation $\phi \in \Phi$, there is an interpretation $\psi \in Ext(\phi, v(A))$ where $v(A)$ is the set of variables in A, and $\psi \in \Phi_1$.* In other words for every interpretation of variables of A there is an extension of that interpretation to include variables in C, such that the extension is an element of Φ_1.

Note: If h' is a TL_n formula not containing temporal operators, then by the remarks preceding Theorem 3.5, then we can consider h' as a predicate from $\mathcal{C}^n$ to $\mathcal{Q}$. For convenience, if h' is strictly dependent on nodes $\{n_1, \ldots, n_r\}$, then we write $h'(x_1, \ldots, x_r)$.

Theorem 6.1.

Let A be a trajectory formula, and $h = \mathtt{Next}\, h'$ be a TL_n formula such that h' contains no temporal operators. Let h' be strictly dependent on $N = \{n_1, \ldots, n_r\}$.

Let $C = \mathtt{Next}\,(\wedge_1^r [n_j] = w_j)$ where the w_j are distinct and disjoint from the variables in A; let $W = \{w_1, \ldots, w_r\}$. Suppose:

1. $\Phi_1 = \langle\!\langle A \Longrightarrow C \rangle\!\rangle$;

2. $\forall \psi \in \Phi_1, \psi(h'(w_1, \ldots, w_r)) = \mathbf{t}$;

3. $\forall \phi \in \Phi, \exists \psi \in \mathit{Ext}(\phi, W)$, and $\psi \in \Phi_1$; and

4. $\forall \psi \in \Phi_1; i = 0, 1; j = 1, \ldots, n: \tau_i^{\psi(A)}[j] \neq \mathsf{Z}$.

Then $\models \langle\!| A \Longrightarrow\!\!\triangleright h |\!\rangle$.

Proof.

(1)	$\forall \phi \in \Phi, \exists \psi \in \mathit{Ext}(\phi, W), \psi \in \Phi_1$	Hyp. 3.
(2)	$\forall \phi \in \Phi, \exists \psi \in \mathit{Ext}(\phi, W), \psi(A) \Longrightarrow\!\!\triangleright \psi(C)$	(1), Hyp. 1.
(3)	$\forall \phi \in \Phi, \exists \psi \in \mathit{Ext}(\phi, W), \delta_1^{\psi(C)} \sqsubseteq \tau_1^{\psi(A)}$	(2), Theorem 4.5.
(4)	$\forall \phi \in \Phi, \exists \psi \in \mathit{Ext}(\phi, W), \delta_1^{\psi(C)}[k] \sqsubseteq \tau_1^{\psi(A)}[i], k = 1, \ldots, n$	(3), structure of $\mathcal{C}^n$.
(5)	$\forall \phi \in \Phi, \exists \psi \in \mathit{Ext}(\phi, W), \psi(w_j) \sqsubseteq \tau_1^{\psi(A)}[n_j], j = 1, \ldots, r$	(4), structure of C.
(6)	$\forall \phi \in \Phi, \exists \psi \in \mathit{Ext}(\phi, W), \psi(w_j) = \tau_1^{\psi(A)}[n_j], j = 1, \ldots, r$	(5), Hyp. 4.
(7)	$h'(\psi(w_1), \ldots, \psi(w_r)) = \mathbf{t}$	Hyp. 2, definition of $\psi(h')$.
(8)	$\forall \phi \in \Phi, \exists \psi \in \mathit{Ext}(\phi, W), h'(\tau_1^{\psi(A)}[n_1], \ldots, \tau_1^{\psi(A)}[n_r]) = \mathbf{t}$	(6), (7).
(9)	$\forall \phi \in \Phi, h'(\tau_1^{\phi(A)}[n_1], \ldots, \tau_1^{\phi(A)}[n_r]) = \mathbf{t}$	(8), w_j not in A.
(10)	$\forall \phi \in \Phi, \mathit{Sat}(\tau^{\phi(A)}, \phi(h)) = \mathbf{t}$	(9).
(11)	$\forall \phi \in \Phi, \phi(A) \Longrightarrow\!\!\triangleright \phi(h)$	(10), Lemma 4.4, Hyp. 4.

$\square$

This theorem can be generalised to deal with assertions such as

$$\models \langle\!| A \Longrightarrow\!\!\triangleright (\bigwedge_{j=1}^{d} \mathtt{Next}^j h_j) |\!\rangle$$

and implemented.

Chapter 7

Examples

This chapter shows that the ideas presented in this thesis can be used in practice. The verification of the examples done in this chapter requires a relatively rich temporal logic — trajectory formulas are often not rich enough — and efficient methods of model checking. Efficient algorithms for performing STE are essential, but in themselves not enough; the compositional theory for TL is necessary.

Section 7.1 presents the verification of a number of simple examples performed using the first prototype verification tool. These examples are used as illustrations of the use of the inference rules. Section 7.2 presents an example verification of a circuit that is well suited for verification by traditional BDD-based model checkers such as SMV. The B8ZS encoder chip verified has a small state space which is easily tractable by the traditional methods. While the circuit can easily be represented as a partially ordered model, it is difficult to use the methods proposed by this thesis to verify this circuit completely. This example shows some interesting points about the need for expressive logics, and shows some limitations of the approach proposed in this thesis.

Section 7.3 describes the verification of more substantial circuits, multipliers, which can have up to 20 000 gates. These are circuits that are beyond BDD-based automatic model checkers and require the use of methods such as composition and abstraction. The verification of a number of different multipliers are described and analysed.

One of the verified multipliers is Benchmark 17 of the IFIP WG10.5 Hardware Verification Benchmark Suite. Section 7.4 builds on the verification of this multiplier and shows how its verification is used in the verification of a parallel matrix multiplier circuit (Benchmark 22 of the suite); the largest version of this circuit verified contains over 100 000 gates.

The examples mentioned here all show that these methods are well suited to examples where detailed timings at which events happen is known. Section 7.5 shows how time can be dealt with in a more generalised way. Although this section is more speculative in nature (the verification has not been mechanised) it shows that using the inference rules and inducting over time, allows the practical use of TL in a more expressive way.

Finally, Section 7.6 summarises the results of this chapter and evaluates the methods proposed.

7.1 Simple Example

7.1.1 Simple Example 1

For the first example, consider the circuit shown in Figure 7.1 which takes in three numbers m, n and o on nodes M, N and O, and produces $o + max(m, n)$ on R. There are three parts to the circuit: a comparator compares the value on M with the value on N and produces H on P if the number at M is bigger than the number on N and produces 0 otherwise; a selector takes the values at M, N and O and produces at node Q the value at M if P is set to H, and the value at N otherwise; the third part of the circuit takes the values at node Q and O, and produces their sum at node R. This example is one which could be verified using STE alone, but its small size makes it useful as an example.

Verification starts by checking the correctness of the individual components. The verification of each component is done in the presence of the rest of the system, which means that any unintended interference will be detected. These individual proofs are put together using specialisation, time-shifting and transitivity. An outline of the formal proof follows. To simplify notation, $a \rightarrow b|c$ is used as shorthand for $(a \Rightarrow b) \wedge ((\neg a) \Rightarrow c)$.

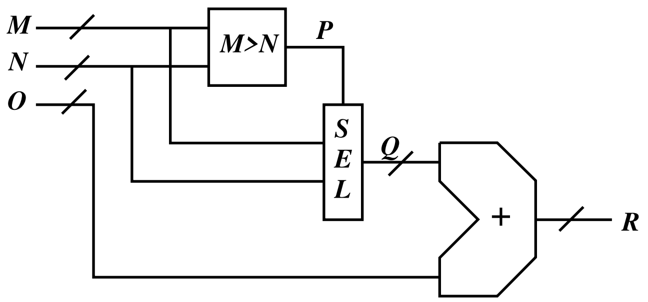


Figure 7.1: Simple Example 1

Let $\mathcal{A}' = ([M] = m) \wedge ([N] = n) \wedge ([O] = o)$.

Let $\mathcal{A} = \mathcal{A}' \wedge \texttt{Next}\, \mathcal{A}' \wedge \texttt{Next}^2 \mathcal{A}'$.

Let $\mathcal{C} = \texttt{Next}^3 (m > n \ \rightarrow\ [R] = m + o \mid [R] = n + o)$.

We wish to show that $\models \langle\!\langle\, \mathcal{A} \Longrightarrow\!\triangleright \mathcal{C} \,\rangle\!\rangle$.

$(1)\quad \models \langle\!\langle\, \mathcal{A} \Longrightarrow\!\triangleright \texttt{Next}\, ([P] = (m > n)) \,\rangle\!\rangle$ — By STE

$(2)\quad \models \langle\!\langle\, \mathcal{A}' \wedge [P] = x \Longrightarrow\!\triangleright \texttt{Next}\, (x \rightarrow\ [Q] = m \mid [Q] = n) \,\rangle\!\rangle$ — By STE

$(3)\quad \models \langle\!\langle\, ([O] = y) \wedge ([Q] = z) \Longrightarrow\!\triangleright \texttt{Next}\, ([R] = y + z) \,\rangle\!\rangle$ — By STE

$(4)\quad \models \langle\!\langle\, \texttt{Next}\, (\mathcal{A}' \wedge ([P] = m > n)) \Longrightarrow\!\triangleright \texttt{Next}^2 (m > n \ \rightarrow\ [Q] = m \mid [Q] = n) \,\rangle\!\rangle$

Time-shift, specialise (2)

$(5)\quad \models \langle\!\langle\, \mathcal{A} \Longrightarrow\!\triangleright \texttt{Next}^2 (m > n \ \rightarrow\ [Q] = m \mid [Q] = n) \,\rangle\!\rangle$ — (1), (4), transitivity.

$(6)\quad \models \langle\!\langle\, \texttt{Next}^2 ([O] = o) \wedge (m > n \ \rightarrow\ [Q] = m \mid [Q] = n) \Longrightarrow\!\triangleright \mathcal{C} \,\rangle\!\rangle$

Time-shift, specialise (3)

$(7)\quad \langle\!\langle\, \mathcal{A} \Longrightarrow\!\gg \mathcal{C} \,\rangle\!\rangle$ — (5), (6), transitivity.

Perhaps the most interesting part of this proof is how specialisation and transitivity are used. Consider how (1) and (2) are combined. Note that $\mathcal{A}$ contains all the information that $\texttt{Next}\, \mathcal{A}'$ does; and note the similarity in structure between $[P] = (m > n)$ and $[P] = x$. By time-shifting (2) as well as substituting $m > n$ for x transforms (2) into (4), which can be combined with (1) using transitivity.

The other place in which specialisation was used was in line (6). Here, using only substitution on line 3 is inadequate; a much richer transformation is needed. Rather than just substituting one expression for z in (3), two different substitutions are made, which are qualified and combined (one substitution is made when $m > n$, and the other when $m \leq n$).

This proof was done in the first verification tool, where it is easier to do than manually because the time-shifts and specialisations are found automatically. Steps 4 and 5 are done with a call to one of the automated rules; and steps 6 and 7 with another call to the same rule. A full description can be found in [76], and the FL proof script can be found in Section C.1.

7.1.2 Hidden Weighted Bit

The hidden weighted bit problem was one of the first to be proved to need exponential space to verify using traditional BDD-based methods [21]. A circuit for an 8-bit version is shown in Figure 7.2. The verification of this was done in the first prototype tool; the proof is outlined here, and a full description including the one page proof script is described in a technical report [76].

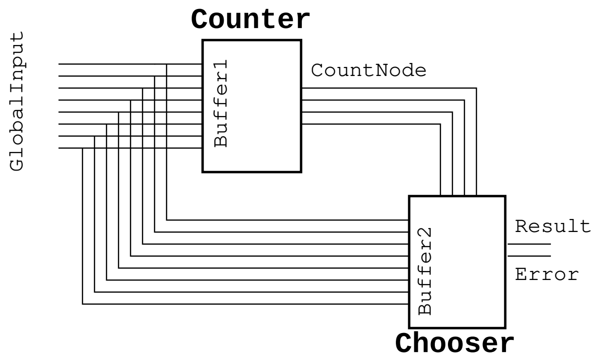


Figure 7.2: Circuit for the 8-bit Hidden Weighted Bit Problem

In this version, the global input $x_1, \ldots, x_n$ is copied to two buffers. The *Counter* part of the circuit computes the number of 1's on the input (i.e. $\Sigma_1^n x_i$). The *Chooser* part of the circuit takes the number j output on *CountNode* (the number is in binary form, hence if there are n input lines, *CountNode* comprises $\lfloor \lg n \rfloor + 1$ lines), and outputs the value x_j on *Result* and on *Error* when $j > 0$. If $j = 0$ then *Error* is set to 1.

Intuitively, a verification of this circuit *as a whole* takes exponential time and space (in n) because the output value on *CountNode* is so complicated, in terms of the boolean variables, that no suitable variable ordering can be found so that the *Chooser* part of the circuit can be verified efficiently. The virtue of the compositional approach is clearly illustrated: by decoupling the verification of the two parts of the circuit, we can choose suitable individual variable orderings for both parts of the circuit; moreover, it is more efficient to verify the chooser circuit for an arbitrary input j (which only needs very simple BDDs to represent it), and then substitute for j the actual input, than to verify for the actual input (which needs more complicated BDDs to represent it).

There are five steps in the proof, in which all the time-shifts and specialisations are found automatically.

- The proof that the copying of the input to the buffer is correct — *BufferTheorem*.
- The verification *Counter* part of the circuit — *CounterTheorem*.
- The composition of *BufferTheorem* and *CounterTheorem*. This is done in two steps: first, *CounterTheorem* is time-shifted along so that transitivity between *BufferTheorem* and *CounterTheorem* can be used to produce *BufferCounterTheorem*. *BufferCounterTheorem* is conjoined with *BufferTheorem* so that we can use the value of *Buffer2* at a later stage. Call the result of this *stage1Theorem*.
- Verification of the *Chooser* part of the circuitry — *ChooserTheorem*.
- Composition of *stage1Theorem* and *ChooserTheorem* by time-shifting *ChooserTheorem* by an appropriate amount and specialising this so that transitivity can be used between

stage1 and *ChooserTheorem*.

Results: We verified the circuit for different values of n (4, 8, 16, 32, 64, 128). For these values, verification takes roughly cubic time (and importantly, space was not an issue). The verification of the 128 bit problem took just under 27 minutes on a Sun 10/51. Compared to this, verification of the system as one unit was not possible for $n = 64$ or larger. The FL script for the verification is shown in Section C.2

7.1.3 Carry-save Adder

The carry save adder (CSA) shown in Figure 6.3 was verified using all three extensions to the STE algorithm described in Chapter 6. Table 7.3 summarises the computational cost of verification of a 64 bit CSA.

	Algorithm	Time (s)
1	Direct	3.8
2	Testing Machine	3.6
3	Mapping information	2.6

Table 7.3: CSA Verification: Experimental Results

The experiments were run on a DEC Alpha 3000, and show that for all three approaches, verification is easily accomplished. The FL script for this is shown in Section C.3. Note that the compositional theory is not used to verify this circuit.

7.2 B8ZS Encoder/Decoder

This example shows the verification of a B8ZS encoder, a very simple circuit but one which would be very difficult to do in traditional STE and illustrates some points about the style of verification. Note that the compositional theory is not used to verify this circuit.

7.2.1 Description of Circuit

Bipolar with eight zero substitution coding (B8ZS) is a method of coding data transmission used in certain networks. Some digital networks use Alternate Mark Inversion: zeros are encoded by '0', and ones are encoded alternately by '+' and '$-$'. The alternation of pluses and minuses is used to help resynchronise the network. If there are too many zeros in a row (over fifteen – something common in data transmission) the clock may wander. B8ZS encoding is used to encode any sequence of eight zeros by a code word. If the preceding 1 was encoded by '$+$', then the code word '000+–0–+' is substituted; if the preceding 1 was encoded with a '$-$', then the code word is '000–+0+–'. Using this encoding, the maximum allowable number of consecutive zeros is seven.

The implementation of the circuit is taken from the design of a CMOS ZPAL implementation of the encoder (and corresponding decoder) by Advanced Micro Devices [4]. The encoder comprises two parts. One PAL detects strings of eight zeros and delays the input stream to ensure alignment. If the first PAL detects eight zeros, the second PAL encodes the data depending on whether eight zeros have been detected or not. Figure 7.3 given an external view of the chip. The inputs are a reset line (active low), and NRZ_IN which provides the input. There are two outputs, PPO and NPO which as a pair represent the encoding: (1,0) is the '$+$' encoding of a one, (0, 1) is the '$-$' encoding of a one, $(0,0)$ encodes a zero, and (1, 1) is not used. Output emerges six clock cycles after input.

7.2.2 Verification

There are two questions one could ask in verification:

1. Does the implementation meet its specification? Here we want to check that the output we see on NPO and PPO is consistent with the input.

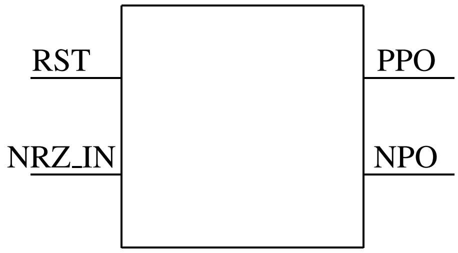


Figure 7.3: B8ZS Encoder

2. Does the implementation have the properties that we expect? (Specification validation) In particular is it the case that:
   - At no stage are there eight consecutive (PPO, NPO) pairs which encode a zero;
   - At no stage are there fifteen or more consecutive zeros on the PPO output; and
   - At no stage are there fifteen or more consecutive zeros on the NPO output.

Checking that the implementation meets the specification is a bit tricky, and shows the need for a richer logic than the set of trajectory formulas. With trajectory formulas, the obvious way to perform verification is to examine the output and check to see that the output produced is determined by the finite state machine which the PALs implement. However, the equations of the FSM are complicated and non-intuitive. Verification that the implementation is 'correct' doesn't give us information about the specification. Worse, essentially the verification conditions would be a duplicate of the implementation, increasing the likelihood of an error being duplicated. And there don't seem to be easier, higher level ways of expressing correctness using trajectory formulas since the circuit has the property that the n-th output bit is dependent on the first input bit.

Using the richer logic, a far better way of verifying the circuit is to show that the input can be inferred from the output. Suppose that we want to check the output bit pair at time k (recall that the output is encoded as the (PPO, NPO) pair). If this bit pair is in the middle of one of the code words then the input bit at time $k - 6$ must be a zero; otherwise the $(k - 6)$-th input bit can be inferred directly from the value of the bit pair.

The testing machine method was used in verification. To test that the bits are correctly translated, the proof first shows that after being reset the encoder enters a set of reachable states, and that once in a reachable state the encoder remains in this set of states. Next, the proof shows that if the encoder starts in the reachable set then the output of the encoder is correct. The computational cost of all of this is approximately 30s on a Sun 10/51.

The second step is to check that the implementation has properties that cannot be directly inferred from the design. In particular we want to show that at no stage are there eight or more zeros consecutively produced by the encoding of PPO and NPO and also that if we look at PPO and NPO individually that at no stage are there fifteen or more zeros consecutively. These conditions can be expressed succinctly in TL, while they could not be expressed as trajectory formulas. The major restriction here is that using testing machines, the antecedent can only be a finite formula. We cannot check that this result holds for arbitrary input. What we can show is that given arbitrary input of length n the circuit has the properties we expect. Using testing machines, verification for $n = 100$ presents no problem (10s on a Sun 10/51). In principle, the direct method could verify the general case.

The final verification that was done was to implement the complementary B8ZS decoder and to check that when the output of the encoder is given as input to the decoder, then the output of the decoder is just the input of the encoder, suitably delayed. Again, it was possible using the testing machine method to check this for finite input prefixes. An error was detected: the initial states of the encoder and decoder are not synchronised. If the first eight input bits given the encoder are zero, the code word used by the encoder is ‘000+-0-+’; however, the decoder expects the other code word to be used if the first eight bits are zero. This error only occurs when the first eight bits are zero as the state transition table of the decoder has the pleasant property that the first encoded 1 (either a '+' or '$-$') emitted by the encoder synchronises the decoder.

This example illustrates some interesting points about verification. However, it is not a good example for trajectory evaluation; since the state space of the circuit is quite small (fewer than 20 state holding components), other verification methods work well.

7.3 Multipliers

Since BDDs are not able to represent the multiplication of two numbers efficiently [21], automatic model checking algorithms find the verification of multipliers very challenging. For this reason, multipliers have received much attention in the literature. The methods proposed in this thesis have been used successfully to verify a number of multipliers: three of these examples are briefly discussed, and then one case is presented in great detail. The section concludes by comparing these verification studies to other work.

7.3.1 Preliminary Work

The first multiplier verified using a compositional theory for STE was a simple n-bit multiplier consisting of n full adders. The verification is accomplished by using STE to prove that each adder works correctly, and then by applying the inference rules to show that the collection of adders performs multiplication. The key inference rules used were time-shifting, specialisation, transitivity, and rules of consequence.

Of immense practical importance in the prototype tool used to perform the verification was the ability to use a simple theorem prover coupled with some decision procedures to reason about integers. This enabled the tool to break the limitation of BDDs. Also important for ease of use of the system is that specialisations and time-shifts were all found automatically by the tool.

The complete verification of this 64 bit multiplier took just less than 15 minutes of CPU time on a Sun 10/51. For this verification, trajectory formulas were sufficient to express all needed properties. A full description of the verification, including the proof script can be found in a technical report [76].

The next step — the verification of a Wallace tree multiplier [66] — showed the need for a richer logic. A Wallace tree multiplier uses Carry-Save adders (CSA) as its basic components. Example 5.9 indicated that what is important in the verification of a CSA is to show that the sum of the two outputs is the sum of the three inputs. This cannot be represented as a trajectory formula. What trajectory formulas can represent is the particular values of each output, which is not helpful.

As a preliminary test, the mapping method was used to extend the expressiveness of trajectory evaluation based verification, and the verification completed. The implementation of the prototype algorithm was not particularly efficient, but the need for a richer logic, and the feasibility of the approach was demonstrated.

7.3.2 IEEE Floating Point Multiplier

One of the largest verifications done using the theory presented in this thesis is the verification of an IEEE compliant floating point multiplier by Aagaard and Seger [2]. The multiplier, implemented in structural VHDL, includes the following features:

- double precision floating point;
- radix eight multiplier array with carry-save adders;
- four stage pipeline; and
- three 56-bit carry-select adders.

The circuit verified is approximately 33 000 gates in size.

The verification was done using the VossProver, a proof system built by Seger on top of Voss. Based on the first prototype tool discussed here, this implements the theory presented in [78], augmented by using the mapping approach to allow a more expressive logic than trajectory formulas. The VossProver contains extensive integer rewriting routines, which are very important in verification proofs.

Aagaard and Seger estimate that verifying the circuit took approximately twenty days of work. The computational cost of the verification was reasonable (a few hours on a DEC Alpha 3000).

7.3.3 IFIP WG10.5 Benchmark Example

Description of Circuit

Benchmark 17 of the IFIP WG10.5 Benchmark Suite is a multiplier which takes two n-bit numbers and returns a $2n$ bit number representing their multiplication. This description is heavily dependent on the IFIP documentation.[1]

Let $A = a_{n-1} \ldots a_1 a_0$ and $B = b_{n-1} \ldots b_1 b_0$. Then $A \times B = \sum_{i=0}^{n-1} 2^i (\sum_{j=0}^{n-1} 2^j a_i b_j)$. Implementing this is straightforward: the basic operation is multiplying one bit of A with one bit of B and adding this to the partial sum. The component that accomplishes this basic operation takes four inputs:

a One bit of the multiplicand,

b One bit of the multiplier,

c One bit of the partial sum previously computed,

CIN A one bit carry from the partial sum previously computed;

and computes $a * b + c + \mathit{CIN}$ producing two outputs:

S One bit partial sum, and

[1] `ftp://goethe.ira.uka.de/pub/benchmarks/Multiplier/`

COUT One bit carry.

The equations for the output are:

$$S = a \wedge b \oplus (c \oplus \mathit{CIN})$$

$$\mathit{COUT} = a \wedge b \wedge c \vee a \wedge b \wedge \mathit{CIN} \vee c \wedge \mathit{CIN}$$

The implementation of the equations (as given in the IFIP documentation) and the graphical symbol used to represent these components is presented in Figure 7.4.

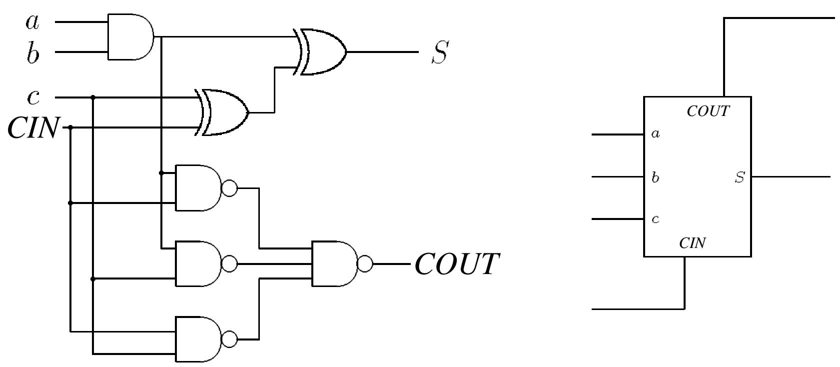


Figure 7.4: Base Module for Multiplier

A vector of these components multiplies one bit of B with the whole of A and adds in any partial answer already computed. It might seem appropriate rather than just having a vector of these components to also have an adder which added in carries from less significant columns to the results of more significant bits. The problem with doing that is that each stage would be limited by the need for possible carries from the least significant bit to be propagated to the most significant bit, with concomitant increases in the time and number of gates needed.

The approach used in the implementation is to produce two outputs: the first output is the sum of the bit-wise addition of the two inputs, ignoring the carries; and the second output is the carries of the bit-wise addition. Both of these outputs are forwarded to the next stage; here the carries are added in and new carries generated. We can consider the vector of S outputs as one n-bit number and the vector of *COUT* outputs as another n-bit number. If we consider stage k by itself, if the vector of a inputs is $\tilde{x}$, if the b inputs are all the bit y, and if the vector of c inputs is $\tilde{z}$, then we shall have that $S + 2^{k+1}\mathit{COUT} = \tilde{x}y + z$. (This is something that must be proved in the verification.)

These components are arranged in a grid (Figure 7.5 shows how a 4 bit multiplier is arranged). The multiplier contains n stages, each of which multiplies one bit of B with A and adds it to the partial result computed so far. After k stages, $n + k$ bits of the partial answer have been computed. The components making up each stage are arranged in columns in the figure. The components making up a row compute one bit of the final answer; carries from less significant bits are added in, and any generated carries are output for the more significant rows to take care of.

In the Figure 7.5, each of the base components is labelled with indices: $i\colon j$ indicates that the component is the j-th component of the i-th stage.

Having passed through n stages, the full multiplication has been computed. However, as the final stage still outputs two numbers, the carries must now all be added in. Therefore the final step in the multiplier is a row of $n - 1$ full adders that adds in carries. These full adders are labelled **FA** in Figure 7.5.

The implementation of the circuit was done in Voss's EXE format as a detailed gate-level description of the circuit. A unit-delay model was used, although this is essential neither to the implementation nor the verification.

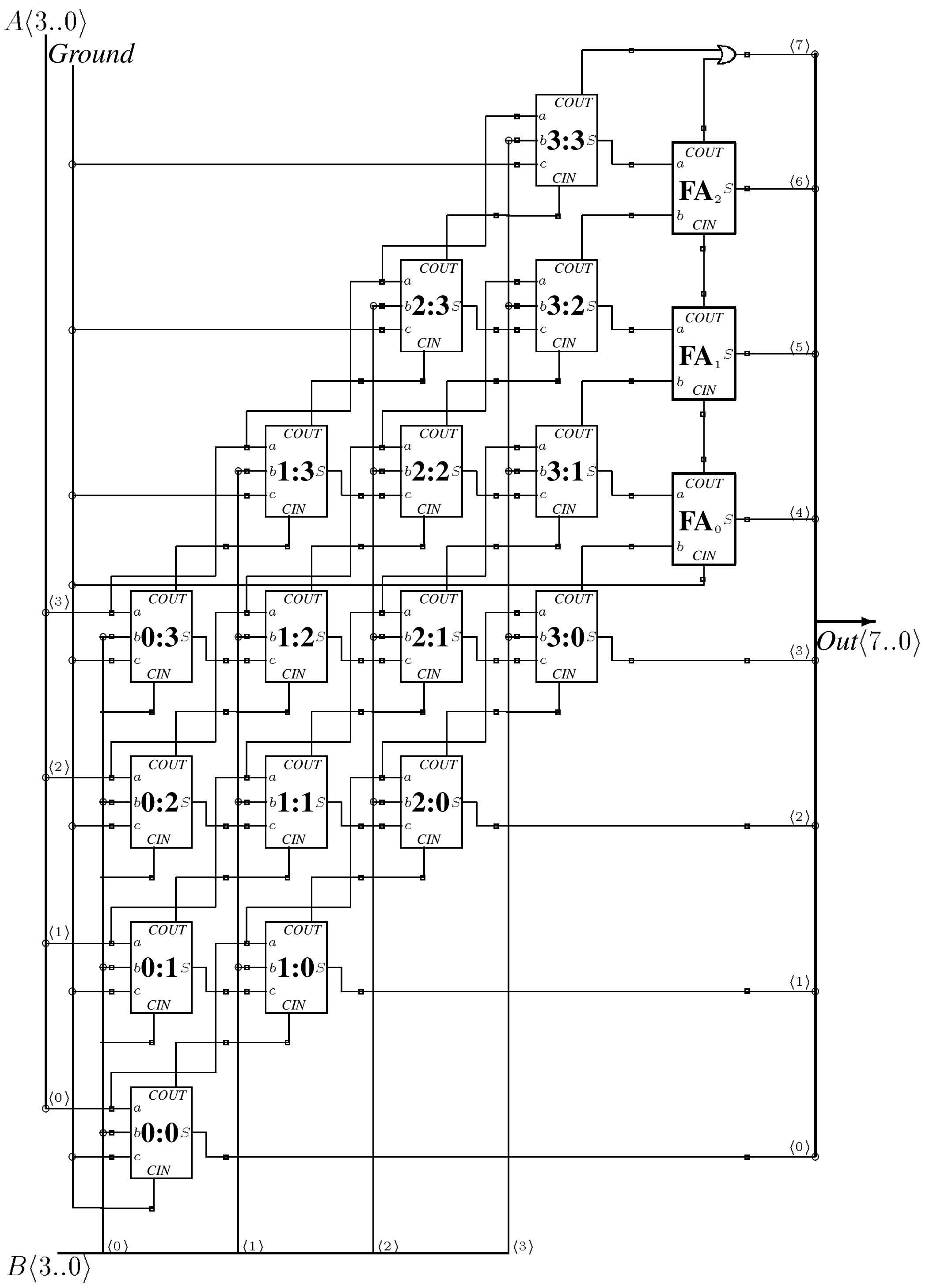


Figure 7.5: Schematic of Multiplier

Verification

This section presents a detailed description of the verification of the four bit multiplier presented in Figure 7.5. This example is small enough that the complete proof can be described, and this is useful to show how the inference rules are used. However, the example is big enough that there is some tedium involved too; it must be emphasised that in practice the verification is done using FL as the proof script language, which alleviates much of the tedium.

It is also worth mentioning that the verification of a four bit multiplier is well within the capacity of trajectory evaluation. Although the proof is not independent of data path width since issues of timing are important, it may be useful to do the verification for a small bit width first using trajectory evaluation by itself.

Identifying structure Using the inference rules relies on using the properties of integers to break the limitations of BDDs. Therefore, the first step in the proof is to identify some structure, in particular to identify which collections of nodes should be treated as integers.

Notation: *BM*$(i\colon j)(x)$ refers to node x in the basic module $i\colon j$; *FA*$_i(x)$ refers to node x of the full adder *FA*$_i$. For each stage, we consider the collection of a inputs as an integer, the collection of b inputs as an integer, and so on ... Similarly, the collection of S outputs and *COUT* outputs are both considered as integers. Table 7.4 presents the correspondences.

The following bit vector *variables* are used:

a stands for the bit vector $\langle a_3, \ldots, a_0\rangle$;

b stands for the bit vector $\langle b_3, \ldots, b_0\rangle$ (a and b are the inputs to the circuit);

c stands for the bit vector $\langle c_7, \ldots, c_0\rangle$;

d stands for the bit vector $\langle d_2, \ldots, d_0\rangle$.

If N is a bit vector, then $N\langle i\rangle$ refers to the i-th least significant bit (so $N\langle 0\rangle$ is the least significant bit), and $N\langle i..j\rangle$ refers to the (sub)bit vector $\langle N\langle i\rangle, \ldots, N\langle j\rangle\rangle$. We also use the

Integer node	Vector of bit nodes
A	The four bit integer input
B	The four bit integer input
O	Output of the or gate
RS_i	S output of stage i for $i = 0, \ldots, 3$
	$\langle \mathit{BM}(i\colon 3)(S), \ldots, \mathit{BM}(i\colon 0)(S), \mathit{BM}(i-1:0)(S), \ldots, \mathit{BM}(0:0)(S)\rangle$
RC_i	*COUT* output of stage i for $i = 0, \ldots, 3$
	$\langle \mathit{BM}(i\colon 3)(\mathit{COUT}), \ldots, \mathit{BM}(i\colon 0)(\mathit{COUT})$
RS_4	The output *Out*
	$\langle O, \mathit{FA}_2, \ldots, \mathit{FA}_0, \mathit{BM}(3\colon 0)(S), \ldots, \mathit{BM}(0\colon 0)(S)\rangle$

Table 7.4: Benchmark 17: Correspondence Between Integer and Bit Nodes

short hand that $RC_i = d$ is short for $RC_i\langle 2..0\rangle = d$ (RC_i is four bits wide, d is three bits wide).

Defining this correspondence has two advantages: the level of abstraction is raised since the verifier can think in terms of integers rather than bit vectors; and the verifier can use properties of integers to prove theorems without having to convert everything into BDDs.

Anomalies in circuit implementation There are a number of aspects of the circuit that can be criticised and improved. The most obvious is that $\mathit{BM}(i\colon 3)(\mathit{COUT}) = 0$ for all i. In turn, this means that one of the inputs to the or gate is always 0, i.e. $RS_4\langle 7\rangle$ depends entirely on $\mathit{FA}_2(\mathit{COUT})$. The only advantage of this implementation is that it makes the circuit description (slightly) more regular. The cost is the extra circuitry and time required to perform the computation. Furthermore, this implementation makes the proof more complicated. The final step in the proof below will be to show that since $RS_3 + 2^4 RC_3 = ab$ that $RS_4 = ab$; this is only true because the one input to the or gate is zero. Therefore, as the proof is constructed, we shall prove that $\mathit{BM}(i, 3)(\mathit{COUT}) = 0$, complicating the proof slightly. A better implementation would have meant a simpler proof.

The Proof

Stage 0 The first step is to show the first stage performs the correct multiplication/addition.

$\models$ By STE

$$\langle\!| \, \texttt{Global}\,[(0,100)]\ ([A]=a \wedge [B]\langle 0\rangle = b_0) \tag{7.1}$$
$$\Longrightarrow \quad \texttt{Global}\,[(3,100)]\ ([RS_0] + 2^1[RC_0] = ab\langle 0\rangle \wedge [RC_0]\langle 3\rangle = 0)\, |\!\rangle$$

To make STE as efficient as possible, we use as little information as possible by considering only one bit of b. However, at a later stage we shall want to use all the bits of b, so the next step is to include the rest of b in the result. There are a number ways of doing this. One would be to use the identity rule to show that B has any value imposed on it and then use conjunction with Result 7.1. However, in this case it is easier to use one of the rules of consequence (Theorem 5.18) and strengthen the antecedent.

$\models$ By rule of consequence from Result 7.1

$$\langle\!| \, \texttt{Global}\,[(0,100)]\ ([A]=a \wedge [B] = b) \tag{7.2}$$
$$\Longrightarrow \quad \texttt{Global}\,[(3,100)]\ ([RS_0] + 2^1[RC_0] = ab\langle 0\rangle \wedge [RC_0]\langle 3\rangle = 0)\, |\!\rangle$$

This use of the rule of consequence relies on Lemma 5.27, and is motivated by the fact that the antecedent of Result 7.2 uses more information than that of Result 7.1

Stage 1 The first step is to show Stage 1 performs the correct multiplication/addition. Note, the proof is done for arbitrary input for RS_0 and RC_0 rather than the actual input. This is important because STE is used to do the proof; if the actual input (which is a function of A and B) were used, in general STE would not be able to cope.

$\models$ By STE

$$\begin{array}{l} \langle\!| \, \texttt{Global}\,[(3,100)] \\ \qquad [A]=a \,\wedge\, [B]\langle 1\rangle = b_1 \,\wedge\, [RS_0]=c\langle 3..0\rangle \,\wedge\, [RC_0]=d \,\wedge\, [RC_0]\langle 3\rangle = 0 \\ \quad \Longrightarrow \quad \texttt{Global}\,[(6,100)] \\ \qquad\qquad [RS_1] + 2^2[RC_1] = c\langle 3..0\rangle + 2^1 d + 2^1 ab\langle 1\rangle \,\wedge\, [RC_1]\langle 3\rangle = 0 \,|\!\rangle \end{array} \tag{7.3}$$

In proving this result, STE is used; this implies that BDDs are used to represent data as this is necessary for STE. However, once the proof is done, the result is only stored symbolically, and the BDDs used to represent Result 7.3 are garbage collected.

Having proved this, we now combine Results 7.2 and 7.3 using a combination of transitivity and specialisation. This is useful to do since we know something about the values of RS_0 and RC_0; it is feasible to do since the consequent of Result 7.3 is strictly dependent on the nodes RS_0 and RC_0 — this means that Generalised Transitivity — Theorem 5.31 — can be used. Informally, Theorem 5.31 says that $c\langle 3..0\rangle + 2^1 d = ab\langle 0\rangle$.

$\models$ By Generalised Transitivity

$$\begin{array}{l} \langle\!| \, \texttt{Global}\,[(0,100)]\ ([A]=a \,\wedge\, [B]=b) \\ \quad \Longrightarrow \quad \texttt{Global}\,[(6,100)] \\ \qquad\qquad [RS_1] + 2^2[RC_1] = ab\langle 0\rangle + 2^1 ab\langle 1\rangle \,\wedge\, [RC_1]\langle 3\rangle = 0 \,|\!\rangle \end{array} \tag{7.4}$$

Now we have the output of stage 1 solely in terms of a and b. This can be rewritten into a more elegant form. The proving system has integer rewriting procedures which automatically rewrites $ab\langle n-1..0\rangle + 2^n ab\langle n\rangle$ as $ab\langle n..0\rangle$. Thus applying Lemma 5.24 and the rule of consequence, Theorem 5.18, yields the next result:

$\models$ By rule of consequence

$$\langle\!| \, \texttt{Global}\,[(0,100)]\ ([A]=a \wedge [B]=b) \Longrightarrow \texttt{Global}\,[(6,100)]\ [RS_1] + 2^2[RC_1] = ab\langle 1..0\rangle \wedge [RC_1]\langle 3\rangle = 0 \,|\!\rangle \tag{7.5}$$

Stages 2 and 3 The steps are exactly the same as stage 1.

$\models$ By STE

$$\langle\!| \, \texttt{Global}\,[(6,100)]\ [A]=a \wedge [B\langle 2\rangle]=b_2 \wedge [RS_1]=c\langle 4..0\rangle \wedge [RC_1]=d \wedge [RC_1]\langle 3\rangle = 0) \Longrightarrow \texttt{Global}\,[(9,100)]\ [RS_2] + 2^3[RC_2] = c\langle 4..0\rangle + 2^2 d + 2^2 ab\langle 2\rangle \wedge [RC_2]\langle 3\rangle = 0 \,|\!\rangle \tag{7.6}$$

$\models$ By Generalised Transitivity (Results 7.5 and 7.6)

$$\langle\!| \, \texttt{Global}\,[(0,100)]\ ([A]=a \wedge [B]=b) \Longrightarrow \texttt{Global}\,[(9,100)]\ [RS_2] + 2^3[RC_2] = ab\langle 1..0\rangle + 2^2 ab\langle 2\rangle \wedge [RC_2]\langle 3\rangle = 0 \,|\!\rangle \tag{7.7}$$

$\models$ By rule of consequence from Result 7.7

$$\langle\!| \, \texttt{Global}\,[(0,100)]\ ([A]=a \wedge [B]=b) \Longrightarrow \texttt{Global}\,[(9,100)]\ [RS_2] + 2^3[RC_2] = ab\langle 2..0\rangle \wedge [RC_2]\langle 3\rangle = 0 \,|\!\rangle \tag{7.8}$$

$\models$ By STE

$$\langle\!| \ \texttt{Global}\,[(9,100)] \quad [A] = a \wedge [B]\langle 3\rangle = b_3 \wedge [RS_2] = c\langle 5..0\rangle \wedge [RC_2] = d \wedge [RC_2]\langle 3\rangle = 0 \quad \Longrightarrow \quad \texttt{Global}\,[(12,100)] \quad [RS_3] + 2^4[RC_3] = c\langle 5..0\rangle + 2^3 d + 2^3 ab\langle 3\rangle \ \wedge [RC_3]\langle 3\rangle = 0 \ |\!\rangle \tag{7.9}$$

$\models$ By Generalised Transitivity (Results 7.8 and 7.9)

$$\langle\!| \ \texttt{Global}\,[(0,100)] \ ([A] = a \wedge [B] = b) \quad \Longrightarrow \quad \texttt{Global}\,[(12,100)] \quad [RS_3] + 2^4[RC_3] = ab\langle 2..0\rangle + 2^3 ab\langle 3\rangle \ \wedge [RC_3]\langle 3\rangle = 0 \ |\!\rangle \tag{7.10}$$

$\models$ By rule of consequence from Result 7.10

$$\langle\!| \ \texttt{Global}\,[(0,100)] \ ([A] = a \wedge [B] = b) \quad \Longrightarrow \quad \texttt{Global}\,[(12,100)] \quad [RS_3] + 2^4[RC_3] = ab \wedge [RC_3]\langle 3\rangle = 0 \ |\!\rangle \tag{7.11}$$

The adder stage The final step in the proof is to ensure that the last, adder stage, adds in the carries correctly. Here possible carries in the least significant bit must be passed to the most significant bit. For large bit widths, this adder stage may take tens or hundreds of nanoseconds, so timing may be important here.

$\models$ By STE

$$\langle\!| \ \texttt{Global}\,[(12,100)] \ ([RS_3] = c\langle 6..0\rangle \wedge [RC_3] = d \wedge [RC_3]\langle 3\rangle = 0) \quad \Longrightarrow \quad \texttt{Global}\,[(22,100)] \ ([RS_4] = (c\langle 6..0\rangle + 2^4 d)\langle 7..0\rangle) \ |\!\rangle \tag{7.12}$$

Now, using general transitivity, we have:

Bit width	**Number of gates**	**D Time (s)**	**T Time (s)**
4	135	3.9	5.4
8	473	9.8	15.0
16	1841	36.0	60.8
32	7265	168.7	371.4
64	28865	1081.9	> 6000

Table 7.5: Verification Times for Benchmark 17 Multiplier

$\models$ By Generalised Transitivity (Results 7.11 and 7.12)

$$\langle\!| \, \texttt{Global}\,[(0,100)]\ ([A]=a \wedge [B]=b) \tag{7.13}$$

$$\Longrightarrow\!\!\triangleright \quad \texttt{Global}\,[(22,100)]\ ([RS_4]=a*b) \, |\!\rangle$$

Again, the automatic rewrite systems recognises that ab is an eight bit number, and so rewrites $a*b\langle 7..0\rangle$ as $a*b$. This concludes the proof.

Appendix C has the FL proof script for the multiplier example.

Experimental results and comments This IFIP WG10.5 Benchmark 17 multiplier was verified for a number of bit widths (the n bit width case multiplies two n-bit numbers and produces a $2n$ bit number). The time taken to perform the verification on a DEC Alpha 3000 is shown in Table 7.5: the column labelled 'D Time' shows the time taken using the direct method, and the column labelled 'T Time' shows the time taken using the testing machine approach (all times shown in seconds). These results are useful for evaluating the testing machine approach, and are used in the discussion on testing machines in Section 7.4.4.

The proof script itself is short (less than 200 lines, about 50 of which are declarations) and straightforward to write, relying only on simple properties of integers. The full script can be found in Section C.4. Once structure in the circuit is identified by associating integers with collections of bit valued nodes, the verification no longer has to deal with bits, and at no stage does the verification have to concern itself with how the full adders or the base components are actually implemented.

The reason why STE cannot deal with the verification by itself is not because of the size of the circuit; the problem is that there is no good variable ordering for the multiplication of two bit vectors. However, good variable orderings are definitely possible for verifying the individual components of the multiplier with STE, and good heuristics to find good ordering can easily be automated.

7.3.4 Other Multiplier Verification

One of the main examples used in this thesis is the verification of a multiplier circuit. To put the thesis work in context, other work on multipliers is surveyed. Multipliers represent an important class of circuit, because arithmetic circuits are in themselves important, and because they are particularly challenging for BDD-based approaches.

Simonis uses a simple proof checker to verify a multiplier in [118]. The circuit description is represented in a Prolog-like language, and the correctness proof simulates a hand proof: nine correctness conditions are identified and checked (although it is not proved that these nine conditions imply that the multiplier works correctly). Each of the conditions is checked by a Prolog routine. Although the computational costs of verification were low, the correctness of the proof relies on the correctness of the nine conditions and the correctness of the Prolog routines. Timing is not checked.

Pierre presents the verification of the WG 10.5 multiplier in [108, 109]. The proof is done in the Boyer-Moore prover Nqthm. The work presented is not completely automated in that manual work is needed to translate the behavioural description from VHDL into the first-order logic used by Nqthm. The proof itself is based on a methodology supporting induction developed by Pierre for the verification of replicated structures. Provided certain design criteria are met, the proof can be automatically done by the system. Using replication and induction a general proof can be done for an n-bit multiplier rather than having to do individual proofs for individual bit-widths. Moreover, the approach is computationally efficient so duplicated work can be avoided.

The disadvantage of this approach is that it relies on the VHDL programs being written in a certain way. This is probably not too critical since the restrictions are not unreasonable. More seriously, timing issues are not dealt with. This may be a problem since while the functionality is independent of the bit width, timing is not. As timing is an important part of low-level verification, this approach needs further development.

Equivalence methods have also been used to verify multipliers. Van Eijk and Janssen use a BDD-based tool to show equivalence between different implementations of multipliers [30]. Their method relies on (automatically) finding structural and functional equivalences between different implementations of the circuit. For some circuits they get excellent experimental results. However, they too do not consider timing. Typically, one of the circuits is derived from the other through a number of design steps; thus, the confidence in the verification depends on the confidence on the correctness of the original circuit.

Although the compositional method proposed in this thesis relies on some structure of the circuit being identified, it is not necessary to decompose the circuit, or that clearly defined gross structure be determined. To be useful, it is only important to be able to identify circuit nodes with 'interesting values'; this makes it relatively robust to circuit optimisation.

An advantage of the compositional theory is that it incorporates a good model of time, which may be important in many applications. This advantage outweighs the disadvantage of having to verify circuit designs for each bit-width, which theorem proving approaches may obviate.

As discussed in Section 2.3.3, Kurshan and Lamport also explored combining theorem proving and model checking, and have applied their technique to verifying multipliers [93]. The work was not fully mechanised, and the implementation of the multiplier given at a high level.

However, although exploratory, their work suggested that combining different approaches would be successful.

7.4 Matrix Multiplier

A filter circuit based on a design of Mead and Conway is Benchmark 22 of the IFIP WG10.5 suite [100]. The filter is a matrix multiplication circuit for band matrices. A band matrix of band width w is a matrix in which zeros must be in certain positions (the matrices contain natural numbers), and the maximum number of non-zero items in a row or column is w. This circuit is called 2Syst. Section 7.4.1 discusses the specification of the circuit; Section 7.4.2 discusses its implementation; Section 7.4.3 presents its verification; and Section 7.4.4 analyses and comments on the verification in which a significant timing error was discovered. Sections 7.4.1 and 7.4.2 rely heavily on the benchmark documentation.[2]

7.4.1 Specification

The suite documentation does not give a general specification of the circuit (it does give a general implementation), but presents the case of $w = 4$. A circuit implemented for a band-width of w can be used to multiply matrices of any size — larger matrices just take longer to multiply; the documentation does not consider the general case, and gives only a specification for 4×4 matrices.

Let A and B be the two 4×4 matrices given below:

[2]The URL for the documentation is `ftp://goethe.ira.uka.de/pub/benchmarks/2Syst/`. This section is based on the documentation of this benchmark dated 16 November 1994. As a result of this research, the documentation has been revised, and the new version will be released shortly.

$$A = \begin{bmatrix} a_{11} & a_{12} & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 \\ a_{31} & a_{32} & a_{33} & a_{34} \\ 0 & a_{42} & a_{43} & a_{44} \end{bmatrix} \qquad B = \begin{bmatrix} b_{11} & b_{12} & b_{13} & 0 \\ b_{21} & b_{22} & b_{23} & b_{24} \\ 0 & b_{32} & b_{33} & b_{34} \\ 0 & 0 & b_{43} & b_{44} \end{bmatrix},$$

and let $C = A \times B$ be the matrix:

$$C = \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{bmatrix}$$

The external interface of the 2Syst circuit is shown in Figure 7.6. The coefficients of matrix A are input on the inputs `a0`, ... , `a3`, the coefficients of B are input on `b0`, ... , `b3`, and the coefficients of C, the result, is output on outputs `c0` to `c6`. (What this picture, taken from the documentation, does not show is that the circuit is clocked and there should be a pin for clock input too.)

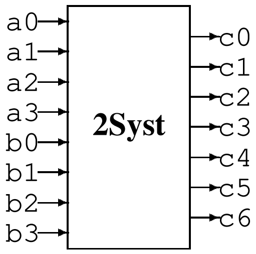


Figure 7.6: Black Box View of 2Syst

Timing

The timing of when and where the inputs must be applied and the outputs become available is critical. The timing for the inputs is presented in Table 7.6. In clock cycles 0 to 3, all the inputs are initialised by having zero applied to them. Then, for the next ten cycles the matrix coefficients are input to the circuit. For example, in cycle 9, the coefficients a_{23}, a_{42}, b_{32} and b_{42} are input on pins `a0`, `a3`, `b0`, and `b3` respectively, while all other pins have zero applied to them.

clock	**a0**	**a1**	**a2**	**a3**	**b0**	**b1**	**b2**	**b3**
$0-3$	0	0	0	0	0	0	0	0
4	0	a_{11}	0	0	0	b_{11}	0	0
5	0	0	a_{21}	0	0	0	b_{12}	0
6	a_{12}	0	0	a_{31}	b_{21}	0	0	b_{13}
7	0	a_{22}	0	0	0	b_{22}	0	0
8	0	0	a_{32}	0	0	0	b_{23}	0
9	a_{23}	0	0	a_{42}	b_{32}	0	0	b_{24}
10	0	a_{33}	0	0	0	b_{33}	0	0
11	0	0	a_{43}	0	0	0	b_{34}	0
12	a_{34}	0	0	0	b_{43}	0	0	0
13	0	a_{44}	0	0	0	b_{44}	0	0

Table 7.6: Inputs for the 2Syst Circuit

Table 7.7 shows when and where the coefficients of the output matrix can be found. The specification gives some freedom in timing here. It requires that the output be given in clock cycles $t_0, \ldots, t_6$, but does not specify values for the t_j; and, while $t_0 < t_1 \ldots < t_6$, the t_j need not be consecutive clock cycles. This gives some latitude in the implementation of the circuit.

7.4.2 Implementation

The matrix multiplication $C = A \times B$ can be defined in different ways. Assuming for simplicity that A and B are both $r \times r$ matrices, the usual definition of C is through defining each $c_{ij} =$

cycle	**c0**	**c1**	**c2**	**c3**	**c4**	**c5**	**c6**
t_0				c_{11}			
t_1			c_{12}		c_{21}		
t_2		c_{13}		c_{22}		c_{31}	
t_3	c_{14}		c_{23}		c_{32}		c_{41}
t_4		c_{24}		c_{33}		c_{42}	
t_5			c_{34}		c_{43}		
t_6				c_{44}			

Table 7.7: Outputs of the 2Syst Circuit

$\sum_{k=1}^{r} a_{ik}b_{ki}$. An alternative definition is useful in implementing parallel hardware to perform the multiplication: matrix multiplication can also be defined by the recursive equation 7.14.

$$\begin{aligned} c_{ij}^{(1)} &= 0 \\ c_{ij}^{(k+1)} &= c_{ij}^{(k)} + a_{ik}b_{kj} \\ c_{ij} &= c_{ij}^{(r+1)} \end{aligned} \tag{7.14}$$

The entries in arrays A and B are n-bit numbers. If the band-width of the matrices is w, the maximum number of non-zero terms in any c_{ij} is w, which means that each entry in c_{ij} is of bit-width $m = 2n + r - 1$.

The basic operation of Equation 7.14 is performing an addition and a multiplication; this is modelled in the implementation, where the basic cell has an integer multiplier and adder to perform this. The external interface of these cells is shown in Figure 7.7. The cell has three inputs: `C_In` is an m bit number, containing a partial sum; and `A_In` and `B_In` are n bit data which are either zero or coefficients of the A and B matrices. `A_Out`, `B_Out` are two n-bit output values and `C_Out` is an m-bit output. If in one clock cycle `A_In`, `B_In` and `C_In` have the values a, b and c respectively, then at the start of the next cycle: `A_Out` $= a$, `B_Out` $= b$, `C_Out` $= ab + c$.

Thus, the cell has two purposes: it acts as a one clock-cycle delay buffer for coefficients of the matrices (which are passed on to neighbouring cells), and performs the basic operation of

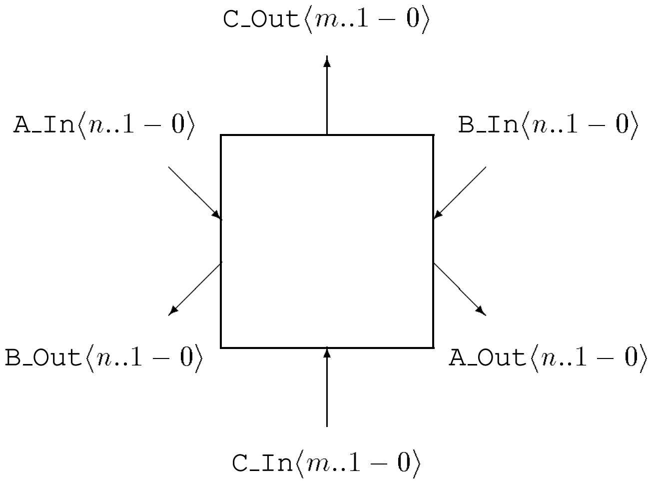


Figure 7.7: Cell Representation

an addition and multiplication.

Figure 7.8 shows how the cells are implemented. Each cell contains a multiplier, an adder, and three registers. The multiplier is the one discussed and verified in the previous section, and the adder is a conventional $2n$-bit adder. Each register has an input, an output, and a clock and select pin. By connecting the select and clock pins to the same global clock, the registers become positive-edge triggered: when the clock rises the value at the register's input is latched, output, and maintained until the clock rises again.

These cells are connected in a systolic array: each clock cycle cells performs an addition and multiplication and then passes its results to its neighbours for use in the next cycle. The cells are arranged as presented in Figure 7.9, and the timings given in Table 7.6 are designed so that cells get the right inputs at the right time. A simple example will illustrate how this works. To help the description, each cell in the systolic array has been labelled by $i\colon j$.

The circuit is implemented in Voss's EXE format as a detailed gate level description, using a unit delay model. The implementation is based on the VHDL program given in the benchmark suite documentation.

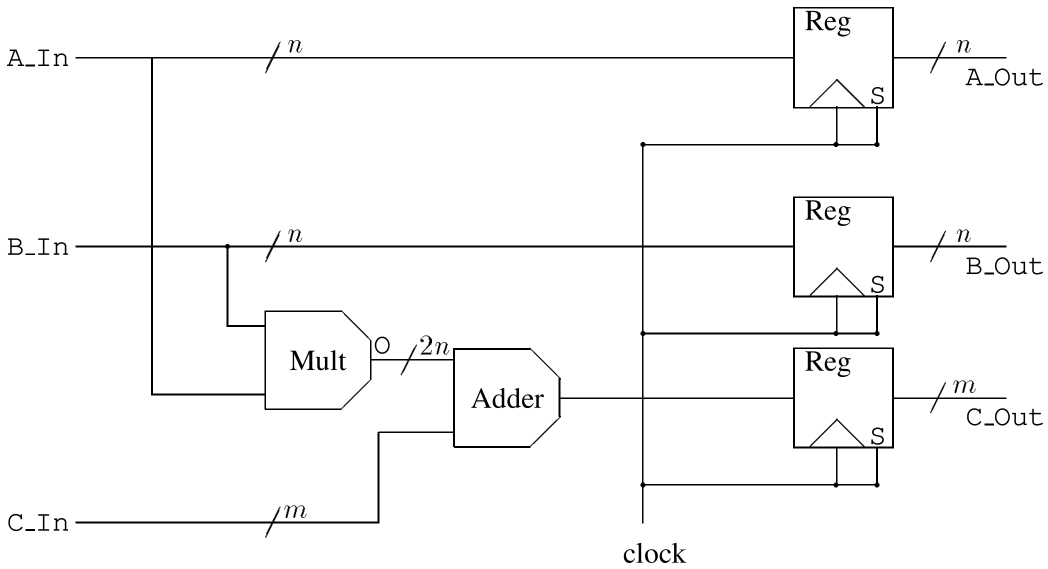


Figure 7.8: Implementation of Cell

Example 7.1.

Consider the computation of $c_{21} = a_{21}b_{11} + a_{22}b_{21}$. In the first three clock cycles the circuit is initialised so that at the start of the fourth cycle, all inputs have value zero.

Cycle 4: b_{11} is input on `b1` (input `B_In` of Cell 1:0). (a_{11} is also input in this cycle, but in the example, we only consider values contributing to c_{21}).

Cycle 5: Cell 1:0 will have passed b_{11} to its neighbour, so that b_{11} now becomes an input for Cell 1:1. a_{21} is input on `a2` (the `A_In` input of Cell 0:2).

Cycle 6: Cell 1:1 will have passed b_{11} to the `B_In` input of Cell 1:2, and Cell 0:2 will have passed a_{11} to the `A_In` input of Cell 1:2. At this stage, the `C_In` input of Cell 1:2 has the value 0. Cell 1:2 therefore computes $a_{11}b_{11}$. At the same time, b_{21} appears as input on `b0`, which is input `B_In` of Cell 0:0.

Cycle 7: Cell 1:2 will have passed $a_{11}b_{11}$ to Cell 0:1 as its `C_In` input. Cell 0:0 will have passed on b_{21} to Cell 0:1 as its `B_In` input. a_{22} appears on `a1`, which is the `A_In` input of Cell 0:1. Cell 0:1 computes $a_{11}b_{11} + a_{22}b_{21}$.

Cycle 8: Cell 0:1 outputs $a_{11}b_{11} + a_{22}b_{21}$ on its `C_Out` port (which is `c4`).

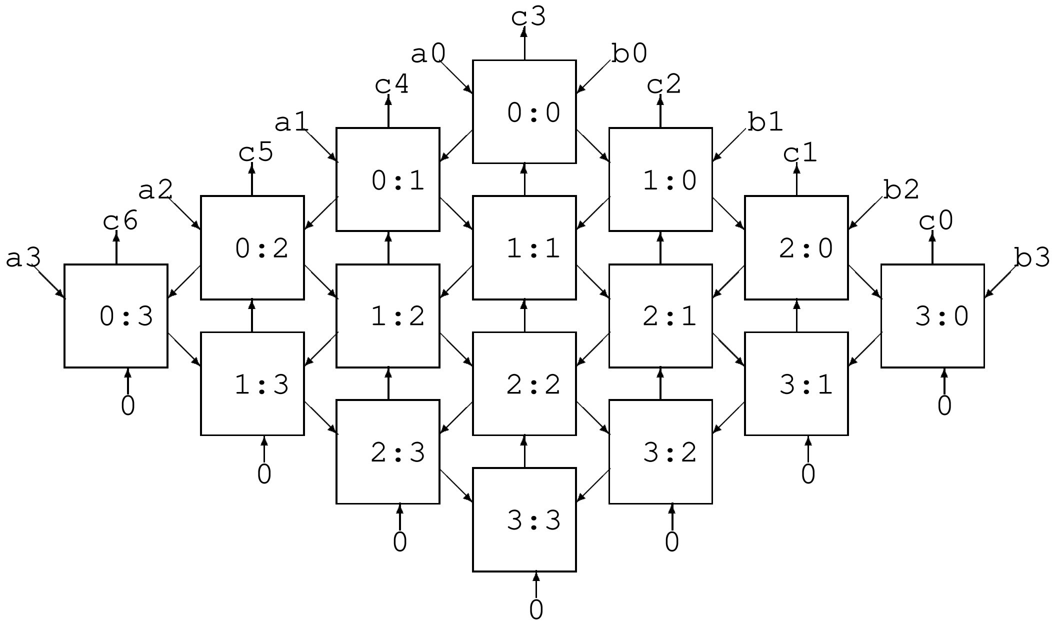


Figure 7.9: Systolic Array

7.4.3 Verification

The verification task can be divided into two parts, the verification of the individual components, and using the verification of the components to show that whole array is correct.

Verifying the Cells

The verification of a cell must show the multiplier, adder and registers all work correctly. Each cell must be verified individually. This section describes the verification of Cell $u{:}v$, and assumes for the sake of this exposition that the clock cycle is 200ns, and the bit-width is 4.

In the discussion below, the $\mathtt{A\_In}_{uv}$ and $\mathtt{B\_In}_{uv}$ are four-bit nodes, while all variables are 12 bit values. To simplify notation, in all the discussion below, a and b are short hands for $a\langle 3..0\rangle$ and $b\langle 3..0\rangle$ respectively.

It turns out that it useful to divide this proof into three parts:

- Given value a on $\mathtt{A\_In}_{uv}$, b on $\mathtt{B\_In}_{uv}$, and c on $\mathtt{C\_In}_{uv}$, one clock cycle later $a * b + c$ appears on $\mathtt{C\_Out}_{uv}$;
- Given value a on $\mathtt{A\_In}_{uv}$, one cycle later a appears on $\mathtt{A\_Out}_{uv}$; and
- Given value b on $\mathtt{B\_In}_{uv}$, one cycle later b appears on $\mathtt{B\_Out}_{uv}$.

When the cells are connected together, port $\mathtt{C\_In}_{uv}$ is connected to $\mathtt{C\_Out}_{(u+1)(v+1)}$, port $\mathtt{A\_Out}_{uv}$ is connected to $\mathtt{A\_In}_{u(v+1)}$, and $\mathtt{B\_Out}_{uv}$ is connected to $\mathtt{B\_In}_{(u+1)v}$. Therefore, the above verification conditions are rewritten as:

- Given value a on $\mathtt{A\_In}_{uv}$, b on $\mathtt{B\_In}_{uv}$, and c on $\mathtt{C\_Out}_{(u+1)(v+1)}$, one clock cycle later $a * b + c$ appears on $\mathtt{C\_Out}_{uv}$;
- Given value a on $\mathtt{A\_In}_{uv}$, one cycle later a appears on $\mathtt{A\_In}_{u(v+1)}$; and
- Given value b on $\mathtt{B\_In}_{uv}$, one cycle later b appears on $\mathtt{B\_In}_{(u+1)v}$.

Of course, it is possible to combine all three into one, stronger result. However, having three weaker results makes the proof more flexible since at some stages the proof needs only the weaker result, and using a stronger result would clutter things up and be more inefficient.

The costliest part of the proof is to show the multiplier works correctly. As Section 7.3.3 showed how the Benchmark 17 multiplier can be verified, for the purpose of this section, Result 7.15 is assumed (in the actual verification, the multiplier for each cell is reverified).

$\models$ By various rules

$$\langle\!| \, \texttt{Global}\,[(0,100)]\ \ ([\texttt{A\_In}_{uv}] = a \ \wedge\ [\texttt{B\_In}_{uv}] = b) \quad \Longrightarrow \quad \texttt{Global}\,[(22,100)]\ \ ([\texttt{O}_{uv}] = a * b)\, |\!\rangle \tag{7.15}$$

In the cell, the clock has an important effect; to include information of when clocking happens, the rule of consequence is often used to strengthen the antecedent of a result. For convenience, let

$$\begin{aligned} \mathit{Clock}_k \ &= \ \texttt{Global}\,[(200k, 200k+99), (200(k+1), 200(k+1)+99)]\ \ ([\texttt{clock}] = \mathbf{f})\ \wedge \\ &\qquad \texttt{Global}\,[(200k+100, 200k+199)]\ \ ([\texttt{clock}] = \mathbf{t}) \end{aligned}$$

which is the information about clocking which is needed in the proof of the k-th cycle. This formula says that the clock is low from time $200k$ to time $200k+99$, then high from time $200k+100$ to $200k+199$, and then low again from time $200k+200$ to $200k+299$.

Using this idea, Result 7.15 is transformed strengthening the antecedent, as well as taking into account the input on $\texttt{C\_In}$. Although, this is not useful for its own sake, it is useful in using the essence of Result 7.15.

$\models$ By Theorem 5.7

$$\langle\!| \, \texttt{Global}\,[(0,100)]\ \ [\texttt{A\_In}_{uv}] = a \ \wedge\ [\texttt{B\_In}_{uv}] = b \ \wedge\ [\texttt{C\_Out}_{(u+1)(v+1)}] = c \ \wedge\ \mathit{Clock}_0 \quad \Longrightarrow \quad \texttt{Global}\,[(22,100)]\ \ ([\texttt{O}_{uv}] = a * b)\, |\!\rangle \tag{7.16}$$

In the next step we show that the adder works correctly and that the output of the adder is latched for the appropriate time. This can be done with one trajectory evaluation. Note that the time interval in the consequent could be made bigger, but the one given suffices.

$$\models \text{ By STE}$$
$$\langle\!|\, \texttt{Global}\,[(22,100)]\;\; ([\texttt{O}_{uv}] = d\langle 7..0\rangle \;\wedge\; [\texttt{C\_Out}_{(u+1)(v+1)}] = c \;\wedge\; \mathit{Clock}_0) \quad\quad (7.17)$$
$$\Longrightarrow \quad \texttt{Global}\,[(200,300)]\;\; ([\texttt{C\_Out}_{uv}] = c + d\langle 7..0\rangle)\,|\!\rangle$$

Results 7.16 and 7.17 are now combined by specialising the latter result (substituting ab for d), and using transitivity. Note that this is just a special case of General Transitivity (Theorem 5.31).

$$\models \text{ By Theorem 5.31}$$
$$\langle\!|\, \texttt{Global}\,[(0,100)]\;\; ([\texttt{A\_In}_{uv}] = a \;\wedge\; [\texttt{B\_In}_{uv}] = b \;\wedge\; [\texttt{C\_Out}_{(u+1)(v+1)}] = c \;\wedge\; \mathit{Clock}_0) \quad\quad (7.18)$$
$$\Longrightarrow \quad \texttt{Global}\,[(200,300)]\;\; ([\texttt{C\_Out}_{uv}] = c + a * b)\,|\!\rangle$$

Result 7.18 is the core result that has to be proved about the cell. The next two results show that the cell also acts as one cycle delay buffers for values of the A and B matrices. Both of these results can easily be done using STE alone.

$$\models \text{ By STE}$$
$$\langle\!|\, \texttt{Global}\,[(0,100)]\;\; ([\texttt{A\_In}_{uv}] = a \;\wedge\; \mathit{Clock}_0) \quad\quad (7.19)$$
$$\Longrightarrow \quad \texttt{Global}\,[(200,300)]\;\; ([\texttt{A\_In}_{u(v+1)}] = a)\,|\!\rangle$$

$$\models \text{ By STE}$$
$$\langle\!|\, \texttt{Global}\,[(0,100)]\;\; ([\texttt{B\_In}_{uv}] = b \;\wedge\; \mathit{Clock}_0) \quad\quad (7.20)$$
$$\Longrightarrow \quad \texttt{Global}\,[(200,300)]\;\; ([\texttt{B\_In}_{(u+1)v}] = b)\,|\!\rangle$$

Overall Verification

Once each of the cells has been individually verified, the proofs about the individual cells must be combined to prove that the systolic array as a whole works correctly.

The proof is modelled on how the systolic array computes its results; in its development the proof traces the behaviour of the circuit as it uses its inputs, computes results, and outputs the answers.

Consider the operation of one cell, Cell $u{:}v$. It has three *input* neighbours from which it gets values (the boundary cells are special cases and easily taken care of):

- Cell $u{:}(v-1)$, its A-left-neighbour from which it gets a value of the A matrix,
- Cell $(u-1){:}v$, its B-right-neighbour from which it gets a value of the B matrix, and
- Cell $(u+1){:}(v+1)$ its C-down-neighbour from which it gets a partial sum;

and three *output* neighbours to which it gives values:

- Cell $u{:}(v+1)$, its A-right-neighbour, to which it gives a value of the A matrix,
- Cell $(u+1){:}v$, its B-left-neighbour, to which it gives a value of the B matrix, and
- Cell $(u-1){:}(v-1)$ its C-up-neighbour, to which it gives a partial sum;

At the beginning of clock cycle k, none, some, or all of the following will be known about Cell $u{:}v$'s input neighbours (recall that a clock cycle is 200 time units long), where the I_j are antecedent TL formulas, and the θ_x are integer expressions:

$$\models \langle\!|\, I_1 \Longrightarrow \texttt{Global}\,[(200k, 200k+100)]\;\; [\texttt{A\_In}_{uv}] = \theta_a \,|\!\rangle \tag{7.21}$$

$$\models \langle\!|\, I_2 \Longrightarrow \texttt{Global}\,[(200k, 200k+100)]\;\; [\texttt{B\_In}_{uv}] = \theta_b \,|\!\rangle \tag{7.22}$$

$$\models \langle\!|\, I_1 \Longrightarrow \texttt{Global}\,[(200k, 200k+100)]\;\; [\texttt{C\_Out}_{(u+1)(v+1)}] = \theta_c \,|\!\rangle \tag{7.23}$$

If all three results are known, then we use conjunction on Results 7.21–7.23, and introduce new clocking information. For convenience, let

$$I_4 = I_1 \wedge I_2 \wedge I_3 \wedge \mathit{Clock}_k.$$

This is the conjunction of I_1, I_2 and I_3 and contains necessary clocking information for the k-th cycle. Then we have:

$$\models \text{ By Conjunction and Rule of Consequence}$$
$$\langle\!|\, I_4 \Longrightarrow \texttt{Global}\,[(200k, 200k+100)]\; [\texttt{A\_In}_{uv}] = \theta_a \wedge [\texttt{B\_In}_{uv}] = \theta_b \wedge [\texttt{C\_Out}_{uv}] = \theta_c. \,|\!\rangle \tag{7.24}$$

Then Result 7.18 is time-shifted forward by k-clock cycles to get:

$$\models \text{ By time-shifting}$$
$$\langle\!|\, \texttt{Global}\,[(200k, 200k+100)]\; ([\texttt{A\_In}_{uv}] = a \wedge [\texttt{B\_In}_{uv}] = b \wedge [\texttt{C\_Out}_{(u+1)(v+1)}] = c \wedge \mathit{Clock}_k) \Longrightarrow \texttt{Global}\,[(200(k+1), 200(k+1)+100)]\; ([\texttt{C\_Out}_{uv}] = c + a * b) \,|\!\rangle \tag{7.25}$$

Using General Transitivity on Results 7.24 and 7.25 leads to:

$$\models \text{ By Theorem 5.31}$$
$$\langle\!|\, I_4 \Longrightarrow \texttt{Global}\,[(200(k+1), 200(k+1)+100)]\; ([\texttt{C\_Out}_{uv}] = \theta_c + \theta_a * \theta_b) \,|\!\rangle \tag{7.26}$$

This is a proof of what Cell $u{:}v$ computes in the k-th cycle. In proving what happens in the $(k+1)$-th cycle, Result 7.26 is used in the proof of the behaviour of Cell $(u+1){:}(v+1)$, which is Cell $u{:}v$'s up-C-neighbour.

Similarly, if Result 7.21 is known, then precondition strengthening is used to introduce new clocking information to get:

$$\models \text{ By Theorem 5.7}$$
$$\langle\!|\, I_1 \wedge \mathit{Clock}_k \Longrightarrow \texttt{Global}\,[(200k, 200k+100)]\; [\texttt{A\_In}_{uv}] = \theta_a \,|\!\rangle \tag{7.27}$$

Then Result 7.19 is time-shifted by k clock cycles to get:

$\models$ By STE

$$\langle\!|\, \texttt{Global}\,[(200k, 200k+100)] \;\; ([\texttt{A\_In}_{uv}] = a) \;\wedge\; \mathit{Clock}_k \qquad (7.28)$$
$$\Longrightarrow \quad \texttt{Global}\,[(200(k+1), 200(k+1)+100)] \;\; ([\texttt{A\_In}_{u(v+1)}] = a)\,|\!\rangle$$

General Transitivity between Results 7.27 and 7.28 then yields:

$\models$ By Theorem 5.7

$$\langle\!|\, I_1 \;\wedge\; \mathit{Clock}_k \qquad (7.29)$$
$$\Longrightarrow \quad \texttt{Global}\,[(200(k+1), 200(k+1)+100)] \;\; ([\texttt{A\_In}_{u(v+1)}] = \theta_a)\,|\!\rangle$$

This shows what Cell $u{:}v$ passes to its A-right neighbour at the end of the k-th cycle, and this result will be used to prove properties of Cell $u:(v+1)$ in the $(k+1)$-th cycle. A similar result shows that in the k-th Cell $u{:}v$ also passes on the value input on its `B_In` port,

$\models$ By various rules

$$\langle\!|\, I_2 \;\wedge\; \mathit{Clock}_k \qquad (7.30)$$
$$\Longrightarrow \quad \texttt{Global}\,[(200(k+1), 200(k+1)+100] \;\; ([\texttt{B\_In}_{(u+1)v)}] = \theta_b)\,|\!\rangle$$

FL Proof script The FL proof script that performs the proof uses the approach outlined above. First, the behaviour of each cell is individually verified. Then, the proof proceeds by proving properties of the circuit in each clock cycle.

A two dimensional array of proofs is kept: at the start of the k-th cycle, the array's (u, v) entry contains proofs of what the output of Cell $u{:}v$ input neighbour's are at the end of the $(k-1)$-th cycle. The proof then uses this information to infer as much as possible about the output of Cell $u{:}v$ at the end of the k-th cycle, and this information is then used to update the array of proofs so that Cell $u{:}v$'s output neighbours can use this information in the $(k+1)$-th cycle.

Start of cycle	c_0 Cell 3:0	c_1 Cell 2:0	c_2 Cell 1:0	c_3 Cell 0:0	c_4 Cell 0:1	c_5 Cell 0:2	c_4 Cell 0:3
7				c_{11}			
8			c_{12}		c_{21}		
9		c_{13}				c_{31}	
10	c_{14}			c_{22}			c_{41}
11			c_{23}		c_{32}		
12		c_{24}				c_{42}	
13				c_{33}			
14			c_{34}		c_{43}		
15							
16				c_{44}			

Table 7.8: Benchmark 22: Actual Output Times

7.4.4 Analysis and Comments

The FL proof script uses STE and the inference rules to prove what the output of the circuit is at different stages – this is summarised in Table 7.8.

Comparison between Tables 7.7 and 7.8 shows that even given the ability for the designer to choose the values of $t_1, \ldots, t_6$, the implementation does not meet the specification.

There are two possibilities. The easier and probably better solution would be to change the specification, in accordance with the results shown in Table 7.8. However, another solution would be to place one cycle delay buffers on the outputs `c_0`, `c_1`, `c_5` and `c_6`; the amount of extra circuitry is small, would not slow down the circuit, and would lead to a more elegant specification.

The proof script, including the proof of the correctness of all the multipliers and declarations, is approximately 500 lines long, of which about 100 lines are declarations. The proof script can be found in Section C.5. The program itself is straightforward, although the use of a two dimensional array does not show off a functional, interpreted language at its best. The complete verification of a 4×4 systolic array of 32 bit multipliers (roughly 110 000 gates) takes just over 10 hours of CPU time on a DEC Alpha 3000 using the testing machine approach, and just under three hours using the direct method.

This verification uses the testing machine algorithm for STE, showing the weakness of using testing machines. The data structure needed to represent the model of the circuit is approximately 4M in size, making composition of circuit and testing machines difficult. While other implementations of machine composition are possible, the sheer size of the circuits remains an inherent problem. A similar problem can be seen in the verification of the multiplier (Table 7.5). Since both the size of the circuitry and the number of trajectory evaluations is quadratic in the bit-width, if every time trajectory evaluation must be done, circuit composition must be too, the resulting algorithm will be at least quartic. This explains why the verification of large bit widths becomes so expensive for testing machines.

The second part of the verification — showing that when connected together the multipliers produce the correct answer — is essentially performing symbolic simulation. Zhu and Seger have shown that given a set of trajectory assertion results, there is a weakest machine which satisfies these assertions [130]; this weakest machine is a conservative approximation of the circuit as any assertion that is true of the approximation is also true of the circuit.

This suggests an alternative verification methodology. The verification of the correctness of each of the multipliers extracts the essence of the behaviour of the circuit. From these assertions it should be possible to automatically generate a conservative approximation of the entire systolic array. This representation of this approximation would not use BDDs; in fact it would be at a higher level of abstraction. STE could then be used on the verification of the entire systolic array.

7.5 Single Pulser

This example shows how the fundamental compositional theory introduced in Chapter 5 can be built on; particularly through the use of induction on time, composite, problem-specific inference rules can be developed.

7.5.1 The Problem

Johnson has used the Single Pulser — a textbook example circuit — to study different verification methods [88]. The original problem statement for the circuit is:

> We have a debounced pushbutton, on (true) in the down position, off (false) in the up position. Devise a circuit to sense the depression of the button and assert an output signal for one clock pulse. The system should not allow additional assertions of the output until after the output has released the button.

Johnson reformulates this into:

- the pulser emits a single unit-time pulse on its output for each pulse received on i,
- there is exactly one output pulse for every input pulse, and
- the output pulse is in the neighbourhood of the input.

Figure 7.10 illustrates the external interface of the pulser. The port `In` is the button to be pressed (if it has the value `H`, the button is pressed, if `L` then it is not), and `Out` is the output.

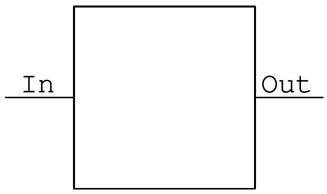


Figure 7.10: Single Pulser

Johnson presents the verification of this circuit in a number of different systems. This section attempts the verification using the compositional theory of STE. This attempt is not as general as some of Johnson's approaches since the specification is very specific about the timing of the output with relation to the button being pressed.

7.5.2 An Example Composite Compositional Rule

The motivation for the lemma below is that the essence of the behaviour of the pulser can be described by three assertions that show how the pulser reacts immediately to stimulation. By using induction over time, these results can be combined and generalised.

Lemma 7.1.

Let s, t, and u be arbitrary integers such that $0 \leq s \leq t < u$. Suppose:

1. $\models \langle\!\langle\, \neg g_1 \Longrightarrow\!\!\!\triangleright \texttt{Next}\, h_1 \,\rangle\!\rangle$,
2. $\models \langle\!\langle\, (\neg g_1 \,\wedge\, \texttt{Next}\, g_1) \Longrightarrow\!\!\!\triangleright (\texttt{Next}^2 h_2) \,\rangle\!\rangle$, and
3. $\models \langle\!\langle\, g_1 \Longrightarrow\!\!\!\triangleright \texttt{Next}^2 h_1 \,\rangle\!\rangle$;

then

1. $\models \langle\!\langle\, \texttt{Global}\,[(s,t)]\;(\neg g_1) \Longrightarrow\!\!\!\triangleright \texttt{Global}\,[(s+1,t+1)]\;h_1 \,\rangle\!\rangle$.
2. $\models \langle\!\langle\, (\texttt{Global}\,[(s,t)]\;(\neg g_1) \,\wedge\, \texttt{Global}\,[(t+1,u)]\;g_1)$

$$\Longrightarrow\!\!\!\triangleright$$

$$(\texttt{Global}\,[(s+1,t+1)]\;h_1) \wedge (\texttt{Next}^{(t+2)} h_2) \wedge (\texttt{Global}\,[(t+3,u+2)]\;h_2) \,\rangle\!\rangle$$

Proof. The proof of 1 comes straight from Corollary 5.23. For 2, let s, t, and u be arbitrary natural numbers such that $s \leq t < u$.

(1) $\models \langle\!\langle\, \texttt{Global}\,[(s,t)]\;(\neg g_1) \Longrightarrow \texttt{Global}\,[(s+1,t+1)]\; h_1 \,\rangle\!\rangle$

From hypothesis (1) by Lemma 5.22

(2) $\models \langle\!\langle\, \texttt{Next}^{t}(\neg g_1) \wedge \texttt{Next}^{(t+1)} g_1 \Longrightarrow \texttt{Next}^{(t+2)} h_2 \,\rangle\!\rangle$

Time-shifting hypothesis (2)

(3) $\models \langle\!\langle\, \texttt{Global}\,[t+1,u)]\; g_1 \Longrightarrow \texttt{Global}\,[(t+3,u+2)]\; h_1 \,\rangle\!\rangle$

From hypothesis 3, by Lemma 5.22

(4) $\models \langle\!\langle (\texttt{Global}\,[(s,t)]\;(\neg g_1) \wedge \texttt{Global}\,[(t+1,u)]\; g_1)$

$\Longrightarrow (\texttt{Global}\,[(s+1,t+1)]\; h_1 \wedge \texttt{Next}^{(t+2)} h_2 \wedge$

$\texttt{Global}\,[(t+3,u+2)]\; h_2)\rangle\!\rangle$

Conjunction of (1), (2), (3)

7.5.3 Application to Single Pulser

Given a candidate circuit, it should be possible to use STE to verify the following three properties:

1. $\models \langle\!\langle\, (\neg[\texttt{In}]) \Longrightarrow \texttt{Next}\,(\neg[\texttt{Out}]) \,\rangle\!\rangle$;
2. $\models \langle\!\langle\, (\neg[\texttt{In}] \wedge \texttt{Next}\,[\texttt{In}]) \Longrightarrow (\texttt{Next}^2[\texttt{Out}]) \,\rangle\!\rangle$, and
3. $\models \langle\!\langle\, [\texttt{In}] \Longrightarrow \texttt{Next}^2(\neg[\texttt{Out}]) \,\rangle\!\rangle$.

Using these results, the above lemma can be invoked to show that

1. $\models \langle\!\langle\, \texttt{Global}\,[(s,t)]\;(\neg[\texttt{In}]) \Longrightarrow \texttt{Global}\,[(s+1,t+1)]\; \neg[\texttt{Out}] \,\rangle\!\rangle$, and
2. $\models \langle\!\langle\, (\texttt{Global}\,[(s,t)]\;(\neg[\texttt{In}]) \wedge \texttt{Global}\,[(t+1,u)]\;[\texttt{In}])$

$$\Longrightarrow$$

$$(\texttt{Global}\,[(s+1,t+1)]\;(\neg[\texttt{Out}]) \wedge \texttt{Next}^{(t+2)}[\texttt{Out}] \wedge$$

$$\texttt{Global}\,[(t+3,u+2)]\;(\neg[\texttt{Out}]))\rangle\!\rangle$$

The first result says that if the input does not go high (the button is not pushed), then the output does not go high. The second result says when the button is pushed (input goes from low to high), the output goes high for exactly one pulse and then goes low and stays low at least as long as the button is still pushed.

I argue that these two properties capture the intuitive specification of Johnson. However, the specification is more restrictive; there are valid implementations that satisfy Johnson's specification which would not pass this specification, showing the limitations of our current methods. It is possible to give a more general specification based on Johnson's SMV specification[3], but currently there are not efficient model checking algorithms for these specification.

7.6 Evaluation

The experiments reported in this chapter showed that the compositional theory can be successfully implemented in a combined theorem prover-trajectory evaluation system, thereby enabling circuits with extremely large state spaces to be fully verified with reasonable human and computational costs. The following table summarises the examples verified (in the size column, n refers to the bit-width).

Description of circuit	How verified	Approx. size (gates)
Simple comparator	STE/Compositional Theory	$O(n^2)$
Hidden weighted bit	STE/Compositional Theory	$O(n^2)$
Carry-save adder	STE	200
B8ZS encoder	STE	75
IEEE floating point multiplier	STE/Compositional Theory	33 000
Simple 64-bit multiplier	STE/Compositional Theory	25 000
Benchmark 17 multiplier	STE/Compositional Theory	28 000
Benchmark 22 systolic array	STE/Compositional Theory	115 000

[3]Note that although the timing constraints in the SMV specification are more general, this SMV specification is also implementation dependent — in particular, it requires some knowledge of the internal structure of the implementation, which this proof does not.

In using the verification system, a key issue is the user interface to the system. Both the STE and the other inference rules are provided in one common, integrated framework. This not only makes it easier for the human verifier to use, but reduces the chance of error. Providing STE as an inference rule for the theorem prover to use proved useful. The ability to use FL as a script language was extremely important for increasing flexibility and ease of use.

The method of data representation proved to be very successful. It allowed BDDs to be used where appropriate, and other representations where BDDs are inappropriate. Decision procedures and other domain knowledge are critical for the success of the approach.

The results presented show that the increased expressiveness of TL not only allows a richer set of properties to be expressed, but can make specification cleaner too.

This chapter also shows that all three extensions to STE are feasible and can be applied successfully. However, both the testing machines and the mapping method have significant drawbacks in different circumstances.

Testing machines are not appropriate to use when the circuit being verified is very large, and when a number of trajectory evaluations will be run requiring different testing machines. Although the cost of automatically constructing testing machines is reasonable, the overhead of performing circuit composition repeatedly can be very large. On the other hand, once the new circuit is constructed, trajectory evaluation is efficient, and therefore the method may be appropriate where only a few trajectory evaluations will be done, and where the consequents are complicated.

The mapping method suffers from the need to introduce extra boolean variables. This is particularly the case when wishing to show that a state predicate holds for a sequence of states, where although the individual states are different, the relationship between state components stays constant. For example, we may wish to show that for a sequence of n states, at any time exactly one of m of the state's components have a H value. Using the mapping method would require the introduction of nm variables. A different example is the B8ZS verification, where we wish to show that too many zeros do not appear consecutively. The testing machine and direct methods require no new boolean variables; the mapping method would require two new boolean variables for each time step.

Chapter 8

Conclusion

Verification of large circuits is feasible using the appropriate logical framework. Chapters 3, 4 and 5 presented such a framework. Chapters 6 and 7 showed how this theory can be successfully implemented and illustrated the method by verifying a number of circuits. A summary of the research findings is given in Section 8.1, and some issues for future research is given in Section 8.2.

8.1 Summary of Research Findings

8.1.1 Lattice-based Models and the Quaternary logic $\mathcal{Q}$

The motivation of model checking is to use a logic to describe properties of the model of the system under study, and to verify the behaviour of the model by checking whether the properties (written as logic formulas) are satisfied by the model. The key questions are: how the model is represented; which logic is used; and how satisfaction is checked.

Using a lattice model structure has significant advantages for automatic model checking. By using a partial order to represent an information ordering, much larger state spaces can be modelled directly than with more traditional representation schemes. Previous work described earlier showed the advantage of this method of model representation.

This information ordering has a direct effect on what can be known about the model. A two-valued propositional logic is too crude a tool to use — it must conflate lack of knowledge with falseness. This is not only wrong in principle; the technical properties of a two-valued logic make it impossible to support negation fully.

The quaternary logic $\mathcal{Q}$ is suitable for describing the state of lattice-based models since it can describe systems with incomplete or inconsistent information. This makes it possible to distinguish clearly between truth and inconsistency, and falseness and incomplete information. Moreover, it supports a much richer temporal logic.

On the whole, the use of $\mathcal{Q}$ has been very successful. However, there are some minor points which need some attention. As discussed in Chapter 3 the definition of $\mathcal{Q}$ given in Table 3.1 on page 49 is not the only one possible. For example, in the definition given here, $\perp \vee \top = \mathtt{t}$. This definition is not without its problems — although it does have the advantage of very efficient implementation, it complicates some of the proofs and, notwithstanding the usual intuitive motivation, seems difficult to justify in the context of a temporal logic. $\perp \vee \top = \top$, would seem to be a better definition. In order to keep monotonicity constraints this would necessitate defining $\mathtt{t} \vee \top = \top$ too. These redefinitions would mean that disjunction in $\mathcal{Q}$ would not be the meet with respect to the truth ordering of $\mathcal{Q}$. Which would be the better definition is not clear; more theoretical and practical work must be done.

8.1.2 The Temporal Logic TL

$\mathcal{Q}$ can only describe the instantaneous state of a model. The temporal logic TL uses $\mathcal{Q}$ as its base, and can describe the evolving behaviour of the model over time. Note that the choice of $\mathcal{Q}$ as the base of the temporal logic leaves much freedom in choosing the temporal operators of a temporal logic, and other temporal logics could be built on top of $\mathcal{Q}$.

Previous temporal logics proposed for model checking partially ordered state space could not be as expressive as TL because they were based on a two valued logic. In particular TL supports negation and disjunction fully. In the examples explored in this thesis, the expressiveness of the logic was quite sufficient (the problems encountered with some of the verifications were caused by limitations in the shortcomings of the model checking algorithms). Nevertheless, whether introducing new temporal operators is worthwhile is an interesting one, especially if the model structure were extended (see Section 8.2.1).

8.1.3 Symbolic Trajectory Evaluation

STE has been used successfully in the past for model checking partially ordered state spaces. However, previous work only supported a restricted temporal logic. The thesis showed that the theory of STE could be generalised to deal with the whole of TL, and a number of practical algorithms were proposed for model checking a significant subset of TL. In particular, the four-valued logic of $\mathcal{Q}$ proved a good technical framework for STE-based algorithms.

8.1.4 Compositional Theory

The increase in expressiveness makes the need to overcome the performance bottlenecks of model checking more alluring and more important computationally. One of the primary contributions of the research is the development of a sound compositional theory for STE-based model checking using TL formulas. A set of sound inference rules can be used to deduce results: the base rule uses STE to verify a property of a model; the other rules can be used to combine properties previously proved.

At a practical level, the compositional theory can be used to implement a hybrid verification system that uses both theorem proving and model checking for verification. BDD-based model checking algorithms are extremely effective in proving many properties. However, there are inherent computational limits in what these methods can do; by using a theorem prover which implements the compositional theory, these limits can be overcome to a great extent. By providing automatic assistance, increasing the level of abstraction, and, most importantly, by providing a powerful and flexible user interface to the theorem prover (through FL), the task of the human verifier using the theorem prover can be made easier.

Features of this approach are:

- An appropriate verification methodology can be applied at the appropriate level — model checking at the low level, theorem proving at a higher level.
- STE supports a good model of time. This makes it suitable to verify not only functional correctness, but many timing properties.
- In the verification, although the implementation is given at a low level (e.g. at the gate or switch level), the correctness specification (*viz.* the TL formulas used) is, through the use of data abstraction, at a fairly high level.
- User intervention is necessary. Low level verification through STE, and important heuristics in the theorem proving component are important in alleviating the burden the verifier might otherwise encounter.

To illustrate the effectiveness of the approach, a number of circuits were completely verified. The largest of these circuits is one of the circuits in the IFIP WG10.5 Benchmark suite and contains over 100 000 gates. A serious timing error was discovered in the verification. This experimental work showed that increasing the expressiveness of the temporal logics that STE supports not only means that more properties can be expressed, but that through the use of the compositional theory, is computationally feasible.

8.2 Future Research

The research has raised a number of research issues, and left some questions only partially answered.

8.2.1 Non-determinism

The lattice structure of the state space means that although the next state function is deterministic, non-determinism can be implicitly represented through the use of X values. Although suitable for dealing with non-deterministic behaviour of inputs of circuit models, this treatment of non-determinism is not very sophisticated. One avenue of research would be to investigate the possibility of incorporating non-determinism explicitly within the model structure by replacing the next state function $\mathbf{Y}$ with a next state relation. Whether the semantics would be linear or branching time needs exploration, although I conjecture that a branching time semantics would be more suitable. Trees, rather than sequences or trajectories, would be used to model behaviour (and properties verified using symbolic trajectree evaluation). This would clearly raise the issue of the expressiveness of TL, and the need for operators that express path switching.

8.2.2 Completeness and Model Synthesis

The work of Zhu and Seger [130] showed that the compositionality theory for trajectory formulas [78] with minor modification is complete in the following sense. If K is a set of assertions, there is a weakest model $\mathcal{M}$ such that each assertion in K holds of the model. Moreover, any assertion that is true of $\mathcal{M}$ can be derived from K using the compositional theory. Whether the same thing is true of the compositional theory for TL needs further investigation.

This question is important from a practical point of view. Being able to construct such a weakest model from a set of assertions can be very useful for specification validation. It can also be used for verification, as discussed in Section 7.4.4, where a possible verification strategy for the verification of systolic array multiplier was outlined. After proving that the individual base modules of the circuit work correctly, it should be possible to construct a model (the extracted model) of the circuit from the set of assertions proved of the base modules; these assertions extract out the essential behaviour of the circuit. Then, the overall behaviour of the circuit can be verified by performing STE on the extracted model.

This raises the question of how to execute temporal logics efficiently, which involves interesting theoretical and practical questions (see [57] for an introduction). The key in making this efficient is, I conjecture, that the appropriate data structures should be used for representing the extracted model. In particular, given that BDDs are a very good representation of bit-level descriptions of the circuit, it is unlikely that using a BDD representation for the extracted model will gain significant improvement in performance, and for a multiplier circuit, it will certainly fail. Rather, the extracted model should be used as a method for finding a higher-level description of the circuit. For example, in the case of the array multiplier, an integer level description would be suitable. Even using a non-canonical representation of integers would allow STE to be accomplished in this particular case. What is important is that it should be easy to apply domain information to the problem. Note that from a practical point of view, it may not be necessary for the compositional theory to be complete, provided that all, or most, interesting properties can be derived. If the compositional theory is not complete, then the usefulness of this approach must be determined experimentally.

8.2.3 Improving STE Algorithms

Although the STE-based algorithms presented here were shown to be effective, they are not capable of model checking all assertions. There are two major aspects that need research.

- *Enriching the antecedent*

  So far the STE-based algorithms require that the antecedents be trajectory formulas. Although the use of the compositional theory ameliorates this restriction, it would be desirable to support richer antecedents. The key question is how sets of sequences can efficiently be represented. Through the introduction of fresh boolean variables it is possible to represent the union of two sets, thereby increasing the types of formulas for which relatively simple representations exist for their defining sequence sets. How efficiently can this be implemented? Are there alternative representation techniques?

- *Supporting the infinite temporal operator*

  At present there are no general algorithms for supporting the until operator and the derived infinite temporal operators. To do this requires not only an efficient way to represent a set of states, but also efficient methods of performing operations such as set union and comparison. STE uses parametric representation of state, which allows extremely large state spaces to be represented. This representation does not yet support efficient set manipulation operations. Thus, an important research question is how these operations can be implemented efficiently.

8.2.4 Other Model Checking Algorithms

This leads on to the question of whether model checking algorithms other than those based on symbolic trajectory evaluation would be effective. It appears that adapting the traditional BDD-style model checking algorithms such as those described in [26] to deal with partially-ordered state spaces would be possible. The logical framework developed here — $\mathcal{Q}$, TL and the various satisfaction relations — would form the basis of such adaptation. The research question is how these model checking algorithms could be adapted to make use of partial information in an effective way. Particularly if extended to deal with non-determinism, an advantage of these model checking algorithms is that they would support model checking of properties requiring more expressive formulas than those of the style of verification supported by STE.

8.2.5 Tool Development

The prototypes developed in the course of this research have showed that efficient, usable tools can be developed to support the compositional theory. The key components are supporting powerful, easy use of domain knowledge, and the provision of a flexible user interface through FL. Although the prototypes were successful, they were prototypes and contained a number of *ad hoc* features. Not only is a cleaner implementation required, but there are some issues which need further attention.

- Forward or backward style of proof. The prototypes used the forward style of proof, whereas Seger's VossProver used the backward style of proof. While I believe that the forward style of proof is more appropriate for hardware verifications using this approach, the issue is not clear.

- Incorporating new domain knowledge. The use of decision procedures and the incorporation of domain knowledge in other ways (e.g. through decision procedures) is important. Standard packages for types such as bit vectors and integers must be provided, and it would be desirable to have a clean way for users to integrate new theories or extend old ones.

- Partial automation of theorem proving. Although using STE for much of the verification alleviates much of the tedium traditionally associated with low-level of verification using theorem provers, it is desirable to automate as much as possible. The use of heuristics for finding time-shifts and specialisations needs to be extended.

- Debugging facilities. When errors are detected it is important that meaningful error messages be provided. One issue is relating higher-level concepts (e.g. an equation involving integers) to lower-level concepts (e.g. values on bit-valued nodes). Another issue is intelligent intervention when errors occur — determining what information is needed for the user to correct the proof and presenting it in a meaningful way. This is a general lesson for verification systems [105].

Epilogue

Verification is a central theoretical and practical problem of computer science, and much research is being done on different facets of the problem.

Systems with very large state spaces pose a particular challenge for verification, especially when a detailed account of timing is important. For these types of state space, partial order representations can be very effective. The three major contributions of this thesis have been:

- Developing a suitable theoretical framework for a temporal logic used to describe the behaviour of finite state systems with lattice-structured state spaces;
- Extending symbolic trajectory evaluation techniques to provide effective model checking for an important class of assertions about these systems; and
- Developing and implementing a compositional theory for model checking, which allows the successful integration of theorem proving and automatic model checking approaches in a practical tool that can successfully verify large circuits.

Bibliography

[1] A. Arnold and S. Brlek. Automatic Verification of Properties in Transition Systems. *Software — Practice and Experience*, 25(6):579–596, June 1995.

[2] M. Aagaard and C.-J.H. Seger. The Formal Verification of a Pipelined Double-Precision IEEE Floating-Point Multiplier. In *ACM/IEEE International Conference on Computer-Aided Design*, pages 7–10, November 1995.

[3] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–172, January 1993.

[4] Advanced Micro Devices. *PAL Device Handbook*. Advanced Micro Devices, Inc., 1988.

[5] H.R. Andersen, C. Stirling, and G. Winskel. A Compositional Proof System for the Modal μ-calculus. In *Proceedings of the 9th Annual Symposium on Logic in Computer Science*, June 1994.

[6] A. Aziz, T.R. Shiple, V. Singhal, and A.L. Sangiovanni-Vincentelli. Formula-Dependent Equivalence for Compositional CTL Model Checking. In Dill [48], pages 324–337.

[7] R.C. Backhouse. *Program Construction and Verification*. Prentice-Hall, London, 1986.

[8] D.L. Beatty. *A Methodology for Formal Hardware Verification with Application to Microprocessors*. PhD thesis, Carnegie-Mellon University, School of Computer Science, 1993.

[9] I. Beer, S. Ben-David, D. Geist, R. Gewirtzman, and M. Yoeli. Methodology and system for practical formal verification of reactive hardware. In Dill [48], pages 182–193.

[10] N.D. Belnap. A useful four-valued logic. In J.M. Dunn and G. Epstein, editors, *Modern Uses of Multiple Valued Logic*. D. Reidel, Dordrecht, 1977.

[11] S.A. Berezine. Model checking in μ-calculus for distributed systems. *Technical Paper*, Department of Mathematics, Novosibirsk State University, 1994.

[12] O. Bernholtz and O. Grumberg. Buy One, Get One Free !!! In Gabbay and Ohlbach [61], pages 210–224.

[13] M.A. Bezem and J. Groote. A Correctness Proof of a One-bit Sliding Window Protocol in μCRC. *The Computer Journal*, 37(4):289–307, 1994.

[14] B. Boigelot and P. Wolper. Symbolic Verification with Periodic Sets. In Dill [48], pages 55–67.

[15] T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[16] S. Bose and A. Fisher. Automatic verification of synchronous circuits using symbolic logic simulation and temporal logic. In Claesen [33], pages 151–158.

[17] R.S. Boyer and J.S. Moore. *A Computational Logic Handbook*. Academic Press, 1988.

[18] J.C. Bradfield. A Proof Assistant for Symbolic Model-Checking. In G. von Bochmann and D.K. Probst, editors, *CAV '92: Proceedings of the Fourth International Workshop on Computer Aided Verification*, Lecture Notes in Computer Science 663, pages 316–329, Berlin, 1992. Springer-Verlag.

[19] J.C. Bradfield. *Verifying Temporal Properties of Systems*. Birkhäuser, Boston, 1992.

[20] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A Theory of Communicating Sequential Processes. *Journal of the Association for Computing Machinery*, 31(3):560–599, July 1984.

[21] R.E. Bryant. On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication. *IEEE Transactions on Computers*, 40(2):205–213, February 1991.

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

[23] R.E. Bryant, D.L. Beatty, and C.-J. H. Seger. Formal Hardware Verification by Symbolic Ternary Trajectory Evaluation. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pages 397–407. 1991.

[24] R.E. Bryant and Y.-A. Chen. Verification of Arithmetic Functions with Binary Moment Diagrams. Technical Report CMU-CS-94-160, School of Computer Science, Carnegie Mellon University, May 1994.

[25] R.E. Bryant and C.-J. H. Seger. Formal Verification of Digital Circuits by Symbolic Ternary System Models. In E.M Clarke and R.P. Kurshan, editors, *Proceedings of Computer-Aided Verification '90*, pages 121–146. American Mathematical Society, 1991.

[26] J.R Burch, E.M. Clarke, D.E. Long, K.L. McMillan, and D.L. Dill. Symbolic Model Checking for Sequential Circuit Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, April 1994.

[27] J.R. Burch, E.M. Clarke, and K.L. McMillan. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, June 1992.

[28] J.R. Burch and D.L. Dill. Automatic Verification of Pipelined Microprocessor Control. In Dill [48], pages 68–80.

[29] O. Burkart and B. Steffen. Model Checking for Context-Free Processes. In W.R. Cleaveland, editor, *CONCUR '92: Proceedings of the Third International Conference on Concurrency Theory*, Lecture Notes in Computer Science 630, pages 123–137, Berlin, 1992. Springer-Verlag.

[30] C.A.J. van Eijk and G.L.J.M. Janssen. Exploiting Structural Similarities in a BDD-based Verification Method. In Kumar and Kropf [92], pages 110–125.

[31] L. Carroll. *Through the Looking Glass and what Alice saw there*. Macmillan and Co., London, 1894.

[32] S. Christensen, H. Hüttel, and C. Stirling. Bisimulation Equivalence is Decidable for all Context-Free Processes. Technical Report ECS-LFCS-92-218, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, June 1992.

[33] L.J.M. Claesen, editor. *Proceedings of the IFIP WG 10.2/WG 10.5 International Workshop on Applied Formal Methods for Correct VLSI Design* (November 1989), Amsterdam, 1990. North-Holland.

[34] E. Clarke, M. Fujita, and X. Zhao. Hybrid Decision Diagrams: Overcoming the Limitations of MTBDDs and BMDs. Technical Report CMU-CS-95-159, School of Computer Science, Carnegie Mellon University, April 1995.

[35] E. Clarke and X. Zhao. Word Level Symbolic Model Checking: A New Approach for Verifying Arithmetic Circuits. Technical Report CMU-CS-95-161, School of Computer Science, Carnegie Mellon University, May 1995.

[36] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[37] E.M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In Courcoubetis [45], pages 450–462.

[38] E.M. Clarke, O. Grumberg, and K. Hamaguchi. Another Look at LTL Model Checking. In Dill [48], pages 415–427.

[39] E.M. Clarke, O. Grumberg, and D.E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5), September 1994.

[40] E.M. Clarke, D.E. Long, and K.L. McMillan. Compositional Model Checking. In *IEEE Fourth Annual Symposium on Logic in Computer Science*, Washington, D.C., 1989. IEEE Computer Society.

[41] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A Semantics-Based Tool for the Verification of Concurrent Systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[42] A. Cohn. The Notion of Proof in Hardware Verification. *Journal of Automated Reasoning*, 5(2):127–139, June 1989.

[43] O. Coudert, C. Berthet, and J.C. Madre. Verification of Sequential Machines Using Boolean Functional Vectors. In Claesen [33], pages 179–195.

[44] O. Coudert and J.C. Madre. The Implicit Set Paradigm: A New Approach to Finite State System Verification. *Formal Methods in System Design*, 6(2):133–145, March 1995.

[45] C. Courcoubetis, editor. *Proceedings of the 5th International Conference on Computer-Aided Verification*, Lecture Notes in Computer Science 697, Berlin, July 1993. Springer-Verlag.

[46] B. Cousin and J. Hélary. Performance Improvements of State Space Exploration by Regular and Differential Hashing Functions. In Dill [48], pages 364–376.

[47] M. Darwish. Formal Verification of a 32-Bit Pipelined RISC Processor. MASc Thesis, University of British Columbia, Department of Electrical Engineering, 1994.

[48] D.I. Dill, editor. *CAV '94: Proceedings of the Sixth International Conference on Computer Aided Verification*, Lecture Notes in Computer Science 818, Berlin, June 1994. Springer-Verlag.

[49] J. Dingel and T. Filkorn. Model checking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving. In Wolper [128], pages 55–69.

[50] M. Donat. Verification Using Abstract Domains. Unpublished paper, Department of Computer Science, University of British Columbia, April 1993.

[51] A. Conan Doyle. *The Adventures of Sherlock Holmes*. Oxford University Press, Oxford, 1993.

[52] E.A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B (Formal Models and Semantics), chapter 16, pages 995–1072. Elsevier, Amsterdam, 1990.

[53] E.A. Emerson and J.Y. Halpern. "Sometimes" and "Not Never" Revisited. *Journal of the Association for Computing Machinery*, 33(1):151–178, January 1986.

[54] R. Enders, T. Filkorn, and D. Taubner. Generating BDDs for Symbolic Model Checking in CCS. In K.G. Larsen and A. Skou, editors, *CAV '91: Proceedings of the Third International Conference on Computer Aided Verification*, Lecture Notes in Computer Science 571, pages 203–213, Berlin, June 1991. Springer-Verlag.

[55] J. Esparza. On the decidability of model checking for several μ-calculi and Petri nets. Technical Report ECS-LFCS-93-274, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, July 1993.

[56] J.C. Fernandez, A. Kerbrat, and L. Mounier. Symbolic Equivalence Checking. In Courcoubetis [45], pages 85–96.

[57] M. Fisher and R. Owens. An Introduction to Executable Modal and Temporal Logics. In M. Fisher and R. Owens, editors, *IJCAI' 93 Workshop: Executable Modal and Temporal Logics*, Lecture Notes in Artificial Intelligence 897, pages 1–39, Berlin, Aug 1993.

[58] M. Fitting. Bilattices and the Theory of Truth. *Journal of Philosophical Logic*, 18(3):225–256, August 1989.

[59] M. Fitting. Bilattices and the Semantics of Logic Programming. *The Journal of Logic Programming*, 11(2):91–116, August 1991.

[60] K. Frenkel. An Interview with Robin Milner. *Communications of the ACM*, 36(1):90–97, January 1993.

[61] D.M. Gabbay and H.J. Ohlbach, editors. *ICTL' 94: Proceedings of the First International Conference on Temporal Logic*, Lecture Notes in Artificial Intelligence 827, Berlin, Aug 1994.

[62] A. Galton, editor. *Temporal Logics and their Applications*. Academic Press, London, 1987.

[63] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York, 1979.

[64] Globe and Mail. Intel finds 'subtle flaw' in chip. *The Globe and Mail*, 25 November 1994, p. B16, November 1994.

[65] Globe and Mail. Intel takes charge. *The Globe and Mail*, 18 January 1995, p. B2, February 1995.

[66] D. Goldberg. Computer arithmetic. In *Computer Architecture: a quantitative approach* by J.L. Hennessy and D.A. Patterson, chapter Appendix A, pages A0–A65. Morgan Kaufmann, San Mateo, California, 1990.

[67] M.J.C. Gordon. *Programming Language Theory and its Implementation*. Prentice-Hall, London, 1988.

[68] M.J.C. Gordon, editor. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, Cambridge, 1993.

[69] M.J.C. Gordon, C.P. Wadsworth, and R. Milner. *Edinburgh LCF : a mechanised logic of computation*. Lecture Notes in Computer Science 78. Springer-Verlag, Berlin, 1979.

[70] S. Graf and C. Loiseaux. A tool for symbolic program verification and abstraction. In Courcoubetis [45], pages 71–84.

[71] O. Grumberg and R.P. Kurhsan. How Linear Can Branching-time Be? In Gabbay and Ohlbach [61], pages 180–194.

[72] O. Grumberg and D.E. Long. Model Checking and Modular Verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.

[73] A. Gupta. Formal hardware verification methods: A survey. *Formal Methods in System Design*, 1(2/3):151–238, October 1992.

[74] N.A. Harman and J.V. Tucker. Algebraic models and the correctness of microprocessors. In Milne and Pierre [101], pages 92–108.

[75] J. Harrison. Binary decision diagrams as a HOL derived rule. *The Computer Journal*, 38(2):162–170, 1995.

[76] S. Hazelhurst and C.-J. H. Seger. A Simple Theorem Prover Based on Symbolic Trajectory Evaluation and OBDDs. Technical Report 93-41, Department of Computer Science, University of British Columbia, November 1993. Available by anonymous ftp as ftp://ftp.cs.ubc.ca/pub/local/techreports/1993/TR-93-41.ps.gz.

[77] S. Hazelhurst and C.-J. H. Seger. Composing symbolic trajectory evaluation results. In Dill [48], pages 273–285.

[78] S. Hazelhurst and C.-J.H. Seger. A Simple Theorem Prover Based on Symbolic Trajectory Evaluation and BDD's. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(4):413–422, April 1995.

[79] P. Heath. *The Philosopher's Alice*. Academy Editions, London, 1974.

[80] M. Hennessy. *Algebraic Theory of Processes*. MIT Press, Cambridge, MA., 1988.

[81] M. Hennessy and R. Milner. Algebraic Laws for Non-determinism and Concurrency. *Journal of the Association for Computing Machinery*, 32(1):137–161, January 1985.

[82] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 583, October 1969.

[83] R. Hojati and R.K. Brayton. Automatic Datapath Abstraction in Hardware Systems. In Wolper [128], pages 98–113.

[84] H. Hungar. Combining Model Checking and Theorem Proving to Verify Parallel Processes. In Courcoubetis [45], pages 154–165.

[85] H. Hungar and B. Steffen. Local Model Checking for Context Free Processes. In A. Lingas, R. Karlsson, and S. Carlsson, editors, *Proceedings of the 20th International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science 700, pages 593–605, Berlin, July 1993. Springer-Verlag.

[86] W.A. Hunt. *FM8501: A Verified Microprocessor*. Lecture Notes in Artificial Intelligence 795. Springer-Verlag, Berlin, 1994.

[87] P. Jain and G. Gopalakrishnan. Efficient symbolic simulation-based verification using the parametric form of boolean expressions. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(8):1005–1015, August 1994.

[88] S.D. Johnson, P.S. Miner, and A. Camilleri. Studies of the Single Pulser in Various Reasoning Systems. In Kumar and Kropf [92], pages 126–145.

[89] B. Jonsson. Compositional specification and verification of distributed systems. *ACM Transactions on Programming Languages and Systems*, 16(2):259–303, March 1994.

[90] J.J. Joyce and C.-J.H. Seger. Linking BDD-based Symbolic Evaluation to Interactive Theorem-Proving. In *Proceedings of the 30th Design Automation Conference*. IEEE Computer Society Press, June 1993.

[91] T. Kropf. Benchmark-Circuits for Hardware-Verification. In Kumar and Kropf [92], pages 1–12.

[92] R. Kumar and T. Kropf, editors. *TPCD'94: Proceedings of the Second International Conference on Theorem Provers in Circuit Design*, Lecture Notes in Computer Science 901, Berlin, September 1994. Springer-Verlag.

[93] R.P. Kurshan and L. Lamport. Verification of a Multiplier: 64 Bits and Beyond. In Courcoubetis [45], pages 166–179.

[94] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3), May 1994.

[95] D.E. Long. *Model Checking, Abstraction, and Compositional Verification*. PhD thesis, Carnegie-Mellon University, School of Computer Science, July 1993. Technical report CMU-CS-93-178.

[96] K. Marzullo, F.B. Schneider, and J. Dehn. Refinement for Fault-Tolerance: An Aircraft Hand-off Protocol. Technical Report 94-1417, Department of Computer Science, Cornell University, April 1994.

[97] M.C. McFarland. Formal Verification of Sequential Hardware: A Tutorial. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(5):633–654, May 1993.

[98] K.L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Carnegie-Mellon University, School of Computer Science, 1993.

[99] K.L. McMillan. Hierarchical representations of discrete functions, with applications to model checking. In Dill [48], pages 41–54.

[100] C. Mead and L. Conway. *Introduction to VLSI Design*. Addison-Wesley, Reading, Massachusetts, 1980.

[101] G.J. Milne and L. Pierre, editors. *CHARME '93: IFIP WG10.2 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, Lecture Notes in Computer Science 683, Berlin, May 1993. Springer-Verlag.

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

[103] F. Moller and S.A. Smolka. On the Computational Complexity of Bisimulation. *ACM Computing Surveys*, 27(2):287–289, June 1995.

[104] New Scientist. Flawed chips bug angry users. *New Scientist*, (1955):18, December 1994.

[105] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[106] S. Owre, J.M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction — CADE-11*, Lecture Notes in Computer Science 607, pages 748–752, Berlin, March 1992. Springer-Verlag.

[107] L.C. Paulson. *ML for the working programmer*. Cambridge University Press, New York, 1991.

[108] L. Pierre. VHDL Description and Formal Verification of Systolic Multipliers. In D. Agnew, L. Claesen, and R. Camposano, editors, *Computer hardware description languages and their applications : proceedings of the 11th IFIP WG10.2 International Conference on Computer Hardware Description Languages and their Applications - CHDL'93*, number 32 in IFIP Transactions A, pages 225–242, Berlin, 1993. Springer-Verlag.

[109] L. Pierre. An automatic generalization method for the inductive proof of replicated and parallel structures. In Kumar and Kropf [92], pages 72–91.

[110] D. Price. Pentium FDIV Flaw – lessons learned. *IEEE Micro*, 15(12):88–86, April 1995.

[111] S. Rajan, N. Shankar, and M.K. Srivas. An Integration of Model Checking with Automated Proof Checking. In Wolper [128], pages 84–97.

[112] J.M. Rushby and F. von Henke. Formal Verification Algorithms for Critical Systems. *IEEE Transactions on Software Engineering*, 19(1), January 1993.

[113] M. Ryan and M. Sadler. Valuation Systems and Consequence Relations. In S. Abramsky, D.M. Gabbay, and T.S. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 1 (Background: Mathematical Structures), chapter 1, pages 1–78. Clarendon Press, Oxford, 1992.

[114] T. Ralston S. Gerhart, D. Craigen. Experience with Formal Methods in Critical Systems. *IEEE Software*, 11(1):21–28, June 1994.

[115] C.-J.H. Seger. Voss — A Formal Hardware Verification System User's Guide. Technical Report 93-45, Department of Computer Science, University of British Columbia, November 1993. Available by anonymous ftp as ftp://ftp.cs.ubc.ca/pub/local/techreports/1993/TR-93-45.ps.gz.

[116] C.-J.H. Seger and R.E. Bryant. Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories. *Formal Methods in Systems Design*, 6:147–189, March 1995.

[117] C.-J.H. Seger and J.J. Joyce. A Mathematically Precise Two-Level Hardware Verification Methodology. Technical Report 92-34, Department of Computer Science, University of British Columbia, December 1992.

[118] H. Simonis. Formal verification of multipliers. In Claesen [33], pages 267–286.

[119] V. Stavridou. Formal methods and VLSI engineering practice. *The Computer Journal*, 37(2), 1994.

[120] C. Stirling. Modal and temporal logics. In S. Abramsky, D.M. Gabbay, and T.S. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2 (Background: Computational Structures), pages 477–563. Clarendon Press, Oxford, 1992.

[121] C. Stirling. Modal and Temporal Logics for Processes. Technical Report ECS-LFCS-92-221, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, June 1992.

[122] C. Stirling and D. Walker. Local model checking in the modal mu-calculus. *Theoretical Computer Science*, 89(1):161–177, October 1991.

[123] P.A. Subrahmanyam. Towards Verifying Large(r) Systems: A strategy and an experiment. In Milne and Pierre [101], pages 135–154.

[124] A.M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society* (Second Series), 42:230–265, 1937.

[125] A. Visser. Four Valued Semantics and the Liar. *Journal of Philosophical Logic*, 13(2):181–212, May 1984.

[126] O. Wilde. *The Importance of being Earnest : a trivial comedy for serious people*. Vanguard Press, New York, 1987.

[127] G. Winskel. A note on model checking the modal ν-calculus. In G. Ausiello, M. Dezani-Ciancagnlini, and S. Ronchi Della Rocca, editors, *Proceedings of the 16th International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science 372, pages 761–771, Berlin, 1989. Springer-Verlag.

[128] P. Wolper, editor. *CAV '95: Proceedings of the Seventh International Conference on Computer Aided Verification*, Lecture Notes in Computer Science 939, Berlin, July 1995. Springer-Verlag.

[129] Z. Zhu. A Compositional Circuit Model and Verification by Composition. In Kumar and Kropf [92], pages 92–109.

[130] Z. Zhu and C.-J.H. Seger. The Completeness of a Hardware Inference System. In Dill [48], pages 286–298.

Appendix A

Proofs

A.1 Proof of Properties of TL

A.1.1 Proof of Lemma 3.3

Lemma A.1 (Lemma 3.3).

If $g, h \colon \mathcal{S} \to \mathcal{Q}$ are simple, then $D(g) = D(h)$ implies that $\forall s \in \mathcal{S}, g(s) = h(s)$.

Proof. To emphasise that $D(g) = D(h)$, we set $D = D(g)$. Let $s \in \mathcal{S}$. Let $E = \{q \in \mathcal{Q} : (s_q, q) \in D \wedge s_q \sqsubseteq s\}$ and let $e = \sqcup E$. The proof first shows that $g(s) = e$.

(a) $g(s) \preceq e$

(1)	$\exists s_p \in \mathcal{S} \ni (s_p, g(s)) \in D$	g is simple.
(2)	$s_p \sqsubseteq s$	Definition of defining pair.
(3)	$g(s) \in E$	Definition of E, (1), (2).
(4)	$g(s) \preceq \sqcup E$	Definition of join.

(b) $e \preceq g(s)$

(1)	$\forall (s_q, q) \in D, q \preceq g(s)$	$q = g(s_q)$,$s_q \sqsubseteq s$, g is monotone.
(2)	$g(s)$ is an upperbound of E	(1)
(3)	$\sqcup E \preceq g(s)$	Property of join.

Thus $g(s) = e$.

Similarly, $h(s) = e$.

Therefore, $g(s) = h(s)$. As s was arbitrary $\forall s \in \mathcal{S}, g(s) = h(s)$. $\square$

Note that the proof does not rely on the particular structure of $\mathcal{Q}$; it only relies on $\mathcal{Q}$ being a complete lattice.

A.1.2 Proof of Theorem 3.5

The idea behind the proof is to partition the domain of an arbitrary $p : \mathcal{S} \to \mathcal{Q}$ depending on the value of $p(s)$. Then, we construct a function which enables us to determine which partition an element falls in. We can now use this information in reverse — once we know which partition an element falls into, we can return the value of the function for that element. The complication of the proof is to use the properties of $\mathcal{Q}$ to combine all this information together. As an analogy suppose that $g : \mathcal{S} \to \{-10, 10\}$. Suppose we know that $g_-(s) = 1$ if $g(s) = -10$ and $g_-(s) = 0$ otherwise; and $g_+(s) = 1$ if $g(s) = 10$ and $g_+(s) = 0$ otherwise. Then we can write $g(s) = -10g_-(s) + 10g_+(s)$. The two steps in doing this were to find the g_- and g_+ functions, and then to determine how to combine them. The proof of Theorem 3.5 follows a similar pattern: first the functions that are the equivalent of g_- and g_+ are given; after that it is shown that these functions can be combined to simulate p, and that they can be constructed from simple predicates.

The functions given in the next definition are the analogues to the g_- and g_+ functions.

Definition A.1.
Suppose that we have an arbitrary monotonic function $p : \mathcal{S} \to \mathcal{Q}$. Define the following:

$$\chi_{\mathbf{f}}(u) = \begin{cases} \top & \exists s \in \mathcal{S}, s \sqsubseteq u, p(s) = \mathbf{f} \\ \mathbf{t} & \text{otherwise.} \end{cases}$$

$$\chi_{\mathbf{t}}(u) = \begin{cases} \mathbf{t} & \exists s \in \mathcal{S}, s \sqsubseteq u, p(s) = \mathbf{t} \\ \bot & \text{otherwise.} \end{cases}$$

$$\chi_{\top}(u) = \begin{cases} \top & \exists s \in \mathcal{S}, s \sqsubseteq u, p(s) = \top \\ \mathbf{t} & \text{otherwise.} \end{cases}$$

□

The proof of Theorem 3.5 comes in two parts: first, Lemma A.2 demonstrates how to combine the χ_q functions to construct a function p' that is equivalent to p; and Theorem A.3 concludes by showing that the functions $\chi_{\mathbf{f}}$, $\chi_{\mathbf{t}}$ and $\chi_{\top}$ can be defined from the simple predicates using TL operators.

Lemma A.2.

Let $p : \mathcal{S} \to \mathcal{Q}$ be a monotonic predicate. Define $p'(u)$ by

$$p'(u) = \chi_{\mathbf{t}}(u) \wedge \chi_{\mathbf{f}}(u) \wedge \chi_{\top}(u) \ \vee \ \neg\chi_{\top}(u).$$

Then, $\forall s \in \mathcal{S}, p(s) = p'(s)$.

Proof.

(a) Suppose $p(u) = \perp$

(1)	$\chi_{\mathbf{f}}(u) = \chi_{\top}(u) = \mathbf{t}$, $\chi_{\mathbf{t}}(u) = \perp$	By definition and monotonicity of p.
(2)	$p'(u) = \mathbf{t} \wedge \perp \wedge \mathbf{t} \ \vee \ \mathbf{f} = \perp = p(u)$	(1)

(b) Suppose $p(u) = \mathbf{f}$

(1)	$\chi_{\mathbf{f}}(u) = \top$, $\chi_{\mathbf{t}}(u) = \perp$, $\chi_{\top}(u) = \mathbf{t}$	By definition and monotonicity of p.
(2)	$p'(u) = \perp \wedge \top \wedge \mathbf{t} \ \vee \ \mathbf{f} = \mathbf{f} = p(u)$	(1)

(c) Suppose $p(u) = \mathbf{t}$

(1)	$\chi_{\mathbf{t}}(u) = \mathbf{t}$, $\chi_{\mathbf{f}}(u) = \chi_{\top}(u) = \mathbf{t}$.	By definition and monotonicity of p.
(2)	$p'(u) = \mathbf{t} \wedge \mathbf{t} \wedge \mathbf{t} \ \vee \ \mathbf{f} = \mathbf{t} = p(u)$	(1)

(d) Suppose $p(\top) = \top$

(1)	$\chi_\top(u) = \top,\ \perp \preceq \chi_{\mathbf{t}}(u),$ $\mathbf{t} \preceq \chi_{\mathbf{f}}(u)$	By definition and monotonicity of p.
(2)	$p'(u) \succeq \perp \wedge \mathbf{t} \wedge \top \vee \top$ $= \mathbf{f} \vee \top = \top$ $= p(u)$	(1) and monotonicity of $\wedge$ and $\vee$.
(3)	$p'(u) = p(u).$	Since $\top = \sqcup \mathcal{Q}$.

□

The final part of the proof is to show that the χ_q functions can be constructed from the simple predicates.

Theorem A.3 (Theorem 3.5).

For all monotonic predicates $p : \mathcal{S} \to \mathcal{Q}$, $\exists p' \in$ TL such that $\forall s \in \mathcal{S}$, $p(s) = p'(s)$.

Proof.

Partition $\mathcal{S}$ according to the value of p:

$$S_\perp = \{s \in \mathcal{S} : p(s) = \perp\} \qquad S_{\mathbf{f}} = \{s \in \mathcal{S} : p(s) = \mathbf{f}\}$$
$$S_{\mathbf{t}} = \{s \in \mathcal{S} : p(s) = \mathbf{t}\} \qquad S_\top = \{s \in \mathcal{S} : p(s) = \top\}.$$

Some of these sets may be empty. Now, for each $s \in \mathcal{S}$ we define χ'_s: $\mathcal{S} \to \mathcal{Q}$ (each χ'_s characterises all elements at least as big as s) as follows:

$$\chi'_s(u) = \begin{cases} \mathbf{t} & s \sqsubseteq t \\ \perp & s \not\sqsubseteq t \end{cases}$$

Note that each χ'_s is simple. For the purpose of this lemma, define $\vee \emptyset = \perp$. The χ' have the following two properties.

Suppose $\exists s \in \mathcal{S} \ni s \sqsubseteq u, p(s) = q$ *for some* $q \in \mathcal{Q}$.

(1) $\chi'_s(u) = \mathbf{t}$ Definition of χ'.

(2) $\bigvee \{\chi'_v(u) \,:\, v \in S_q\} = \mathbf{t}$ $s \in S_q$ by supposition.

Suppose $\nexists s \in \mathcal{S} \ni s \sqsubseteq u, p(s) = q$ *for some* $q \in \mathcal{Q}$.

(1) $\forall v \in S_q, v \not\sqsubseteq u$ Supposition.

(2) $\bigvee \{\chi'_v(u) \,:\, v \in S_q\} = \perp$ Either S_q is empty or follows from (1).

Now, define:

$$\chi_{\mathbf{f}}(u) = \neg(\bigvee \{\chi'_s(u) \,:\, s \in S_{\mathbf{f}}\}) \vee \top$$

$$\chi_{\mathbf{t}}(u) = \bigvee \{\chi'_s(u) \,:\, s \in S_{\mathbf{t}}\}$$

$$\chi_{\top}(u) = \neg(\bigvee \{\chi'_s(u) \,:\, s \in S_{\top}\}) \vee \top$$

Using the properties of χ' proved above, we have that:

$$\chi_{\mathbf{f}}(u) = \begin{cases} \top & \exists s \in \mathcal{S}, s \sqsubseteq u, p(s) = \mathbf{f} \\ \mathbf{t} & \text{otherwise.} \end{cases}$$

$$\chi_{\mathbf{t}}(u) = \begin{cases} \mathbf{t} & \exists s \in \mathcal{S}, s \sqsubseteq u, p(s) = \mathbf{t} \\ \perp & \text{otherwise.} \end{cases}$$

$$\chi_{\top}(u) = \begin{cases} \top & \exists s \in \mathcal{S}, s \sqsubseteq u, p(s) = \top \\ \mathbf{t} & \text{otherwise.} \end{cases}$$

Note that we have constructed from simple predicates the functions χ_s given in Definition A.1. Thus, by Lemma A.2, given an arbitrary monotonic predicate p, we are able to define it from simple predicates using conjunction, disjunction and negation – showing we can consider any monotonic state predicate as a short-hand for a formula of TL. $\square$

A.1.3 Proof of Lemma 3.6

Lemma A.4 (Lemma 3.6).

1. *Commutativity:*

$g_1 \wedge g_2 \ \equiv \ g_2 \wedge g_1,\ g_1 \vee g_2 \ \equiv \ g_2 \vee g_1.$

2. *Associativity:*

$(g_1 \vee g_2) \vee g_3 \ \equiv \ g_1 \vee (g_2 \vee g_3),\ (g_1 \wedge g_2) \wedge g_3 \ \equiv \ g_1 \wedge (g_2 \wedge g_3)$

3. *De Morgan's Law:*

$g_1 \wedge g_2 \ \equiv \ \neg(\neg g_1 \vee \neg g_2), g_1 \vee g_2 \ \equiv \ \neg(\neg g_1 \wedge \neg g_2).$

4. *Distributivity of* $\wedge$ *and* $\vee$:

$h \wedge (g_1 \vee g_2) \ \equiv \ (h \wedge g_1) \vee (h \wedge g_2),\ h \vee (g_1 \wedge g_2) \ \equiv \ (h \vee g_1) \wedge (h \vee g_2)$

5. *Distributivity of* `Next` *:*

$\mathtt{Next}\,(g_1 \wedge g_2) \ \equiv \ (\mathtt{Next}\,g_1) \wedge (\mathtt{Next}\,g_2),\ \mathtt{Next}\,(g_1 \vee g_2) \ \equiv \ (\mathtt{Next}\,g_1) \vee (\mathtt{Next}\,g_2).$

6. *Identity:*

$g \vee C_{\mathbf{f}} \ \equiv \ g,\ g \wedge C_{\mathbf{t}} \ \equiv \ g.$

7. *Double negation:*

$\neg\neg g \ \equiv \ g$

Proof. The proofs all rely on the application of the definition of satisfaction and the properties of $\mathcal{Q}$. Let $\sigma \in \mathcal{S}^\omega$ be given.

1. Follows from the commutativity of $\mathcal{Q}$.

2. Follows from the associativity of $\mathcal{Q}$.

3. $$\begin{aligned} Sat(\sigma, g_1 \wedge g_2) &= Sat(\sigma, g_1) \wedge Sat(\sigma, g_2) \\ &= \neg(\neg Sat(\sigma, g_1) \vee \neg Sat(\sigma, g_2)) = \neg Sat(\sigma, \neg g_1 \ \vee \ \neg g_2) \\ &= Sat(\sigma, \neg(\neg g_1 \ \vee \ \neg g_2)). \end{aligned}$$

   Similarly, $g_1 \vee g_2 \ \equiv \ \neg(\neg g_1 \wedge \neg g_2)$.

4. Follows from the distributivity of $\mathcal{Q}$.

5. $$\begin{aligned} Sat(\sigma, \texttt{Next}\,(g_1 \wedge g_2)) &= Sat(\sigma_{\geq 1}, g_1 \wedge g_2) \\ &= Sat(\sigma_{\geq 1}, g_1) \wedge Sat(\sigma_{\geq 1}, g_2). \\ &= Sat(\sigma, \texttt{Next}\, g_1) \wedge Sat(\sigma, \texttt{Next}\, g_2). \\ &= Sat(\sigma, (\texttt{Next}\, g_1) \wedge (\texttt{Next}\, g_2)). \end{aligned}$$

   The proof for disjunction is similar.

6. Follows since $\mathtt{t}$ is the identity for $\wedge$ with respect to $\mathcal{Q}$ and $\mathtt{f}$ is the identity for $\vee$ with respect to $\mathcal{Q}$.

7. $Sat(\sigma, \neg\neg g) = \neg Sat(\sigma, \neg g) = \neg\neg Sat(\sigma, g) = Sat(\sigma, g)$

Since σ is arbitrary the result follows. □

A.1.4 Proof of Lemma 3.7

Lemma A.5 (Lemma 3.7).

If p is a simple predicate over $\mathcal{C}^n$, then there is a predicate $g_p \in \mathrm{TL}_n$ such that $p \equiv g_p$.

Proof.

Consider $(s_q, q) \in D(p)$, and suppose that $p(u) = q$. Then, since p is simple, for all $i = 1, \ldots, n$, $s_q[i] \sqsubseteq u[i]$. What we will do is construct functions that enable us to check whether for all $i = 1, \ldots, n$, $s_q[i] \sqsubseteq u[i]$. This will be enough of a building block to complete the proof. We define the functions χ'_q so that $\chi'_q(i, v) = \mathtt{t}$ if $s_q[i] \sqsubseteq v[i]$ and $\chi'_q(i, v) = \perp$ if $s_q[i] \not\sqsubseteq v[i]$. Formally, the χ'_is are defined as:

$$\chi'_q(i, v) = \begin{cases} \perp \;\vee \neg[i] \text{ when } s_q[i] = \mathsf{L}. \\ \perp \;\vee [i] \text{ when } s_q[i] = \mathsf{H}. \\ \perp \;\vee (\neg[i] \wedge [i]) \text{ when } s_q[i] = \mathsf{Z}. \end{cases}$$

Informally, this means that $\chi'_(q, v)$ indicates whether v is greater than the i-th component of p's defining value for q. Extending this, we get that $\bigwedge_{i=1}^{n}(\chi'_q(i, v))$ returns $\mathbf{t}$ if $s_q \sqsubseteq v$ and returns $\perp$ otherwise. Extend this further by defining:

$$
\begin{aligned}
\chi_{\mathbf{f}}(s) &= \neg(\bigwedge_{i=1}^{n}(\chi'_{\mathbf{f}}(i, s))) \vee \top, \quad \text{if } \exists(s_{\mathbf{f}}, \mathbf{f}) \in D(p) \\
&= \perp, \quad \text{otherwise.} \\
\chi_{\mathbf{t}}(s) &= \bigwedge_{i=1}^{n}(\chi'_{\mathbf{t}}(i, s)), \quad \text{if } \exists(s_{\mathbf{t}}, \mathbf{t}) \in D(p) \\
&= \perp, \quad \text{otherwise.} \\
\chi_{\top}(s) &= \neg(\bigwedge_{i=1}^{n}(\chi'_{\top}(i, s))) \vee \top, \quad \text{if } \exists(s_{\top}, \mathbf{f}) \in D(p) \\
&= \perp, \quad \text{otherwise.}
\end{aligned}
$$

Consider $\chi_{\mathbf{t}}$. Suppose $\exists s$ such that $s \sqsubseteq v$ and $p(s) = \mathbf{t}$. Since p is simple, $s_{\mathbf{t}} \sqsubseteq s$. By transitivity $s_{\mathbf{t}} \sqsubseteq v$. Hence by the remarks above, $\chi_{\mathbf{t}}(v) = \mathbf{t}$. On the other hand, if $\nexists s$ such that $s \sqsubseteq v$ and $p(s) = \mathbf{t}$, then either $\mathbf{t}$ is not in the range of p or $s_{\mathbf{t}} \not\sqsubseteq v$. In either case $\chi_{\mathbf{t}}(v) = \perp$.

For the case of $q = \mathbf{f}, \top$, suppose $\exists s$ such that $s \sqsubseteq v$ and $p(s) = q$. Since p is simple, $s_q \sqsubseteq s$. By transitivity $s_q \sqsubseteq v$. Hence by the remarks above, $\chi_q(v) = \top$. On the other hand, if $\nexists s$ such that $s \sqsubseteq v$ and $p(s) = q$, then either q is not in the range of p or $s_q \not\sqsubseteq v$. In either case $\chi_q(v) = \mathbf{t}$.

This implies that the definitions given of χ_q here are equivalent to those of Definition A.1, and thus we can apply Lemma A.2. As the χ_q are constructed here as formulas of TL$_n$, the proof is complete. $\square$

A.2 Proofs of Properties of STE

This section contains proofs of theorems and lemmas stated in Chapter 4.

Proof of Lemma 4.3

First, an auxiliary result.

Lemma A.6.

If $g \in$ TL, $q = \mathbf{f}, \mathbf{t}$, $\delta \in \Delta^q(g)$, then $q \preceq$ *Sat*(δ, g).

Proof. Let $g \in$ TL, $\delta = s_0 s_1 s_2 \ldots \in \Delta^q(g)$. Proof by structural induction.

If g is simple (base case of induction):

(1)	Either $(s_0, q) \in D(g)$ or $(s_0, \top) \in D(g)$	Definition of Δ^q
(2)	*Sat*$(\delta, g) \in \{q, \top\}$	Definition of *Sat*.
(3)	$q \preceq$ *Sat*(δ, g)	Definition of $\preceq$.

Let $g = g_1 \wedge g_2$.

(1)	*Sat*$(\delta, g) =$ *Sat*$(\delta, g_1) \wedge$ *Sat*(δ, g_2)	By definition.
	Suppose $q = \mathbf{t}$, i.e. $\delta \in \Delta^{\mathbf{t}}(g)$.	
(2a)	$\exists \delta^1 \in \Delta^q(g_1), \delta^2 \in \Delta^q(g_2) \ni \delta = \delta^1 \sqcup \delta^2$	Construction of Δ
(3a)	$q \preceq$ *Sat*(δ^1, g_1), $q \preceq$ *Sat*(δ^2, g_2)	Inductive assumption.
(4a)	$q \preceq$ *Sat*(δ, g_1), $q \preceq$ *Sat*(δ, g_2)	Monotonicity.
(5a)	$q \preceq$ *Sat*(δ, g)	(1), (4a), monotonicity of $\wedge$.
	Suppose $q = \mathbf{f}$, i.e. $\delta \in \Delta^{\mathbf{f}}(g)$.	
(2b)	Either (or both) $\delta \in \Delta^q(g_1)$ or $\delta \in \Delta^q(g_2)$	Construction of Δ
	Suppose (without loss of generality) that $\delta \in \Delta^q(g_1)$.	
(3b)	$\mathbf{f} \preceq$ *Sat*(δ^1, g_1)	Inductive assumption.
(4b)	Trivially, $\perp \preceq$ *Sat*(δ^2, g_2)	
(5b)	$\mathbf{f} \wedge \perp \preceq$ *Sat*$(\delta^1, g_1) \wedge$ *Sat*$(\delta^2, g_2) =$ *Sat*(δ, g).	(3b), (4b)
(6b)	But $\mathbf{f} \wedge \perp = \mathbf{f}$ which concludes the proof.	

Let $g = \neg g_1$.

(1) $\delta \in \Delta^{\neg q}(g_1)$ Construction of Δ.

(2) $\neg q \preceq \mathit{Sat}(\delta, g_1)$ Inductive assumption.

(3) $\neg q \preceq \neg\mathit{Sat}(\delta, g)$ $\mathit{Sat}(\delta, g) = \neg\mathit{Sat}(\delta, g_1)$.

(4) Hence $q \preceq \mathit{Sat}(\delta, g)$. Lemma 3.1(2).

Let $g = \texttt{Next}\, g_1$.

(1) $s_0 = \mathsf{X}$ and $s_1 s_2 \ldots \in \Delta^q(g_1)$ Construction of Δ.

(2) $q \preceq \mathit{Sat}(s_1 s_2 \ldots, g_1)$ Inductive assumption.

(3) $q \preceq \mathit{Sat}(\mathsf{X} s_1 s_2 \ldots, \texttt{Next}\, g_1)$ From (2), definition of *Sat*.

(4) $q \preceq \mathit{Sat}(\delta, g)$ Monotonicity of *Sat*.

Suppose $g = g_1\, \texttt{Until}\, g_2$.

By definition,

$\mathit{Sat}(\sigma, g_1\, \texttt{Until}\, g_2) \;=\; \vee_{i=0}^{\infty}(\mathit{Sat}(\sigma_{\geq 0}, g_1) \wedge \ldots \wedge \mathit{Sat}(\sigma_{\geq i-1}, g_1) \wedge \mathit{Sat}(\sigma_{\geq i}, g_2))$.

Let $\delta \in \Delta^q(g)$ be given.

Suppose $q = \mathtt{t}$, *i.e.* $\delta \in \Delta^{\mathtt{t}}(g_1\, \texttt{Until}\, g_2)$

(1) $\exists i \ni \delta \in \Delta^{\mathtt{t}}(\texttt{Next}^0 g_1) \amalg \ldots \amalg \Delta^{\mathtt{t}}(\texttt{Next}^{(i-1)} g_1) \amalg \Delta^{\mathtt{t}}(\texttt{Next}^i g_2)$

Construction of Δ

(2) $\forall j = 0, \ldots, i,\ \exists \delta^j \in \Delta^{\mathtt{t}}(\texttt{Next}^j g_1)$ such that $\sqcup\{\delta^j : j = 0, \ldots, i\} = \delta$

Definition of $\amalg$.

(3) $\delta^j \sqsubseteq \delta, j = 0, \ldots, i$ (2), property of join

(4) $\mathbf{t} \preceq \mathit{Sat}(\delta^j, \mathtt{Next}^j g_1), j = 0, \ldots, i-1$ — Inductive assumption.

(5) $\mathbf{t} \preceq \mathit{Sat}(\delta, \mathtt{Next}^j g_1)$ — (4), monotonicity.

(6) Similarly $\mathbf{t} \preceq \mathit{Sat}(\delta, \mathtt{Next}^i g_2)$

(7) $\mathbf{t} \preceq \mathit{Sat}(\delta, (\mathtt{Next}^0 g_1) \wedge \ldots \wedge (\mathtt{Next}^{(i-1)} g_1) \wedge (\mathtt{Next}^i g_2))$

(5) and (6).

(8) $\mathbf{t} \preceq \mathit{Sat}(\delta, g_1 \, \mathtt{Until} \, g_2)$ — Definition of *Sat*.

Suppose $q = \mathbf{f}$, *i.e.* $\delta \in \Delta^{\mathbf{f}}(g_1 \, \mathtt{Until} \, g_2)$.

(1) $\forall i = 0, \ldots, \exists \delta^i$ with $\delta^i \sqsubseteq \delta$ and □

$$\delta^i \in \Delta^{\mathbf{f}}(\mathtt{Next}^0 g_1) \cup \ldots \cup \Delta^{\mathbf{f}}(\mathtt{Next}^{(i-1)} g_1) \cup \Delta^{\mathbf{f}}(\mathtt{Next}^i g_2)$$

Construction of Δ.

(2) $\forall i = 0, \ldots, \delta^i \in \Delta^{\mathbf{f}}(\mathtt{Next}^0 g_1 \wedge \ldots \wedge \mathtt{Next}^{(i-1)} g_1) \wedge \mathtt{Next}^i g_2$

Definition of $\Delta^{\mathbf{f}}$

(3) $\mathbf{f} \preceq \mathit{Sat}(\delta^i, \mathtt{Next}^0 g_1 \wedge \ldots \wedge \mathtt{Next}^{(i-1)} g_1 \ \wedge \ \mathtt{Next}^i g_2)$ — Inductive assumption.

(4) $\mathbf{f} \preceq \mathit{Sat}(\delta, g_1 \, \mathtt{Until} \, g_2)$ — Definition of $g_1 \, \mathtt{Until} \, g_2$.

Lemma A.7 (Lemma 4.3).

Let $g \in$ TL, and let $\sigma \in \mathcal{S}^\omega$. For $q = \mathbf{t}, \mathbf{f}$, $q \preceq \mathit{Sat}(\sigma, g)$ iff $\exists \delta^g \in \Delta^q(g)$ with $\delta^g \sqsubseteq \sigma$.

Proof. ($\Longrightarrow$) Assume that $q \preceq \mathit{Sat}(\sigma, g)$. The proof is by structural induction.

Suppose g is simple (base case of induction).

(1) $q \preceq g(\sigma_0)$ and $\exists q' \in \{q, \top\}$ with $(s_{q'}, q') \in D(g)$ and $s_{q'} \sqsubseteq \sigma_0$

g is simple.

(2) $s_{q'}\mathsf{XX}\ldots \sqsubseteq \sigma$ — From (1).

(3) But $s_{q'}\mathsf{XX}\ldots \in \Delta^q(g)$ — Definition of $\Delta^q(g)$.

Suppose $g = g_1 \wedge g_2$.

Suppose $q = \mathtt{t}$.

(1)	$\mathtt{t} \preceq Sat(\sigma, g_1),\ \mathtt{t} \preceq Sat(\sigma, g_2)$	Lemma 3.2(2).
(2)	$\exists \delta^1 \in \Delta^{\mathtt{t}}(g_1), \delta^2 \in \Delta^{\mathtt{t}}(g_2)$ with $\delta^1, \delta^2 \sqsubseteq \sigma$	Inductive assumption.
(3)	Let $\delta = \delta^1 \sqcup \delta^2$	
(4)	$\delta \sqsubseteq \sigma$	From (2).
(5)	$\delta \in \Delta^{\mathtt{t}}(g)$	Definition of $\Delta(g)$

Suppose $q = \mathtt{f}$.

(1)	Either (or both) $\mathtt{f} \preceq Sat(\sigma, g_1)$ or $\mathtt{f} \preceq Sat(\sigma, g_2)$	Lemma 3.2(3)
	Without loss of generality assume $\mathtt{f} \preceq Sat(\sigma, g_1)$.	
(2)	$\exists \delta^1 \in \Delta^{\mathtt{f}}(g_1)$ with $\delta^1 \sqsubseteq \sigma$	Inductive assumption.
(3)	$\delta^1 \in \Delta^{\mathtt{f}}(g)$	Definition of $\Delta(g)$.

Suppose $g = \neg g_1$.

(1)	$q \sqsubseteq \neg Sat(\sigma, g_1)$	Definition of *Sat*
(2)	$\neg q \preceq Sat(\sigma, g_1)$	Lemma 3.1.
(3)	$\exists \delta \in \Delta^{\neg q}(g_1) \ni \delta \sqsubseteq \sigma$	Inductive assumption.
(4)	$\delta \in \Delta^{q}(g)$	Definition of $\Delta(g)$.

Suppose $g = \mathtt{Next}\, g_1$.

(1)	$q \preceq Sat(\sigma_{\geq 1}, g_1)$	Definition of *Sat*.
(2)	$\exists \delta \in \Delta^{q}(g_1) \ni \delta \sqsubseteq \sigma_{\geq 1}$	Inductive assumption.
(3)	$\mathsf{X}\delta \in \Delta^{q}(g)$	Construction of $\Delta(g)$.
(4)	$\mathsf{X}\delta \sqsubseteq \sigma$	$\mathsf{X} \sqsubseteq \sigma_0$

Suppose $g = g_1\, \mathtt{Until}\, g_2$.

Suppose $q = \mathbf{t}$.

(1) $\exists i \ni \mathbf{t} \preceq \mathit{Sat}(\sigma, \mathtt{Next}^0 g_1) \wedge \ldots \wedge \mathit{Sat}(\sigma, \mathtt{Next}^{(i-1)} g_1) \wedge \mathit{Sat}(\sigma, \mathtt{Next}^i g_2)$ — Lemma 3.2(1)

(2) $\mathbf{t} \preceq \mathit{Sat}(\sigma, \mathtt{Next}^i g_2)$ and — From (1), Lemma 3.2(2) .

$\forall j = 0, \ldots, i-1, \mathbf{t} \preceq \mathit{Sat}(\sigma, \mathtt{Next}^j g_1)$

(3) $\exists \delta^i \in \Delta^{\mathbf{t}}(\mathtt{Next}^i g_2) \ni \delta^i \sqsubseteq \sigma$ — Inductive assumption

$\forall j = 0, \ldots, i-1, \exists \delta^j \in \Delta^{\mathbf{t}}(\mathtt{Next}^j g_1) \ni \delta^j \sqsubseteq \sigma$

(4) Let $\delta = \delta^0 \sqcup \ldots \sqcup \delta^i$

(5) $\delta \in \Delta^{\mathbf{t}}(\mathtt{Next}^0 g_1) \amalg \ldots \amalg \Delta^{\mathbf{t}}(\mathtt{Next}^{(i-1)} g_1) \amalg \Delta^{\mathbf{t}}(\mathtt{Next}^i g_2)$ — Construction of Δ

(6) $\delta \sqsubseteq \sigma$ — (3) and (4).

(7) $\delta \in \Delta^{\mathbf{t}}(g_1\, \mathtt{Until}\, g_2)$ — (5), construction of Δ.

Suppose $q = \mathbf{f}$.

(1) $\forall i, \mathbf{f} \preceq \mathit{Sat}(\sigma, \mathtt{Next}^0 g_1) \wedge \ldots \wedge \mathit{Sat}(\sigma, \mathtt{Next}^{(i-1)} g_1) \wedge \mathit{Sat}(\sigma, \mathtt{Next}^i g_2)$ — Lemma 3.2(4)

(2) Either $\mathbf{f} \preceq \mathit{Sat}(\sigma, \mathtt{Next}^i g_2)$ or $\exists j \in 0, \ldots, i-1 \ni \mathbf{f} \preceq \mathit{Sat}(\sigma, \mathtt{Next}^j g_1)$. — Lemma 3.2(3)

(3) Either $\exists \delta'^i \ni \delta'^i \in \Delta^{\mathbf{f}}(\mathtt{Next}^i g_2)$ with $\delta'^i \sqsubseteq \sigma$, or — Inductive assumption.

$\exists \delta^j \in \Delta^{\mathbf{f}}(\mathtt{Next}^j g_1)$ and $\delta^j \sqsubseteq \sigma$.

(4) In either case, $\forall i$, by construction

$\exists \delta^i \in \Delta^{\mathbf{f}}(\mathtt{Next}^0 g_1) \cup \ldots \cup \Delta^{\mathbf{f}}(\mathtt{Next}^{(i-1)} g_1) \cup \Delta^{\mathbf{f}}(\mathtt{Next}^i g_2)$ with $\delta^i \sqsubseteq \sigma$

(5) Let $\delta = \sqcup_{i=0}^{\infty} \delta^i$.

(6) $\delta \in \Delta^{\mathbf{f}}(g_1\, \mathtt{Until}\, g_2)$ — Construction of $\Delta^{\mathbf{f}}$.

(7) $\delta \sqsubseteq \sigma$ — $\mathcal{S}$ is a complete lattice.

($\Longleftarrow$) Let $g \in$ TL, $\sigma \in \mathcal{S}^\omega$, and assume that $\exists \delta^g \in \Delta^q(g)$ such that $\delta^g \sqsubseteq \sigma$.

By Lemma A.6, $q \preceq \mathit{Sat}(\delta^g, g)$. By the monotonicity of *Sat*, $q \preceq \mathit{Sat}(\sigma, g)$. □

A.2.1 Proof of Lemma 4.4

Lemma A.8 (Lemma 4.4).

Let $g \in$ TL, and let σ be a trajectory. For $q = \mathtt{t}, \mathtt{f}$, $q \preceq \mathit{Sat}(\sigma, g)$ if and only if $\exists \tau^g \in T^q(g)$ with $\tau^g \sqsubseteq \sigma$.

Proof. ($\Longrightarrow$) Suppose $q \preceq \mathit{Sat}(\sigma, g)$.

By Lemma 4.3, $\exists \delta^g \in \Delta^q(g)$ such that $\delta^g \sqsubseteq \sigma$.

Let $\tau^g = \tau(\delta^g)$. Note that $\tau^g \in T^q(g)$ by construction and that $\delta^g \sqsubseteq \tau^g$.

$\tau^g \sqsubseteq \sigma$: the proof is by induction.

1. $\tau^g_0 = \delta^g_0 \sqsubseteq \sigma_0$.
2. Assume $\tau^g_i \sqsubseteq \sigma_i$.
3. Since σ is a trajectory,

$\mathbf{Y}(\tau^g_i) \sqsubseteq \mathbf{Y}(\sigma_i)$	Monotonicity of $\mathbf{Y}$
$\sqsubseteq \sigma_{i+1}$	σ is a trajectory.
$\delta^g_{i+1} \sqsubseteq \sigma_{i+1}$	Since $\delta^g \sqsubseteq \sigma$.
$\tau^g_{i+1} = \delta^g_{i+1} \sqcup \mathbf{Y}(\tau^g_i)$	Definition of τ^g.
$\sqsubseteq \sigma_{i+1}$	Property of join.

($\Longleftarrow$) Suppose $\exists \tau^g \in T^q(g)$ such that $\tau^g \sqsubseteq \sigma$.

As $\tau^g \in T^q(g), \exists \delta^g \in \Delta^q(g)$ such that $\delta^g \sqsubseteq \tau^g$.

By transitivity, $\delta^g \sqsubseteq \sigma$. By Lemma 4.3, $q \preceq \mathit{Sat}(\delta^g, g)$.

By monotonicity $q \preceq \mathit{Sat}(\sigma, g)$. □

A.2.2 Proof of Theorem 4.5

Theorem A.9 (Theorem 4.5).

If g and h are TL formulas, then $\Delta^{\mathbf{t}}(h) \sqsubseteq_{\mathcal{P}} T^{\mathbf{t}}(g)$ if and only if $g \Longrightarrow\!\!> h$. □

Proof. ($\Longrightarrow$) Recalling the definition of $\Longrightarrow\!\!>$ on page 71, suppose $\forall \tau^g \in T^{\mathbf{t}}(g), \exists \delta^h \in \Delta^{\mathbf{t}}(h)$ with $\delta^h \sqsubseteq \tau^g$.

Suppose $\mathbf{t} \preceq$ *Sat*(σ, g).

By Lemma 4.4, $\exists \tau^g \in T^q(g)$ such that $\tau^g \sqsubseteq \sigma$.

By assumption then, $\exists \delta^h \in \Delta^q(h)$, with $\delta^h \sqsubseteq \tau^g$. By transitivity, $\delta^h \sqsubseteq \sigma$.

By Lemma 4.3, $q \preceq$ *Sat*(σ, h).

($\Longleftarrow$) Suppose for all trajectories σ, $\mathbf{t} \preceq$ *Sat*(σ, g) implies that $\mathbf{t} \preceq$ *Sat*(σ, h).

Let $\tau^g \in T^{\mathbf{t}}(g)$.

Then by Lemma 4.4, $\mathbf{t} \preceq$ *Sat*(τ^g, g).

By the assumption that $g \Longrightarrow\!\!> h$, $\mathbf{t} \preceq$ *Sat*(τ^g, h).

By Lemma 4.3, $\exists \delta^h \in \Delta^q(h)$ such that $\delta^h \sqsubseteq \tau^g$.

As τ^g was arbitrary, the proof follows. □

A.3 Proofs of Compositional Rules for TL_n

Recall that in this section we are dealing solely with the realisable fragment of TL_n.

Theorem A.10 (Identity – Theorem 5.14).

For all $g \in \mathrm{TL}_n$, $g \Longrightarrow\!\!\rhd g$.

Proof. Let $\mathrm{t} = \mathit{Sat}(\sigma, g)$. Clearly then $\mathrm{t} = \mathit{Sat}(\sigma, g)$. Hence $g \Longrightarrow\!\!\rhd g$. ☐

Lemma A.11.

Suppose $g \Longrightarrow\!\!\rhd h$. Then $\texttt{Next}\, g \Longrightarrow\!\!\rhd \texttt{Next}\, h$

Proof.

Let $\sigma \in \mathcal{R}_{\mathcal{T}} \ni \mathrm{t} = \mathit{Sat}(\sigma, \texttt{Next}\, g)$.

(1)	$\mathrm{t} = \mathit{Sat}(\sigma_{\geq 1}, g)$	Definition of *Sat*.
(2)	$\mathrm{t} = \mathit{Sat}(\sigma_{\geq 1}, h)$	$g \Longrightarrow\!\!\rhd h$.
(3)	$\mathrm{t} = \mathit{Sat}(\mathsf{X}\sigma_{\geq 1}, \texttt{Next}\, h)$	Definition of *Sat*.
(4)	$\mathrm{t} = \mathit{Sat}(\sigma, \texttt{Next}\, h)$	(3), monotonicity of *Sat*, Lemma 4.8.
(5)	$\texttt{Next}\, g \Longrightarrow\!\!\rhd \texttt{Next}\, h$.	

☐

Corollary A.12 (Time-shift – Theorem 5.15).

Suppose $g \Longrightarrow\!\!\rhd h$. Then $\forall t \geq 0$, $\texttt{Next}^t g \Longrightarrow\!\!\rhd \texttt{Next}^t h$.

Proof. Follows from Lemma A.11 by induction. ☐

Theorem A.13 (Conjunction – Theorem 5.16).

Suppose $g_1 \Longrightarrow\!\!\rhd h_1$ and $g_2 \Longrightarrow\!\!\rhd h_2$.

Then $g_1 \wedge g_2 \Longrightarrow\!\!\rhd h_1 \wedge h_2$.

Proof.

Let $\sigma \in \mathcal{R}_{\mathcal{T}}$ and suppose $\mathrm{t} = \mathit{Sat}(\sigma, g_1 \wedge g_2)$.

(1)	$\mathbf{t} = \mathit{Sat}(\sigma, g_1) \wedge \mathit{Sat}(\sigma, g_2)$	Definition of $\mathit{Sat}(\sigma, g_1 \wedge g_2)$.
(2)	$\mathbf{t} = \mathit{Sat}(\sigma, g_i),\ i = 1, 2$	Lemma 3.2(2).
(3)	$\mathbf{t} = \mathit{Sat}(\sigma, h_i),\ i = 1, 2$	Since $g_i \Longrightarrow\!\!\triangleright h_i,\ i = 1, 2$.
(4)	$\mathbf{t} = \mathit{Sat}(\sigma, h_1) \wedge \mathit{Sat}(\sigma, h_2)$	(3)
(5)	$\mathbf{t} = \mathit{Sat}(\sigma, h_1 \wedge h_2)$	Definition of $\mathit{Sat}(\sigma, h_1 \wedge h_2)$.

As σ is arbitrary, $g_1 \wedge g_2 \Longrightarrow\!\!\triangleright h_1 \wedge h_2$. □

Theorem A.14 (Disjunction – Theorem 5.17).

Suppose $g_1 \Longrightarrow\!\!\triangleright h_1$ and $g_2 \Longrightarrow\!\!\triangleright h_2$. Then $g_1 \vee g_2 \Longrightarrow\!\!\triangleright h_1 \vee h_2$.

Proof.

Let $\sigma \in \mathcal{R}_{\mathcal{T}}$ and suppose $\mathbf{t} = \mathit{Sat}(\sigma, g_1 \vee g_2)$.

(1)	$\mathbf{t} = \mathit{Sat}(\sigma, g_1) \vee \mathit{Sat}(\sigma, g_2)$	Definition of $\mathit{Sat}(\sigma, g_1 \vee g_2)$.
(2)	$\mathbf{t} = \mathit{Sat}(\sigma, g_i)$, for $i = 1$ or $i = 2$	Lemma 3.2(1), Lemma 4.8.
(3)	$\mathbf{t} = \mathit{Sat}(\sigma, h_i)$, for $i = 1$ or $i = 2$	Since $g_i \Longrightarrow\!\!\triangleright h_i,\ i = 1, 2$.
(4)	$\mathbf{t} = \mathit{Sat}(\sigma, h_1) \vee \mathit{Sat}(\sigma, h_2)$	(3)
(5)	$\mathbf{t} = \mathit{Sat}(\sigma, h_1 \vee h_2)$	Definition of $\mathit{Sat}(\sigma, h_1 \vee h_2)$.

As σ is arbitrary, $g_1 \vee g_2 \Longrightarrow\!\!\triangleright h_1 \vee h_2$. □

Lemma A.15.

Suppose $\Delta^{\mathbf{t}}(g) \sqsubseteq_{\mathcal{P}} \Delta^{\mathbf{t}}(h)$, $\sigma \in \mathcal{R}_{\mathcal{T}}$ and $\mathbf{t} = \mathit{Sat}(\sigma, h)$.

Then $\mathbf{t} = \mathit{Sat}(\sigma, g)$.

Proof.

(1)	$\mathbf{t} \preceq \mathit{Sat}(\sigma, h)$	$\mathbf{t} = \mathit{Sat}(\sigma, h)$.
(2)	$\exists \delta \in \Delta^{\mathbf{t}}(h) \ni \delta \sqsubseteq \sigma$ and $\mathbf{t} \preceq \mathit{Sat}(\delta, h)$	(1), Lemma 4.3.
(3)	$\exists \delta' \in \Delta^{\mathbf{t}}(g) \ni \delta' \sqsubseteq \delta$	Definition of $\sqsubseteq_{\mathcal{P}}$.
(4)	$\mathbf{t} \preceq \mathit{Sat}(\delta', g)$	Lemma 4.3.

(5)	$\delta' \sqsubseteq \sigma$	Transitivity of (2) and (3).
(6)	$\mathrm{t} \preceq \mathit{Sat}(\sigma, g)$	From (4) and (5) by Lemma 4.3.
(7)	$\mathrm{t} = \mathit{Sat}(\sigma, g)$	(1), Lemma 4.8.

□

Theorem A.16 (Consequence – Theorem 5.18).

Suppose $g \Longrightarrow\!\!\rhd h$ and $\Delta^{\mathsf{t}}(g) \sqsubseteq_{\mathcal{P}} \Delta^{\mathsf{t}}(g_1)$ and $\Delta^{\mathsf{t}}(h_1) \sqsubseteq_{\mathcal{P}} \Delta^{\mathsf{t}}(h)$.

Then $g_1 \Longrightarrow\!\!\rhd h_1$.

Proof.

Suppose $\sigma \in \mathcal{R}_{\mathcal{T}}$ is a trajectory such that $\mathrm{t} = \mathit{Sat}(\sigma, g_1)$.

(1)	$\mathrm{t} = \mathit{Sat}(\sigma, g)$	Lemma A.15.
(2)	$\mathrm{t} = \mathit{Sat}(\sigma, h)$	$g \Longrightarrow\!\!\rhd h$.
(3)	$\mathrm{t} = \mathit{Sat}(\sigma, h_1)$	Lemma A.15.
(4)	$g_1 \Longrightarrow\!\!\rhd h_1$	Since σ is arbitrary.

□

Theorem A.17 (Transitivity – Theorem 5.19).

Suppose $g_1 \Longrightarrow\!\!\rhd h_1$ and $g_2 \Longrightarrow\!\!\rhd h_2$ and that $\Delta^{\mathsf{t}}(g_2) \sqsubseteq_{\mathcal{P}} \Delta^{\mathsf{t}}(g_1) \amalg \Delta^{\mathsf{t}}(h_1)$.

Then $g_1 \Longrightarrow\!\!\rhd h_2$.

Proof.

Suppose $\sigma \in \mathcal{R}_{\mathcal{T}}$ is a trajectory such that $\mathrm{t} = \mathit{Sat}(\sigma, g_1)$.

(1)	$\mathrm{t} = \mathit{Sat}(\sigma, h_1)$	$g_1 \Longrightarrow\!\!\rhd h_1$
(2)	$\mathrm{t} = \mathit{Sat}(\sigma, g_1 \wedge h_1)$	Definition of $\mathit{Sat}(\sigma, g_1 \wedge h_1)$.
(3)	$\exists \delta \in \Delta^{\mathsf{t}}(g_1 \wedge h_1) \ni \delta \sqsubseteq \sigma$	Lemma 4.3.
(4)	$\Delta^{\mathsf{t}}(g_1 \wedge h_1) = \Delta^{\mathsf{t}}(g_1) \amalg \Delta^{\mathsf{t}}(h_1)$	By definition of Δ^{t}.
(5)	$\exists \delta' \in \Delta^{\mathsf{t}}(g_2) \ni \delta' \sqsubseteq \delta$	$\Delta^{\mathsf{t}}(g_2) \sqsubseteq_{\mathcal{P}} \Delta^{\mathsf{t}}(g_1) \amalg \Delta^{\mathsf{t}}(h_1)$.

(6)	$\delta' \sqsubseteq \sigma$	Applying transitivity to (3) and (5).
(7)	$\mathrm{t} \preceq \mathit{Sat}(\sigma, g_2)$	From (6) by Lemma 4.3.
(8)	$\mathrm{t} = \mathit{Sat}(\sigma, g_2)$	From (7) by Lemma 4.8.
(9)	$\mathrm{t} = \mathit{Sat}(\sigma, h_2)$	$g_2 \Longrightarrow\!\!\triangleright h_2$.
(10)	$g_1 \Longrightarrow\!\!\triangleright h_2$	Since σ was arbitrary.

□

Lemma A.18 (Substitution Lemma).

Suppose $\models \langle\!| \, g \Longrightarrow\!\!\triangleright h \, |\!\rangle$ and let ξ be a substitution: then $\models \langle\!| \, \xi(g) \Longrightarrow\!\!\triangleright \xi(h) \, |\!\rangle$.

Proof.

Let ϕ be an arbitrary interpretation of variables and $\sigma \in \mathcal{R}_\mathcal{T}$ be an arbitrary trajectory such that $\mathrm{t} = \mathit{Sat}(\sigma, \phi(\xi(g)))$.

(1)	Let $\phi' = \phi \circ \xi$	
(2)	$\mathrm{t} = \mathit{Sat}(\sigma, \phi'(g))$	Rewriting supposition.
(3)	ϕ' is an interpretation of variables	By construction.
(4)	$\mathrm{t} = \mathit{Sat}(\sigma, \phi'(h))$	$\models \langle\!| \, g \Longrightarrow\!\!\triangleright h \, |\!\rangle$.
(5)	$\mathrm{t} = \mathit{Sat}(\sigma, \phi(\xi(h)))$	Rewriting (4).
(6)	$\models \langle\!| \, \xi(g) \Longrightarrow\!\!\triangleright \xi(h) \, |\!\rangle$	ϕ and σ were arbitrary.

□

Lemma A.19 (Guard lemma).

Suppose $e \in \mathcal{E}$ and $\models \langle\!| \, g \Longrightarrow\!\!\triangleright h \, |\!\rangle$: then $\models \langle\!| \, (e \Rightarrow g) \Longrightarrow\!\!\triangleright (e \Rightarrow h) \, |\!\rangle$.

Proof.

Suppose $\mathrm{t} = \mathit{Sat}(\sigma, e \Rightarrow g)$ for some $\sigma \in \mathcal{R}_\mathcal{T}$. Recall that $e \Rightarrow g \equiv (\neg e) \vee g$, and note that $\mathit{Sat}(\sigma, \neg e) \in \mathcal{B}$. By the definition of the satisfaction relation, either:

(i) $\mathrm{t} = \mathit{Sat}(\sigma, \neg e)$. In this case, by definition of satisfaction, $\mathrm{t} = \mathit{Sat}(\sigma, e \Rightarrow h)$.

(ii) $\mathrm{t} \;=\; \mathit{Sat}(\sigma, g)$. In this case, by assumption $\mathit{Sat}(\sigma, h)$. So, by definition of satisfaction, $\mathrm{t} = \mathit{Sat}(\sigma, e \Rightarrow h)$.

As σ was arbitrary the result follows. □

Theorem A.20 (Specialisation Theorem — Theorem 5.20).

Let $\Xi = [(e_1, \xi_1), \ldots, (e_n, \xi_n)]$ be specialisation, and suppose that $\models \langle\!| \, g \Longrightarrow\!\!\triangleright h \, |\!\rangle$.
Then $\models \langle\!| \, \Xi(g) \Longrightarrow\!\!\triangleright \Xi(h) \, |\!\rangle$.

Proof.

(1)	For $i = 1, \ldots, n$, $\models \langle\!| \, \xi_i(g) \Longrightarrow\!\!\triangleright \xi_i(h) \, |\!\rangle$	By Lemma A.18.
(2)	$\models \langle\!| \, (e_i \Rightarrow \xi_i(g)) \Longrightarrow\!\!\triangleright (e_i \Rightarrow \xi_i(h)) \, |\!\rangle$	By Lemma A.19.
(3)	$\models \langle\!| \, \bigwedge_{i=1}^{n}(e_i \Rightarrow \xi_i(g)) \Longrightarrow\!\!\triangleright \bigwedge_{i=1}^{n}(e_i \Rightarrow \xi_i(h)) \, |\!\rangle$	Repeated application of Theorem A.13.
(4)	$\models \langle\!| \, \Xi(g) \Longrightarrow\!\!\triangleright \Xi(h) \, |\!\rangle$	By definition.

□

Theorem A.21 (Until Theorem — Theorem 5.21).

Suppose $g_1 \Longrightarrow\!\!\triangleright h_1$ and $g_2 \Longrightarrow\!\!\triangleright h_2$. Then $g_1 \,\mathtt{Until}\, g_2 \Longrightarrow\!\!\triangleright h_1 \,\mathtt{Until}\, h_2$.

Proof. Let $\sigma \in \mathcal{R}_{\mathcal{T}}$ be a trajectory such that $\mathrm{t} = \mathit{Sat}(\sigma, g_1 \,\mathtt{Until}\, g_2)$.

(1)	$\exists i \ni$	Definition of *Sat*,
	$\mathrm{t} \;=\; \bigwedge_{j=0}^{i-1} \mathit{Sat}(\sigma, \mathtt{Next}^j g_1) \wedge \mathit{Sat}(\sigma, \mathtt{Next}^i g_2)$	Lemma 3.2(1), Lemma 4.8.
(2)	$\mathrm{t} \;=\; \mathit{Sat}(\sigma, \mathtt{Next}^i g_2)$ and	
	$\mathrm{t} \;=\; \mathit{Sat}(\sigma, \mathtt{Next}^j g_1),\; j = 0, \ldots, i-1$	Lemma 3.2(2).
(3)	$\mathrm{t} \;=\; \mathit{Sat}(\sigma, \mathtt{Next}^i h_2)$ and	$g_2 \Longrightarrow\!\!\triangleright h_2$, Corollary A.12.
	$\mathrm{t} \;=\; \mathit{Sat}(\sigma, \mathtt{Next}^j h_1),\; j = 0, \ldots, i-1$	$g_1 \Longrightarrow\!\!\triangleright h_1$, Corollary A.12.
(4)	$\mathrm{t} \;=\; \bigwedge_{j=0}^{i-1} \mathit{Sat}(\sigma_j, \mathtt{Next}^j h_1) \wedge \mathit{Sat}(\sigma_i, \mathtt{Next}^i h_2)$	Definition of *Sat*.
(5)	$\mathrm{t} = \mathit{Sat}(\sigma, h_1 \,\mathtt{Until}\, h_2)$	Definition of *Sat*.
(6)	$g_1 \,\mathtt{Until}\, g_2 \Longrightarrow\!\!\triangleright h_1 \,\mathtt{Until}\, h_2$	Since σ was arbitrary.

□

Appendix B

Detail of testing machines

This chapter presents the details of testing machines. Section B.1 formally defines composition of machines. Subsequent sections build on this by showing how testing machines can be constructed and composed with the circuit under test: Section B.2 presents some notation used; Section B.3 presents the building blocks from which testing machines are constructed; and Section B.4 shows how model checking is accomplished.

B.1 Structural Composition

The focus of the research on composition is the property composition described in Chapter 5. However, sometimes it is also desirable to reason about different models and use partial results to describe the behaviour of the composition of the models. A full exploration of composing models of partially ordered state spaces is beyond the scope of this thesis — there are important considerations which need attention [129]. A partial exploration of the area is useful though for two reasons: (1) it gives a flavour of how structural composition could be used; and (2) some of the definitions given are needed in justifying the details of testing machines.

The content of this section is very technical. Although conceptually the composition of systems is very simple, the notation needed to keep track of the detail is not. This section is included for completeness and the details of this section are not needed in understanding the thesis.

This section has three parts. First, composition of models is defined formally. Second, inference rules for reasoning about a composed model is given. The third part elaborates on composition for circuit models, where the definition of composition has natural instantiation.

B.1.1 Composition of Models

Definition B.1.

Let $\mathcal{M}_1 = (\langle \mathcal{S}_1, \sqsubseteq_1 \rangle, \mathcal{R}_1, \mathbf{Y}_1)$, $\mathcal{M}_2 = (\langle \mathcal{S}_2, \sqsubseteq_2 \rangle, \mathcal{R}_2, \mathbf{Y}_2)$, and $\mathcal{M} = (\langle \mathcal{S}, \sqsubseteq \rangle, \mathcal{R}, \mathbf{Y})$ be models. Let $\mathsf{X}_1, \mathsf{X}_2$ and X be the bottom elements of $\mathcal{S}_1$, $\mathcal{S}_2$ and $\mathcal{S}$; $\mathsf{Z}_1, \mathsf{Z}_2$, and Z be their top elements; and let G_1, G_2 and G be the simple predicates of $\mathcal{S}_1$, $\mathcal{S}_2$ and $\mathcal{S}$ respectively.

If $\rho : \mathcal{S}_1 \times \mathcal{S}_2 \to \mathcal{S}$, $\rho_1 : G_1 \to G$ and $\rho_2 : G_2 \to G$ then $\mathcal{M}$ is a ρ-composition of $\mathcal{M}_1$ and $\mathcal{M}_2$ if

1. ρ is monotonic;
2. $\rho(\mathsf{X}_1, \mathsf{X}_2) = \mathsf{X}$ and $\rho(\mathsf{Z}_1, \mathsf{Z}_2) = \mathsf{Z}$;
3. $q = g_1(s_1) \implies q = \rho_1(g_1)(\rho(s_1, \mathsf{X}_2))$;
4. $q = g_2(s_2) \implies q = \rho_2(g_2)(\rho(\mathsf{X}_1, s_2))$;
5. $\rho(\mathbf{Y}_1(s_1), \mathbf{Y}_2(s_2)) \sqsubseteq \mathbf{Y}(\rho(s_1, s_2))$.

The required properties on ρ may seem onerous, and indeed in general they may be too restrictive. However, for the application of composition needed in this thesis they are sufficient. In particular, for compositions where the 'outputs' of one circuit are connected to the 'inputs' of another, these conditions will be met.

Definition B.2.

We inductively extend the domain of the ρ_i by defining

- $\rho_i(g \wedge h) = \rho_i(g) \wedge \rho_i(h)$;
- $\rho_i(\neg g) = \neg(\rho_i(g))$;
- $\rho_i(\texttt{Next}\, g) = \texttt{Next}\, \rho_i(g)$;
- $\rho_i(g\, \texttt{Until}\, h) = \rho_i(g)\, \texttt{Until}\, \rho_i(h)$.

Definition B.3.

Let $\mathcal{M}_1, \mathcal{M}_2$ and be models and $\mathcal{M}$ a ρ-composition of $\mathcal{M}_1$ and $\mathcal{M}_2$. Let $\sigma^1 \in \mathcal{S}_1^\omega$, $\sigma^2 \in \mathcal{S}_2^\omega$. $\rho(\sigma^1, \sigma^2) = \rho(\sigma_0^1, \sigma_0^2)\rho(\sigma_1^1, \sigma_1^2)\ldots$

Since we are dealing with different models, we modify the notation for the satisfaction relation and use the notation $Sat_{\mathcal{M}_j}(\sigma, g)$ to refer to whether the sequence σ of the model $\mathcal{M}_j$ satisfies g.

Lemma B.1.

Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be models and $\mathcal{M}$ a ρ-composition of $\mathcal{M}_1$ and $\mathcal{M}_2$. Let $\sigma^i \in \mathcal{S}_i^\omega,\ j = 1, 2$. Suppose $g \in \mathrm{TL}(\mathcal{S}_j)$ and $q = Sat_{\mathcal{M}_j}(\sigma^j, g)$. Then $q \preceq Sat_{\mathcal{M}}(\rho(\sigma^1, \sigma^2), \rho_j(g))$.

Proof. The proof is by induction on the structure of g. We assume without loss of generality that $j = 1$.

Suppose $q = \mathrm{Sat}_{\mathcal{M}_1}(\sigma^1, g)$ *where* g *is simple*

(1)	$q = g(\sigma_0^j)$	Definition of satisfaction.
(2)	$q = \rho_1(g)(\rho(\sigma_0^1, \mathsf{X}_2))$	Definition B.1(3).
(3)	$q \preceq \rho_1(g)(\rho(\sigma_0^1, \sigma_0^2))$	Monotonicity.
(4)	$q \preceq Sat_{\mathcal{M}}(\rho(\sigma^1, \sigma^2), \rho_1(g))$	Definition of satisfaction.

Suppose $q = \mathrm{Sat}_{\mathcal{M}_1}(\sigma^1, g_a \wedge g_b)$

(1)	Let $q_w = Sat_{\mathcal{M}_1}(\sigma^1, g_w),\ w = a, b$	
(2)	$q = q_a \wedge q_b$	Definition of satisfaction.
(3)	$q_w \preceq Sat_{\mathcal{M}}(\rho(\sigma^1, \sigma^2), \rho_1(g_w)), w = a, b$	Inductive assumption.
(4)	$q \preceq Sat_{\mathcal{M}}(\rho(\sigma^1, \sigma^2), \rho_1(g_a \wedge g_b))$	Definition B.2 and satisfaction.

Suppose $q = \mathrm{Sat}_{\mathcal{M}_1}(\sigma^1, \neg h)$

(1)	$\neg q = Sat_{\mathcal{M}_1}(\sigma^1, h)$	$Sat(\sigma^1, \neg f) = \neg Sat(\sigma^1, f)$

(2) $\neg q \preceq \mathit{Sat}_{\mathcal{M}}(\rho(\sigma^1, \sigma^2), \rho_1(h))$ Inductive assumption.

(3) $q \preceq \mathit{Sat}_{\mathcal{M}}(\rho(\sigma^1, \sigma^2), \rho_1(\neg h))$ Definition B.2 and satisfaction.

Suppose $q = \mathrm{Sat}_{\mathcal{M}_1}(\sigma^1, \mathtt{Next}\, h)$

(1) $q = \mathit{Sat}_{\mathcal{M}_1}(\sigma^1_{\geq 1}, h)$ $\mathit{Sat}(\sigma^1, \mathtt{Next}\, h) = \mathit{Sat}(\sigma^1_{\geq 1}, h)$

(2) $q \preceq \mathit{Sat}_{\mathcal{M}}(\rho(\sigma^1_{\geq 1}, \sigma^2_{\geq 1}), \rho_1(h))$ Inductive assumption.

(3) $q \preceq \mathit{Sat}(\rho(\sigma^1, \sigma^2), \rho_1(\mathtt{Next}\, h))$ Definition B.2 and satisfaction.

Suppose $q = \mathrm{Sat}_{\mathcal{M}_1}(\sigma^1, h_1\, \mathtt{Until}\, h_2)$

(1) Let $q_j = \mathit{Sat}_{\mathcal{M}_1}(\sigma^1_{\geq 0}, h_1) \wedge \ldots \wedge \mathit{Sat}_{\mathcal{M}_1}(\sigma^1_{\geq j-1}, h_1) \wedge \mathit{Sat}_{\mathcal{M}_1}(\sigma^1_{\geq j}, h_2)$

(2) $q = \vee_{j=0}^{\infty} q_j$ Definition of satisfaction.

(3) $q_j \preceq \mathit{Sat}_{\mathcal{M}}(\rho(\sigma^1_{\geq 0}, \sigma^2_{\geq 0}), \rho_1(h_1)) \wedge \ldots \wedge \mathit{Sat}_{\mathcal{M}}(\rho(\sigma^1_{\geq j-1}, \sigma^2_{\geq j-1}), \rho_1(h_1)) \wedge$
$\mathit{Sat}_{\mathcal{M}}(\rho(\sigma^1_{\geq j}, \sigma^2_{\geq j}), \rho_1(h_2))$ From (1) by induction assumption.

(4) $q \preceq \vee_{j=0}^{\infty} (\mathit{Sat}_{\mathcal{M}}(\rho(\sigma^1_{\geq 0}, \sigma^2_{\geq 0}), \rho_i(h_1)) \wedge \ldots \wedge \mathit{Sat}_{\mathcal{M}}(\rho(\sigma^1_{\geq j-1}, \sigma^2_{\geq j-1}), \rho_1(h_1)) \wedge$
$\mathit{Sat}_{\mathcal{M}}(\rho(\sigma^1_{\geq j}, \sigma^2_{\geq j}), \rho_1(h_2)))$ (2) and (3)

(5) $q \preceq \mathit{Sat}_{\mathcal{M}}(\rho(\sigma^1, \sigma^2), h_1\, \mathtt{Until}\, h_2)$ Definition of satisfaction.

$\square$

Lemma B.2.

Let $\mathcal{M}_1, \mathcal{M}_2$ and be models and $\mathcal{M}$ a ρ-composition of $\mathcal{M}_1$ and $\mathcal{M}_2$. Let $i \in \{1, 2\}$, and suppose that:

(1) ρ is a surjection;

(2) $\sigma = \rho(\sigma^1, \sigma^2)$, and $\mathrm{t} \preceq \mathit{Sat}_{\mathcal{M}}(\sigma, \rho_i(g_i))$ implies that $\mathrm{t} \preceq \mathit{Sat}_{\mathcal{M}_i}(\sigma^i, g_i)$.

Then $\models_{\mathcal{M}} \langle\!| \rho_i(g_i) \Longrightarrow\!\!\!> \rho_i(h) |\!\rangle$ iff $\models_{\mathcal{M}_i} \langle\!| g_i \Longrightarrow\!\!\!> h_i |\!\rangle$.

Proof.

Suppose $\models_{\mathcal{M}_i} \langle\!\langle\, g_i \Longrightarrow\!\!\!> h_i \,\rangle\!\rangle$

(1) Suppose $\mathbf{t} \preceq \mathit{Sat}_{\mathcal{M}}(\sigma, \rho_i(g_i))$

(2) $\exists \sigma^1, \sigma^2 \ni \sigma = \rho(\sigma^1, \sigma^2)$ — ρ is a surjection.

(3) $\mathbf{t} \preceq \mathit{Sat}_{\mathcal{M}_i}(\sigma^i, g_i)$ — By hypothesis (2).

(4) $\mathbf{t} \preceq \mathit{Sat}_{\mathcal{M}_i}(\sigma^i, h_i)$ — Since $\models_{\mathcal{M}_i} \langle\!\langle\, g_i \Longrightarrow\!\!\!> h_i \,\rangle\!\rangle$.

(5) $\mathbf{t} \preceq \mathit{Sat}_{\mathcal{M}}(\sigma, \rho_i(h_i))$ — By Lemma B.1.

(6) Therefore $\models_{\mathcal{M}} \langle\!\langle\, \rho_i(g_i) \Longrightarrow\!\!\!> \rho_i(h_i) \,\rangle\!\rangle$

Suppose $\models_{\mathcal{M}_i} \langle\!\langle\, \rho_i(g_i) \Longrightarrow\!\!\!> \rho_i(h_i) \,\rangle\!\rangle$

(1) Suppose $\mathbf{t} \preceq \mathit{Sat}_{\mathcal{M}_i}(\sigma_i, g_i)$

(2) $\mathbf{t} \preceq \mathit{Sat}_{\mathcal{M}}(\sigma, \rho_i(g_i))$ — Lemma B.1

(3) $\mathbf{t} \preceq \mathit{Sat}_{\mathcal{M}}(\sigma, \rho_i(h_i))$ — $\mathbf{t} \preceq \mathit{Sat}_{\mathcal{M}_i}(\sigma_i, g_i)$.

(4) $\mathbf{t} \preceq \mathit{Sat}_{\mathcal{M}}(\sigma, h_i)$ — By hypothesis 2.

□

B.1.2 Composition of Circuit Models

For circuit models, there are several natural definitions of composition that have useful properties. There are, of course, other ways of composing circuits, but the one discussed here is simple and useful.

Let $\mathcal{M}_1 = (\langle \mathcal{C}^{m_1}, \sqsubseteq \rangle, \mathcal{A}^{m_1}, \mathbf{Y}_1)$ and $\mathcal{M}_2 = (\langle \mathcal{C}^{m_2}, \sqsubseteq \rangle, \mathcal{A}^{m_2}, \mathbf{Y}_2)$ be two models. For circuit models, the next state function $\mathbf{Y} : \mathcal{C}^n \to \mathcal{C}^n$ is represented as a vector of next state function $\langle \mathbf{Y}[1], \ldots, \mathbf{Y}[n] \rangle$ where each $\mathbf{Y}[j] : \mathcal{C}^n \to \mathcal{C}$ and $\mathbf{Y}(s) = \langle \mathbf{Y}[1](s), \ldots, \mathbf{Y}[n](s) \rangle$.

To compose two circuit models, we identify r pairs of nodes (each pair comprising one node of both circuits) and 'join' the pairs (i.e., informally, think of these pairs as being soldered together, or physically identical). The state space of the composed circuit consists of $m_1 + m_2 - r$ components. The first $m_1 - r$ components are the components of $\mathcal{M}_1$ that are not shared with

$\mathcal{M}_2$. The next $m_2 - r$ components are the components of $\mathcal{M}_2$ that are not shared with $\mathcal{M}_1$. The final r components are the r components shared by both $\mathcal{M}_1$ and $\mathcal{M}_2$.

The formal definition of composition is a little intricate since it identifies state components by indices. The difficult part of the definition is identifying for each state component in the composed circuit the component or components in $\mathcal{M}_1$ and $\mathcal{M}_2$ that make it up.[1] The idea is simple — the book-keeping is unfortunately off-putting.

Let $I_1 = \langle a_1, \ldots, a_r \rangle$, and $J_1 = \langle a'_1, \ldots, a'_{m_1-r} \rangle$ be lists of state components of $\mathcal{M}_1$. If $s \in \mathcal{S}_1$, then the $s[a_j]$ are the components of the state space that are shared with $\mathcal{M}_2$ and the $s[a'_j]$ are the components of the state space that are not. We place the natural restriction that I_1 and J_1 are disjoint, and that their elements are arranged in strictly ascending order. $I_2 = \langle b_1, \ldots, b_r \rangle$ and $J_2 = \langle b'_1, \ldots, b'_{m_2-r} \rangle$ are the corresponding of lists for $\mathcal{M}_2$. Each (a_j, b_j) pair is a pair of state components that must be 'joined'.

Let $conv_1(j)$ be the component of $\mathcal{M}$ to which the j-th component of $\mathcal{M}_1$ contributes. Formally, define

$$conv_1(j) = \begin{cases} k & \text{when } \exists a'_k \in J_1 \ni a'_k = j \\ m_1 + m_2 - r + k & \text{when } \exists a_k \in I_1 \ni a_k = j \end{cases}$$

Similarly, define

$$conv_2(j) = \begin{cases} m_1 - r + k & \text{when } \exists b'_k \in J_2 \ni b'_k = j \\ m_1 + m_2 - r + k & \text{when } \exists b_k \in I_2 \ni b_k = j \end{cases}$$

Since the I_i and J_i are distinct, we can define an inverse to $conv_i$. Define $index_i(j) = k$ where $conv_i(k) = j$. Note that $index_i$ is not defined on all of $\{1, \ldots, m_1 + m_2 - r\}$, but that where it is defined, $index_i(j)$ is the component of the state space of $\mathcal{M}_i$ which contributes to the j-th component of $\mathcal{M}$. With this technical framework, composition can be defined easily. If $s_1 \in$

[1] In practice, composition is a lot easier. Nodes are labelled by names drawn from a global space. We use the convention that if the same name appears in both circuits, then the nodes they label are actually the same physical node. Thus, the pairs that must be connected are implicit and do not have to be given.

$\mathcal{C}^{m_1}$ and $s_2 \in \mathcal{C}^{m_2}$, define $\tilde{\varrho}(s_1, s_2)$ by

$$\begin{aligned}\tilde{\varrho}(s_1, s_2) = \langle & s_1[\mathit{index}_1(1)], \ldots, s_1[\mathit{index}_1(m_1 - r)], \\ & s_2[\mathit{index}_2(m_1 - r + 1)], \ldots, s_2[\mathit{index}_2(m_1 + m_2 - 2r)], \\ & s_1[\mathit{index}_1(m_1 + m_2 - 2r + 1)] \sqcup s_2[\mathit{index}_2(m_1 + m_2 - 2r + 1)], \ldots, \\ & s_1[\mathit{index}_1(r)] \sqcup s_2[\mathit{index}_2(m_1 + m2 - r)] \,\rangle\end{aligned}$$

Part of defining the composition is to define the mapping from simple predicates in $\mathcal{M}_1$ and $\mathcal{M}_2$ to $\mathcal{M}$. For TL$_n$, this is easy since it is only for predicates of the form $[j]$ that a non-trivial mapping has to be defined.

Define

$$\tilde{\varrho}_i(g) = \begin{cases} [\mathit{conv}_i(j)] & \text{when } g = [j] \text{ for some } j \\ g & \text{when } g \text{ a constant predicates.} \end{cases}$$

Then define

$$\mathcal{M} = (\langle \mathcal{S}^{m_1 + m_2 - r}, \sqsubseteq \rangle, \mathbf{Y})$$

where

$$\mathbf{Y}[j](s) = \begin{cases} \mathbf{Y}_1[\mathit{index}_1(j)](\langle s[\mathit{conv}_1(1)], \ldots, s[\mathit{conv}_1(m_1)]\rangle), & j \leq m_1 - r \\ \mathbf{Y}_2[\mathit{index}_2(j)](\langle s[\mathit{conv}_2(1)], \ldots, s[\mathit{conv}_2(m_2)]\rangle), & m_1 - r < j \leq m_1 + m_2 - 2r \\ \mathbf{Y}_1[\mathit{index}_1(j)](\langle s[\mathit{conv}_1(1)], \ldots, s[\mathit{conv}_1(m_1)]\rangle) \sqcup & \\ \mathbf{Y}_2[\mathit{index}_2(j)](\langle s[\mathit{conv}_2(1)], \ldots, s[\mathit{conv}_2(m_2)]\rangle), & m_1 + m_2 - 2r \leq j \end{cases}$$

Then $\mathcal{M}$ is a $\tilde{\varrho}$-composition of $\mathcal{M}_1$ and $\mathcal{M}_2$, denoted $\tilde{\varrho}(\mathcal{M}_1, \mathcal{M}_2)$.

Lemma B.3.

$\tilde{\varrho}$ meets all the criteria given in Definition B.1.

Proof.

(1) $\tilde{\varrho}$ **is monotonic.**

$\tilde{\varrho}$ is defined component-wise.

Each component is constructed from the identity and join functions.

Since both of these are monotonic, monotonicity follows.

(2) $\tilde{\varrho}(\mathsf{U}^{m_1}, \mathsf{U}^{m_2}) = \mathsf{U}^{m_1+m_2-r}$ **and** $\tilde{\varrho}(\mathsf{Z}^{m_1}, \mathsf{Z}^{m_2}) = \mathsf{Z}^{m_1+m_2-r}$

Follows straight from the definition of $\tilde{\varrho}$.

(3) $q = g_1(s_1)$ **implies that** $q \preceq \rho_1(g_1)(\rho_1(s_1, \mathsf{U}_2^{m_2}))$

a. Suppose $q = g_1(s_1)$, let $s = \tilde{\varrho}(s_1, \mathsf{U}^{m_2})$.

b. $s_1[j] = s[\mathit{conv}_1(j)]$ From definition of $\tilde{\varrho}(s_1, \mathsf{U}^{m_2})$.

c. Let $g_1 \in G_1$.

Suppose $g_1 = [j]$ for some j.

d. $\tilde{\varrho}_1([j]) = [\mathit{conv}_1(j)]$. Definition of $\tilde{\varrho}_1$.

e. $\tilde{\varrho}_1(g_1)(s) = g_1(s_1)$ (b) and (d).

f. $\tilde{\varrho}_1(g_1)(s) = q$ By assumption and (e).

Otherwise g_1 must be one of the constant predicates $\{\bot, \mathbf{f}, \mathbf{t}, \top\}$.

g. $\tilde{\varrho}_1(g_1) = g_1$ Definition of $\tilde{\varrho}$.

h. $q = \tilde{\varrho}_1(g_1)(s)$ $\tilde{\varrho}_1(g_1)$ is constant.

(4) Similarly $q = g_2(s_2) \implies q \preceq \varrho_2(g_2)(\varrho(\mathsf{U}_1, s_2))$**.**

(5) $\tilde{\varrho}(\mathbf{Y}_1(s_1), \mathbf{Y}_2(s_2)) \preceq \mathbf{Y}(\tilde{\varrho}(s_1, s_2))$**.** Proved by showing for all j,

$$\tilde{\varrho}(\mathbf{Y}_1(s_1), \mathbf{Y}_2(s_2))[j] \preceq \mathbf{Y}[j](\tilde{\varrho}(s_1, s_2)).$$

Suppose $j \leq m_1 - r$.

$$\begin{aligned}
&\tilde{\varrho}(\mathbf{Y}_1(s_1), \mathbf{Y}_2(s_2))[j] \\
&= \mathbf{Y}_1(s_1)[\mathit{index}_1(j)] \\
&= \mathbf{Y}_1[\mathit{index}_1(j)](s_1) \\
&= \mathbf{Y}_1[\mathit{index}_1(j)] \\
&= (\langle s[\mathit{conv}_1(1)], \ldots, s[\mathit{conv}_1(m_1)]\rangle) \\
&= \mathbf{Y}[j](s)
\end{aligned}$$

Similarly if $m_1 - r < j \leq m_1 + m_2 - 2r$,

$$\tilde{\varrho}(\mathbf{Y}_1(s_1), \mathbf{Y}_2(s_2))[j] = \mathbf{Y}_2[\mathit{index}_2(j)](s_2) = \mathbf{Y}[j](s)$$

Suppose $m_1 + m_2 - 2r \leq j$

$$\begin{aligned}
&\tilde{\varrho}(\mathbf{Y}_1(s_1), \mathbf{Y}_2(s_2))[j] \\
&= \mathbf{Y}_1(s_1)[\mathit{index}_1(j)] \sqcup \mathbf{Y}_2(s_2)[\mathit{index}_2(j)] \\
&= \mathbf{Y}_1[\mathit{index}_1(j)](s_1) \sqcup \mathbf{Y}_2[\mathit{index}_2(j)](s_2) \\
&= \mathbf{Y}[j](s)
\end{aligned}$$

□

Lemma B.3 is important because it means that Lemma B.1 can be used. Furthermore, where composition of machines is done is such a way that the ‘outputs’ of one machine are connected to the ‘inputs’ of the other (so there is no ‘feedback’ — signals go from one machine to the other, but not *vice versa*.), Lemma B.2 applies too (to the circuit that provides the outputs).

This definition is dependent on I_1, I_2, J_1 and J_2; for convenience, the following short-hand is used: $\tilde{\varrho}(A, B)\langle r_1, \ldots, r_k\rangle$ refers to the composition of A and B where the $r_1, \ldots, r_k$ components of A are shared with the first k components of B; formally $I_1 = \langle r_1, \ldots, r_k\rangle$, $J_1 = \langle 1, \ldots, r_1 - 1, r_1 + 1, \ldots, r_k - 1, r_k + 1, \ldots, k\rangle$, $I_2 = \langle 1, \ldots, k\rangle$ and $J_2 = \langle k + 1, \ldots, k\rangle$.

B.2 Mathematical Preliminaries for Testing Machines

This section assumes the state space of the system is $\mathcal{C}^n$ for some n. There is nothing inherent in the method which limits the state space to this. However, from a notational point of view it is easier to explain the method with this simple case; furthermore, this is the important, practical case. The method generalises easily to an arbitrary complete lattice.

Suppose that M_1 and M_2 have both been derived from a common machine, M, using a sequence of compositions. (Assume that $M = \langle \mathcal{C}^n, \mathbf{Y}^* \rangle$, $M_1 = \langle \mathcal{C}^{n+m_1}, \mathbf{Y}_1 \rangle$ and $M_2 = \langle \mathcal{C}^{n+m_2}, \mathbf{Y}_2 \rangle$.) By the definition of composition the two next state functions $\mathbf{Y}_1$ and $\mathbf{Y}_2$ restricted to $\mathcal{C}^n$ are identical and the same as $\mathbf{Y}^*$.

M_i ($i = 1, 2$) consists of M and a tester T_i. The relative composition of M_1 and M_2 with respect to M is the composition of M, T_1 and T_2. All of this could be described by composition, but it is convenient to define a specific notation. Formally, $rel\_comp_M(M_1, M_2) = \langle \mathcal{C}^{n'}, \mathbf{Y} \rangle$, where $n' = n + m_1 + m_2$ and if the current state is s, $\langle t_1, \ldots, t_{n'} \rangle = \mathbf{Y}(s)$ is defined by:

$$t_j = \begin{cases} \mathbf{Y}_1(\langle s_1, \ldots, s_{n+m_1} \rangle)[j] & \text{when } 1 \leq j \leq n + m_1 \\ \mathbf{Y}_2(\langle s_1, \ldots, s_n, s_{n+m_1+1}, \ldots, s_{n'} \rangle)[j] & \text{when } n + m_1 < j \leq n' \end{cases}$$

B.3 Building Blocks

Basic Block BB_A

Some predicates may depend (in some way) on the value of node 1 at a time t_1 and node 2 at time t_2. (Formally, a 'node' is a component of the state space; informally since we are reasoning about physical circuits nodes are wires in the circuit.) The purpose of BB_A is to provide delay slots so that both values of interest are available at the same 'time'. BB_A takes two parameters: a function $g : \mathcal{C} \to \mathcal{C}$ which is used to combine the values, and n which indicates how many delay slots need to be constructed. Figure B.1 depicts $BB_A(g, 4)$.

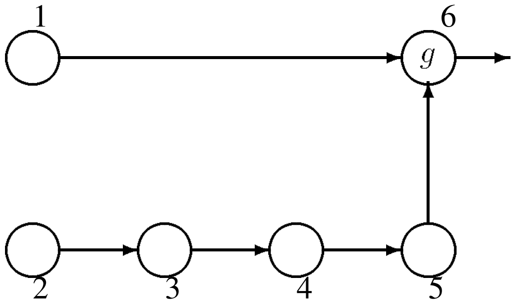


Figure B.1: $BB_A(g, 3)$: a Three Delay-Slot Combiner

$BB_A(g, n)$ consists of two nodes which act as input nodes, n delay slots, and one node which is the output value of the machine. The two inputs nodes are typically part of the original circuit, which is why BB_A's next state function does not affect the first two components. Formally,

$$BB_A(g, n) = \langle \mathcal{C}^{n+3}, \mathbf{Y} \rangle \text{ where if } t = \mathbf{Y}(s), \text{ then } t_j = \begin{cases} \bot, & \text{when } j = 1, 2; \\ s_{j-1}, & \text{when } j = 3, \ldots n+2; \\ g(s_1, s_{n-1}), & \text{when } j = n+3. \end{cases}$$

The $comp\_test$ operator adds the BB_A circuit to an existing circuit. Given a machine M and a predicate g which depends on the value of i_1 at time t_1 and i_2 at time t_2, the composite machine (M plus the testing circuit) is defined by:

$$comp\_test(M, g, (t_1, i_1), (t_2, i_2)) = \begin{cases} \tilde{\varrho}(M, BB_A(g, t_1 - t_2))\langle i_1, i_2 \rangle & \text{if } t_1 \geq t_2 \\ \tilde{\varrho}(M, BB_A(g, t_2 - t_1))\langle i_2, i_1 \rangle & \text{otherwise.} \end{cases} \tag{B.1}$$

The problem with the BB_A testing machine is that if the two defining times are far apart, the testing circuit could be large due to the need to retain and propagate values, which has both space and computation costs associated.

There is an alternative approach – build a memory into the circuit which keeps the needed information. Define $BB_A(g) = \langle \mathcal{C}^5, \mathbf{Y} \rangle$. If the state of the machine is $\langle s_1, \ldots, s_5 \rangle$, think of s_1 and s_2 as the inputs, and s_5 as the output. s_4 is used as the memory, and s_3 indicates whether the memory's value should be reset or maintained. Formally,

$$\mathbf{Y}(s_j) = \begin{cases} \bot, & \text{when } j = 1, 2, 3 \\ s_2, & \text{when } j = 4 \text{ and } s_3 = 0 \\ s_4, & when j = 4 \text{ and } s_3 = 1 \\ g(s_1, s_4), & \text{when } j = 5 \end{cases} \tag{B.2}$$

Define

$$comp\_test(M, g, (t_1, i_1), (t_2, i_2)) = \begin{cases} \tilde{\varrho}(M, BB_A(g))\langle i_1, i_2\rangle & \text{if } t_1 \geq t_2 \\ \tilde{\varrho}(M, BB_A(g))\langle i_2, i_1\rangle & \text{otherwise.} \end{cases} \tag{B.3}$$

Although in general the definition of equation B.3 will be more efficient, it cannot always be used. To see why consider this example. Let g and h be two predicates containing no temporal operators, where the result of g can be found at node i_1 and the value of h at node i_2. Suppose we want to evaluate the predicate $g \wedge \texttt{Next}^3 h$. Implicit in this is that we are interested in i_1 at time 0 and i_2 at time 3. For this we could use the second implementation of $comp\_test$ and get the new machine

$comp\_test(M, (\lambda x, y.x \wedge y), (0, i_1), (3, i_2)).$

Now suppose that we are interested in the predicate $\texttt{Exists}\,[(0, 10)]\,(g \wedge \texttt{Next}^3 h)$. This asks whether there is a time t between 0 and 10 such that g holds at time t and h holds at time $t + 3$. For this predicate, the second implementation will not work since it only remembers the value of g at one particular time, and we need to have the value of g at a sequence of times.

The general rule in choosing between the implementations is that if the predicate for which the tester being constructed is within the scope of temporal operators such as `Exists` and `Global`, then the first implementation must be used; otherwise the second, more efficient implementation can be used.

Basic Block BB_B

BB_B is used when we need to combine the value of a predicate at a number of different times. For example, `Global` g and `Exists` g depend on the value of g at a sequence of times. Define $BB_B(g,k) = \langle \mathcal{C}^{k+2}, \mathbf{Y} \rangle$ where if $t = \mathbf{Y}(s)$ then:

$$t_j = \begin{cases} \perp & \text{when } j = 1 \\ s_1 & \text{when } j = 2 \\ f(s_j, s_{j-1}) & \text{otherwise.} \end{cases}$$

Figure B.2 depicts this graphically.

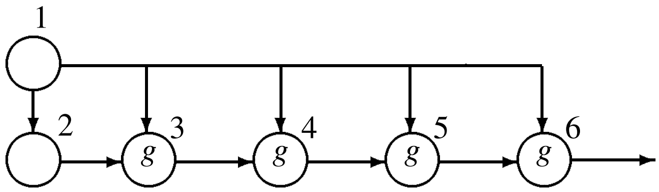


Figure B.2: $BB_B(g, 4)$

Basic Block BB_C

This is just a simple latch with a comparator. $BB_C = \langle \mathcal{C}^3, \mathbf{Y} \rangle$ where $\mathbf{Y}(\langle s_1, s_2, s_3 \rangle) = \langle \perp, s_2, s1 = s2 \rangle$.

Inverter

Define $I = \langle \mathcal{C}^2, \mathbf{Y} \rangle$ where $\mathbf{Y}(\langle s_1, s_2 \rangle) = \langle \perp, \neg s_1 \rangle$.

B.4 Model Checking

This section shows how to accomplish the following:

Given a machine M and an assertion of the form $\models \langle\!| A \Longrightarrow\!\!\triangleright g |\!\rangle$, construct a machine M' and trajectory formulae A', C such that $\models \langle\!| A \Longrightarrow\!\!\triangleright g |\!\rangle \iff \models_{M'} A \wedge A' \Longrightarrow\!\!\triangleright C$.

Every temporal formula, g, has an associated tuple (i, t, A, M'): i indicates that the formula can be evaluated by examining the i-th component of the state space of the new machine; t indicates the time at which the component should be examined; A' gives a set of trajectory formulas which are used as auxiliary antecedents for the new machine; and M' is the new machine. The tuple is defined recursively on the structure of the temporal formula.[2]

1. g is $([i] = v)$. The tester which checks this compares the value of node i to v. The associated tuple is $(n + 2, 1, A', M')$, where
   - $A' = ([n + 2] = v)$
   - $M' = \tilde{\varrho}(M, BB_C)\langle i\rangle$.
2. g is $\texttt{Next}^j g'$. This does not require any extra circuitry — the tester that tests g' is already built in, and the only difference is that the result is checked at a different time. If the tuple associated with g is (i, t, A, M'), the tuple associated with $\texttt{Next}\ jg$ is $(i, t + j, A, M')$.
3. g is $\neg g'$. If the tester for g' is already built, an inverter will compute the answer for g. So, if the tuple associated with g is (i, t, A, M'), the tuple associated with $\neg f$ is $(|M''|, t + 1, A, M'')$, where $M'' = \tilde{\varrho}(M, I)\langle i\rangle$.
4. g is $g'(g_1, g_2)$. Typically g' would be conjunction or disjunction. The tester takes as its input the results of g_1 and g_2 and applies g' to them. Let the tuple associated with g_1 be (i_1, t_1, A_1, M_1) and the tuple associated with g_2 be (i_2, t_2, A_2, M_2). Assume that $|M_1| = n + m_1$ and $|M_2| = n + m_2$.

[2]Note that in this discussion M refers to the original machine, and $n = |M|$.

The tuple associated with $g(g_1, g_2)$ is $(|M'|, \max(t_1, t_2) + 1, A_1 \wedge A_2, M')$ where $M' = \mathit{comp\_test}(M'', g, (t_1, n+m1), (t_2, n+m1+m2))$ and $M'' = \mathit{rel\_comp}_M(M_1, M_2)$.[3]

5. g is `Global` $[(i,j)]$ g'. This can be computed as $\texttt{Next}^i(\texttt{Next}^0 g' \wedge \ldots \wedge (\texttt{Next}^{(j-i)} g'))$. Evaluating this directly is too inefficient (since lots of redundant work will be done). The following approach computes g' exactly once and then provides appropriate circuitry to combine this value produced at various times.

   If the tuple associated with g' is (i_1, t, A_1, M_1), where $|M_1| = m_1$ then the new tuple associated with g is $(|M'|, t+1, A', M')$ where:

   - $M' = \tilde{\varrho}(M, BB_B((\lambda x, y.x \wedge y), (j-i)))\langle i_1 \rangle$

   A smaller, more efficient testing machine can be built provided that `Global` operator is not nested within another temporal operator. In this case, suppose the tuple associated with g is (i_1, t_1, A_1, M_1), where $|M_1| = m_1$. The new tuple is $(|M'|, t_1 + 1, A', M')$ where:

   - $M' = \tilde{\varrho}(M, M'')\langle i_1 \rangle$
   - $M'' = \langle \mathcal{C}^2, \mathbf{Y} \rangle$
   - if $t = \mathbf{Y}(s)$ then:

   $$t_j = \begin{cases} \perp & \text{when } j = 1. \\ s_1 \wedge s_2 & \text{when } j = 2. \end{cases}$$

   - $A' = A_1 \wedge (\texttt{Next}^t_1[m_1 + 1] = 1)$.

6. g = `Exists` $[(i,j)]\, g'$. This is analogous to the `Global` case, and can be computed as $\texttt{Next}^i(\texttt{Next}^0 g' \vee \ldots \vee (\texttt{Next}^{(j-i)} g'))$. All the remarks pertaining to the `Global` operator apply to the `Exists` operator – the difference is that instead of conjunction being

[3]Recall that M is the underlying machine.

applied, disjunction is applied. This shows that De Morgan's laws have a direct correspondence in the testing machines.

7. The bounded strong until, weak until, and periodic operators are all derived operators (see Definition 3.8). A straight-forward approach to model-checking these operators is model-check their more primitive definitions. For all three, smaller and more efficient machines are possible too.

There are competing threads here—the more operators the easier it is for a verifier to express properties, but with the wider choice comes the cost of greater complexity for the verifier. Constructing testing machines for the derived operators using the primitive definitions is not the most efficient approach: if the operators are going to be used, optimised testing machines should be constructed; but, if they are not going to be used the verification system will be more complicated that it needs to be.

There are two further types of optimisations which could be done. Testing machines are not canonical—there are different ways with different complexities of evaluation. The rewriting of formulas could yield improvement. The other issue has been discussed already: in some cases the testing machine needed for a formula depends on whether that formula is embedded within other temporal operators. If a formula stands by itself, then its satisfaction can checked by examining one component of the testing circuit at one instant in time. However, if the formula is embedded within some of the temporal operators, then we need to know the satisfaction of the formula at a number of instants in time.

Appendix C

Program listing

C.1 FL Code for Simple Example 1

```
let  c_size = bit_width;
let  bwidth = ’ c_size;
let  i = IVar  "i";  let  j = IVar  "j";  let  k = IVar  "k";
let  l = IVar  "l";
let GND = "GND";

let  a = Var "a";

let A = "a";    let B = "b";   let C = "c";
let D = "d";    let E = "e";   let FNode = "f";

let GlobalInput = ((A ISINT i)_&_(B ISINT j)_&_(C ISINT k)) FROM 0 TO 100;
let list1   = [("i", 1 upto 8), ("j", 9 upto 16), ("k", 17 upto 24)];
let varmap1 = BVARS list1;
let varmap2 = BVARS (("a", [25]):list1);

let A1 = GlobalInput;
let C1 = D ISBOOL (i ’> j) FROM 10 TO 100;
let T1 = VOSS varmap1  (A1 ==>> C1);

let A2 = GlobalInput _&_ ((D ISBOOL a) FROM 10 TO 100);
let C2 = GlobalInput _&_
         ((E ISINT i WHEN a) _&_ (E ISINT j WHEN (Not a)) FROM 20 TO 100);
let T2 = VOSS varmap2  (A2 ==>> C2);

let A3 = E ISINT l  _&_ C ISINT k   FROM 20 TO 100;
let C3 = FNode ISINT (l ’+ k) FROM 50 TO 100;
let T3 = VOSS varmap1  (A3 ==>> C3);

let proof =
     let G1 = SPTRANS [] T1 T2  in
     let G2 = SPTRANS [] G1 T3  in
       G2;
```

C.2 FL Code for Hidden Weighted Bit

```
let N = bit_width;
let x = IVar "x";                  let j = IVar "j";
let InputNode  = "InputNode";      let CountNode  = "CountNode";
let BufferNode1= "buffer1";        let BufferNode2= "buffer2";
let Chooser    = "chooser";        let Result     = "result";
let Error      = "error";


let varmap1 = BVARS([("x",1 upto N)]);

let  BufferTheorem = VOSS varmap1
                      ((InputNode ISINT x FROM 0 TO 1000)
                         ==>> ((BufferNode1 ISINT x) _&_
                               (BufferNode2 ISINT x) FROM 5 TO 1000));


letrec add_bits x num = x = N => BIT2 ('N) num
                      | (BIT2 ('x) num) '+ (add_bits (x+1) num);

letrec   count_of num =  add_bits 1 num;
let CounterGoal = (BufferNode1  ISINT x FROM 0 TO 990) ==>>
                  (CountNode    ISINT (count_of x) FROM 400 TO 990);
let CounterTheorem = VOSS varmap1 CounterGoal;

let stage1  = CONJUNCT BufferTheorem
                       (AUTOTIME [] BufferTheorem CounterTheorem);

let seg x  =  BWID ('Nbit) x;
let kthBit k var = (BIT2 ('k) var) '= (BIT2 ('1) ('1));

letrec case_analysis var j =
       letrec case k =
        k=1 => Result ISBOOL (kthBit k var) WHEN (j '= (seg ('k)))
             | (Result ISBOOL (kthBit k var) WHEN (j '= (seg ('k))))  _&_
               (case (k-1) ) in
           case N;

infix 3 ISBOOL_VEC;
letrec ISBOOL_VEC [x] [y]   = x ISBOOL y    /\
       ISBOOL_VEC (x:rx) (y:ry) = (x ISBOOL y) _&_
                               (rx ISBOOL_VEC ry);

let  ChooserGoal=
     ((CountNode   ISINT j FROM 0 TO 400) _&_
      (BufferNode2 ISINT x FROM 0 TO 400))
          ==>>
        (((case_analysis x j) _&_
```

```
          (Error ISBOOL (seg('0) '= j)))
                       FROM 300 TO 400);

let ChooserTheorem =  VOSS varmap2 ChooserGoal;

let Proof = ALIGNSUB [] stage1 ChooserTheorem;
```

C.3 FL Code for Carry-Save Adder

```
let A = Nnode "A";  let B = Nnode "B";   let C = Nnode "C";
let D = Nnode "D";  let E = Nnode "E";

let a = Nvar "a";   let b = Nvar "b";   let c = Nvar "c";

let bdd_order = order_int_1 [b, c, a];

let     range       =       (bit_width-1)--0;

let     sum_lhs     =       (D+E)<<range>>;
let     sum_rhs     =       (a+b+c)<<range>>;

let     Ant1    =       ((A == a)??) and ((B == b) ??) and ((C == c)??);
let     Con1    =       NextG 3 ( (sum_lhs == sum_rhs) ??);

let T1 = prove_voss bdd_order adder Ant1 Con1;
```

C.4 FL Code for Multiplier

```
// miscellaneous
let high_bit = entry_width - 1; // 0..entry_width-1
let max_time = 800;
let out_time = 3;

//----------------  Node, variable declarations

let     A = Nnode AINP;
let     B = Nnode BINP;
let     RS i = Nnode (R_S i);
let     RC i = Nnode (R_C i)<<(high_bit-1)--0>>;
let     TopBit i = Nnode (R_C i)<<high_bit>>;

let     a   = (Nvar "a")<<(entry_width-1)--0>>;
let     b   = (Nvar "b")<<(entry_width-1)--0>>;
let     c   = Nvar "c";
let     d   = (Nvar "d")<<(high_bit-1)--0>>;
```

```
let     partial {n :: int}  = c  <<(n+high_bit)--0>>;

// BDD variable ordering for each stage of multiplier

let     m_bdd_order {n::int}   =
              n = 0
                =>  order_int_1 [b, a]
                 |  n=entry_width
                    => order_int_1 [partial n, d]
                       |  order_int_1 [b<<n>>, a, partial n, d];

let     zero_cond i = ((TopBit i)==('0))??;

let     interval  n  =
             n <= entry_width
               => [('(n*out_time), 'max_time)]
                | [('(n*out_time+2*entry_width), 'max_time)];

let     InputAnts = Always (interval 0)
                          (( (A == a) ?? ) and  ( (B == b) ??));
let     OutputCons =
             let lhs = RS entry_width in
             let rhs = (a * b)<<(2*entry_width-1)--0>> in
                   Always (interval (entry_width+1)) ((lhs==rhs)??);

// Antecedent for row n of the multiplier
let     MAnt {n::int}   =
                n = 0
                  =>  Always (interval 0)
                              ( ( (A == a)??) and
                                ( (B<<n>> == b<<n>>)?? )
                              )
                   |  Always (interval n)
                               (( (A == a)??) and
                                ( (B<<n>> == b<<n>>)?? ) and
                                ( (RS (n-1)  == (partial (n-1)))??) and
                                ( (RC (n-1)  == d)??)   and
                                ( zero_cond (n-1)) );

// Consequent of row n of the multiplier
let res_of_row n =
      let power n = Npow ('2) ('n)  in
      let lhs = (RS n) + (power (n+1))*(RC n) in
      let rhs =
        n=0
         => a * b <<0>>
          | ((partial (n-1))+(power n) * d) + (power n)*a *(b <<n>>) in
            ((lhs == rhs)??);
```

```
let     Con_of_stage n =
          let power n = Npow ('2) ('n)  in
          let lhs = (RS n) + (power (n+1))*(RC n) in
          let rhs =  a * b<<n--0>> in
             Always (interval (n+1))
                 ((lhs == rhs)?? and (zero_cond n));


let     MCon {n::int}  =  Always  (interval (n+1))
                            ((res_of_row n ) and (zero_cond n));

let     Mthm n =
          let bdd_order = (m_bdd_order n) in
          let ant       = MAnt n in
          let con       = MCon n in
            prove_voss bdd_order multiplier ant con;

let     preamble_thm =
         let start = Mthm 0 in
          Precondition InputAnts start;

letrec  do_proof_main_stage  n m previous_step  =
           let curr  = Mthm n in
           let curr' = GenTransThm previous_step curr in
           let current  = Postcondition (Con_of_stage n) curr' in
               n =  m
                  => current
                   | do_proof_main_stage (n+1) m current;

let    main_stage  = do_proof_main_stage 1 high_bit preamble_thm;

let   adder_proof  =
     let post_ant_cond =
     (( (RS high_bit)     == (partial high_bit))??) and
     (( (RC high_bit)     == d)??) and
     (( (TopBit high_bit) == ('0))??)
       in
     let post_ant = Always (interval entry_width) post_ant_cond
in
     let power  = Npow ('2) ('entry_width)  in
     let rhs    = ((partial high_bit) + power * d)<<(bit_width-1)--0>> in
     let post_con_cond = ((RS entry_width) == rhs)?? in
     let post_con      = Always (interval (entry_width+1))
 post_con_cond in
       prove_voss (m_bdd_order entry_width) multiplier post_ant post_con;

let    proof = GenTransThm main_stage adder_proof;
```

C.5 FL Code for Matrix Multiplier Proof

```
// miscelleneous

let high_bit = entry_width - 1; // 0..entry_width-1
let max_time = entry_width < 10 => 100 | 10*entry_width;
let clock_time = max_time;   // half a clock cycle
let out_time = 3;
//----------------
let   prove_result = prove_voss_fsm;
let   prove_result_static = prove_voss_static;
//----------------  Node, variable declarations
//----- global
let   Clock  = Bnode CLK;

//----- individual cells
let   A u v = Nnode (AINP u v);    let   B u v = Nnode (BINP u v);
let   IN_C  u v = Nnode (C_Inp u v); let   OUT_C u v = Nnode (C_Out u v);

let   M = make_fsm sys_array;
let   RS u v i = Nnode (R_S u v i);
let   RC u v i = Nnode (R_C u v i)<<(high_bit-1)--0>>;
let   TopBit u v i = Nnode (R_C u v i)<<high_bit>>;


let   a   = (Nvar "a")<<(entry_width-1)--0>>;
let   b   = (Nvar "b")<<(entry_width-1)--0>>;   let   c   = Nvar "c";
let   d   = (Nvar "d")<<(high_bit-1)--0>>;    let   e   = Nvar "e";


let   partial {n :: int}  = e  <<(n+high_bit)--0>>;

// BDD variable ordering for each stage of multiplier
let   m_bdd_order {n::int}   =
        n = 0
        =>  order_int_1 [b, a]
         |  n=entry_width
          => order_int_1 [partial n, d]
             |  order_int_1 [b<<n>>, a, partial n, d];

// timings
let   DuringInterval n f = During (n*out_time, max_time) f;

letrec  ClockAnt n =
     let range = 0 upto (n-1) in
     let false_range = map (\x.(('(2*x*clock_time),
                  '(2*x*clock_time+clock_time-1))))
                 range in
     let true_range  = map (\x.('(2*x*clock_time+clock_time),
```

```
                  '(2*(x+1)*clock_time-1)))
                 (butlast range) in
        (Always false_range ((Clock == Bfalse)??)) and
        (Always true_range  ((Clock == Btrue )??));

let     InputAnts u v = DuringInterval 0
                           ((A u v '= a) and  (B u v '= b));

let     zero_cond u v i = TopBit u v i   '= ('0);

// Antecedent for row n of the multiplier
let     MAnt u v {n::int}   =
     n = 0
       =>    DuringInterval 0
    ( ( A u v '= a ) and
      ( (B u v)<<n>> '= b<<n>>))
|    DuringInterval n
      (( A u v '= a) and
       ( (B u v)<<n>>  '= b<<n>>) and
       ( RS u v (n-1)  '= (partial (n-1))) and
       ( RC u v (n-1)  '= d)      and
       ( zero_cond u v (n-1));

// Consequent of row n of the multiplier
let     res_of_row u v n =
       let power n = Npow ('2) ('n)  in
       let lhs = (RS u v n) + (power (n+1))*(RC u v n) in
       let rhs = n=0
  => a * b <<0>>
   | ((partial (n-1))+(power n)* d) + (power n)*a*(b <<n>>)in
     lhs '= rhs;

let     Con_of_stage u v n =
          let power n = Npow ('2) ('n)  in
          let lhs = (RS u v n) + (power (n+1))*(RC u v n) in
          let rhs =  a * b<<n--0>> in
              DuringInterval (n+1)
                 ((lhs '= rhs) and (zero_cond u v n));

let MCon u v {n::int}  =  DuringInterval (n+1)
                           ((res_of_row u v n ) and (zero_cond u v n));
let     Mthm u v n =
          let bdd_order = (m_bdd_order n) in
          let ant       = MAnt u v n in
          let con       = MCon u v n in
            prove_result bdd_order M ant con;

let     preamble_thm u v =
         print (nl^"Doing preamble"^nl) seq
```

```
        let start = Mthm u v 0 in
        (start catch start) seq
         Precondition (InputAnts u v) start;

letrec  do_proof_main_stage  u v n m previous_step  =
          let curr  = Mthm u v n in
          let curr' = GenTransThm previous_step curr in
          let current  = Postcondition (Con_of_stage u v n) curr' in
          (print (nl^"   Doing M["^(int2str u)^", "^(int2str v)^
                     "]("^(int2str n)^")"^nl^nl) seq
            (current catch current))
            seq
             ( n =  m
                 => current
                  | do_proof_main_stage u v (n+1) m current);

let  main_stage u v  =
        do_proof_main_stage u v 1 high_bit (preamble_thm u v);

let   adders_proof u v  =
        let post_ant_cond =
               ( (RS u v high_bit)     '= (partial high_bit)) and
               ( (RC u v high_bit)     '= d) and
               ( (TopBit u v high_bit) '= ('0))
                 in
       let post_ant = DuringInterval entry_width post_ant_cond in
       let power = Npow ('2) ('entry_width)  in
       let rhs   = ((partial high_bit)+ power*d)<<(bit_width-1)--0>> in
       let post_con_cond = (RS u v entry_width) '= rhs in
       let post_con      =
                During (entry_width*(out_time+2), clock_time)
                           post_con_cond in
       let bdd_order = m_bdd_order entry_width in
       (print "Doing adder" seq (post_con catch post_con)) seq
         prove_result bdd_order M post_ant post_con;

let    cell_out_time = [('(2*clock_time), '(3*clock_time))];
let    register_proof u v =
           let c_ant  = (((RS u v entry_width) '=  (partial entry_width))
                         and ((IN_C u v) '= c)) in
           let c_ant' =
                (ClockAnt 2) and
                (During (entry_width*(out_time+2), clock_time)
                        c_ant) in
           let c_rhs  = (partial entry_width) + c in
           let c_con  = (OUT_C u v) '= c_rhs in
           let c_reg  = prove_result
                          (order_int_1 [c, partial entry_width])
                          M
```

```
                          c_ant'
                          (Always cell_out_time c_con)
              in
               ((print "Doing register") seq c_con catch c_con)
              seq
                  c_reg;


// one_proof u v: proves that the (u,v)-th cell works
// correctly
let    one_proof u v =
        // Prove that multiplier parts work (unclocked)
        let m_stage = main_stage u v in
        (m_stage catch m_stage) seq
        // take into account clocking and the partial sum input
        let new_ants= InputAnts u v  and
                      (ClockAnt 2) and
                      (DuringInterval 0 (IN_C u v  '= c)) in
        let new_thm = Precondition new_ants m_stage in
        // show the adder part of the ceol works
        let a_proof = adders_proof u v in
        (a_proof catch a_proof) seq
        // Add clocking to the adder proof
        let comp_proof =  GenTransThm new_thm a_proof in
        // Show that the registers work
        let r_proof    =  register_proof u v in
          ((r_proof catch r_proof)
            seq
            // stick them all together
        let result = (normaliseCon (GenTransThm comp_proof r_proof)) in
            result);


letrec  make_cell_row_list p_proc u v =
           v=array_depth
               => []
                | let res = p_proc u v in
                  print (snd (time res)) seq
                     (res seq (res:(make_cell_row_list p_proc u (v+1))));

letrec  make_proof_list p_proc u =
u = array_width
    => []
     | (make_cell_row_list p_proc u 0):
(make_proof_list p_proc (u+1));

let    cell_proof_list = make_proof_list one_proof 0;

// Show that the cells also progate their A and B inputs
```

```
let    one_proof_propagateA u v =
        let ants = (DuringInterval 0 (A u v '= a)) and (ClockAnt 2) in
        let ab_con = A u (v+1) '= a in
        let ab_reg =
             prove_result (m_bdd_order 0) M ants
                                         (Always cell_out_time ab_con) in
           ab_reg;

let    one_proof_propagateB u v =
        let ants = (DuringInterval 0 ((B u v) '= b)) and (ClockAnt 2) in
        let ab_con = (B (u+1) v)  '= b in
        let ab_reg =
             prove_result (m_bdd_order 0) M ants
                                         (Always cell_out_time ab_con) in
           ab_reg;

let   Apropagate_proof_list = make_proof_list one_proof_propagateA 0;
let   Bpropagate_proof_list = make_proof_list one_proof_propagateB 0;

let       cell_proof u v = el (v+1) (el (u+1) cell_proof_list);

let       Apropagate_proof u v = el (v+1) (el (u+1) Apropagate_proof_list);
let       Bpropagate_proof u v = el (v+1) (el (u+1) Bpropagate_proof_list);

let     em_thm = ([],[],[]);

//---------------------------------------------------------------
// The *_proof_list contains all the proofs that the individual
// components of the hardware work correctly. The rest of the
// proof shows that when connected together they produce
// the right matrix multiplication result

letrec  InsertActiveTheorem addfn
            ({u::int},{v::int},{new_thm::theorem}) [] =
                                          [(u, [(v, addfn new_thm em_thm)])]
     /\ InsertActiveTheorem addfn (u,v,new_thm)
                               ((au, alist):brest) =
    letrec PutActiveTheoremIn ({v::int}, {new_thm::theorem}) []
      = [(v, addfn new_thm em_thm)]
/\ PutActiveTheoremIn (v, new_thm) ((av, avlist):vrest) =
       v = av
 =>  (av, addfn new_thm avlist):vrest
  |  (av, avlist):
 (PutActiveTheoremIn (v, new_thm) vrest)
      in u = au
      => (au, PutActiveTheoremIn (v, new_thm) alist):brest
        | (au, alist):
   (InsertActiveTheorem addfn(u,v,new_thm) brest);
```

```
letrec  RetrieveTheorem {u::int} {v::int}    [] =      ([],[],[])
     /\ RetrieveTheorem  u v  ((au, alist):brest) =
             letrec GetActiveTheorem v []  = ([],[],[])
                 /\ GetActiveTheorem v ((av, avlist):vrest) =
                        v = av
                          =>  avlist
                           |  GetActiveTheorem  v vrest  in
                     u = au
                       =>  GetActiveTheorem v alist
                        |  RetrieveTheorem  u v brest;

let    InsertActiveList add_fn thm_list current =
          itlist (\x.\y.InsertActiveTheorem add_fn x y) thm_list current;


//   VERIFICATION CONDITION
///---------------------   Input specifications

let  setInput InpNode {u :: int} {v :: int} {i::int} {n_val :: N} =
             let input =    During (i*2*clock_time, (i+1)*2*clock_time-1)
       (InpNode u v '= n_val)
    in  (u, v, Identity input);

let   a11 = Nvar "a11"; let   a12 = Nvar "a12"; let   a13 = Nvar "a13";
let   a14 = Nvar "a14"; let   a21 = Nvar "a21"; let   a22 = Nvar "a22";
let   a23 = Nvar "a23"; let   a24 = Nvar "a24"; let   a31 = Nvar "a31";
let   a32 = Nvar "a32"; let   a33 = Nvar "a33"; let   a34 = Nvar "a34";
let   a41 = Nvar "a41"; let   a42 = Nvar "a42"; let   a43 = Nvar "a43";
let   a44 = Nvar "a44"; let   b11 = Nvar "b11"; let   b12 = Nvar "b12";
let   b13 = Nvar "b13"; let   b14 = Nvar "b14"; let   b21 = Nvar "b21";
let   b22 = Nvar "b22"; let   b23 = Nvar "b23"; let   b24 = Nvar "b24";
let   b31 = Nvar "b31"; let   b32 = Nvar "b32"; let   b33 = Nvar "b33";
let   b34 = Nvar "b34"; let   b41 = Nvar "b41"; let   b42 = Nvar "b42";
let   b43 = Nvar "b43"; let     b44 = Nvar "b44";

let     the_inputs =
           //   a0   a1  a2    a3     b0   b1   b2   b3
           [ ([ '0,  '0,  '0,  '0], [ '0,  '0,  '0,  '0]),   //0
             ([ '0,  '0,  '0,  '0], [ '0,  '0,  '0,  '0]),   //1
             ([ '0,  '0,  '0,  '0], [ '0,  '0,  '0,  '0]),   //2
             ([ '0,  '0,  '0,  '0], [ '0,  '0,  '0,  '0]),   //3
             ([ '0, a11,  '0,  '0], [ '0, b11,  '0,  '0]),   //4
             ([ '0,  '0, a21,  '0], [ '0,  '0, b12,  '0]),   //5
             ([a12,  '0,  '0, a31], [b21,  '0,  '0, b13]),   //6
             ([ '0, a22,  '0,  '0], [ '0, b22,  '0,  '0]),   //7
             ([ '0,  '0, a32,  '0], [ '0,  '0, b23,  '0]),   //8
             ([a23,  '0,  '0, a42], [b32,  '0,  '0, b24]),   //9
             ([ '0, a33,  '0,  '0], [ '0, b33,  '0,  '0]),   //10
             ([ '0,  '0, a43,  '0], [ '0,  '0, b34,  '0]),   //11
```

```
             ([a34,  '0,  '0,  '0], [b43,  '0,  '0,  '0]),   //12
             ([ '0, a44,  '0,  '0], [ '0, b44,  '0,  '0]),   //13
             ([ '0,  '0,  '0,  '0], [ '0,  '0,  '0,  '0]),   //14
             ([ '0,  '0,  '0,  '0], [ '0,  '0,  '0,  '0])];//15;


//---------------   Output specifications
let     timeForOutputs =
             //   1   2   3   4
             // ---------------
              [ [ 6,  7,  8,  9],         //  1
                [ 7,  9, 10, 11],         //  2
                [ 8, 10, 12, 13],         //  3
                [ 9, 11, 13, 15]          //  4
              ];

let     outputFor row col = el col (el row timeForOutputs);

let  InputForCells _ _ = [];
let  addfirst  x (a,b,c) = (x:a,b,c);
let  addsecond x (a,b,c) = (a,x:b,c);
let  addthird  x (a,b,c) = (a,b,x:c);

let  InputAtStage n  the_lists =
       val (avals, bvals) = el (n+1) the_inputs in
       let left_list = map (\x.setInput A {x::int} 0 n (el (x+1) avals))
                    (0 upto (array_depth-1))
                  in
       let right_list =
              map (\x.setInput B 0 x n (el (x+1) bvals))
                    (0 upto (array_width-1)) in
       let down_list  =
              (map (\x.setInput IN_C (array_depth-1) {x::int} n ('0))
                  (0 upto (array_width-1)))@
              (map (\x.setInput IN_C x (array_width-1) {n::int} ('0))
                  (0 upto (array_depth-2))) in
         let res1 = InsertActiveList addfirst left_list the_lists in
         let res2 = InsertActiveList addsecond right_list res1 in
            InsertActiveList addthird down_list res2;

let        start_step = InputAtStage 0 [];
let        this_step  = start_step;         let num_step   = 0;

let        PropagateVal addfn row col ok1 {ok2::bool}  res old_list =
             ok1 AND ok2
                  => InsertActiveTheorem addfn (row, col, res) old_list
                   | old_list;

let        PropagateRes row col all res res_l =
```

```
                let c_index = "C"^(num2str(array_width-col-1+row)) in
                  all AND (row*col = 0)
                   =>  (c_index, res, (row, col)): res_l
                    |  res_l;

letrec     ProcessStageRow n {row::int}   []  so_far   = so_far
   /\      ProcessStageRow n row ((col, colthms):rest)
                                              (prop_list, res_l) =
    let make_step (a, b, c) =
       let ok  a n  = length a > n in
       let all_thms = (Identity(ClockAnt ((n+1)*2))):(a@b@c) in
       let ab_inps  = (a@b) in
       let all      = ok all_thms 3 in
       let curr_gen = all
  => Conjunct [cell_proof row col, Apropagate_proof row col,
       Bpropagate_proof row col] |
  length ab_inps = 2
  => Conjunct
     [Apropagate_proof row col,Bpropagate_proof row col] |
  ok a 0
  => Apropagate_proof row col
   | Bpropagate_proof row col in
       let curr_thm = Transform (TimeShift (2*n*clock_time))
       curr_gen in
       let inps  = Conjunct all_thms in
       let res   = normaliseCon (GenTransThm inps curr_thm) in
       let new_l = PropagateVal addfirst
 row (col+1) (col<(array_width-1))
 (ok a 0) res prop_list in
       let new_r = PropagateVal addsecond
      (row+1) col (row<(array_depth-1))
      (ok b 0) res new_l in
       let new_d = PropagateVal addthird
      (row-1) (col-1) ((row*col) > 0)
      all res new_r in
       let new_rl= PropagateRes row col all res res_l
     in
 empty ab_inps
  =>    (prop_list, res_l)
   |    (new_d, new_rl)
    in
 ProcessStageRow n row rest (make_step  colthms);

letrec     ProcessStageProof n [] so_far   =      so_far
    /\     ProcessStageProof n ((row,rowthms):rest) so_far =
              let current = ProcessStageRow n row rowthms so_far in
                (print ("Doing row "^(int2str row)^nl)) seq
                  (current catch current) seq
                  ProcessStageProof n rest current;
```

```
let  do_step n start_step =
       letrec perform m curr_step =
              let current = ProcessStageProof m (InputAtStage m curr_step)
                                                ([], []) in
              (print ("Performing step "^(int2str m)^nl^nl)) seq
              (current catch current) seq
              m = n
                => [snd current]
                 | (snd  current):(perform (m+1) (fst current)) in
          perform 0 start_step;

let  output_list  = do_step 15 [];

// present results
let  ShowRes t res_list = el (t+1) res_list;

let  Show t node =
      let res = ShowRes t output_list in
          find (\(x,y,a,b).(x=node) AND ((a*{b::int}) = 0)) res;

let  OutputOfArray row col =
       let strip (Always r f) = f in
       val (a, th, b, c) =  Show (outputFor row col)
                                 ("C"^(num2str(3+row-col))) in
            strip (con_of th);

letrec PrintRowOutput row col =
        (col = array_width+1)
            =>  nl^nl
             |  ("("^(int2str row)^" ,"^(int2str col)^") :"^
                        (el2str (OutputOfArray row col))^nl)
                 ^(PrintRowOutput row (col+1));

letrec PrintOutput row =
         row = array_depth + 1
            =>  nl
             |  (PrintRowOutput row 1) ^ (PrintOutput (row+1));
```

Index