Model Checking Partially Ordered State Spaces

S. Hazelhurst and C.-J. H. Seger
Integrated Systems Design Laboratory
Department of Computer Science
University of British Columbia
Vancouver, B.C., Canada V6T 1Z4

21 July 1995

Abstract

The state explosion problem is the fundamental limitation of verification through model checking. In many cases, representing the state space of a system as a lattice is an effective way of ameliorating this problem. The partial order of the state space lattice represents an information ordering. The paper shows why using a lattice structure is desirable, and why a quaternary temporal logic rather than a traditional binary temporal logic is suitable for describing properties in systems represented this way. The quaternary logic not only has necessary technical properties, it also expresses degrees of truth. This is useful to do when dealing with a state space with an information ordering defined on it, where in some states there may be insufficient or contradictory information available. The paper presents the syntax and semantics of a quaternary valued temporal logic.

Symbolic trajectory evaluation (STE) [32] has been used to model check partially ordered state spaces with some success. The limitation of STE so far has been that the temporal logic used (a two-valued logic) has been restricted, whereas a more expressive temporal logic is often useful. This paper generalises the theory of symbolic trajectory evaluation to the quaternary temporal logic, which potentially provides an effective method of model checking an important class of formulas of the logic. Some practical model checking algorithms are briefly described and their use illustrated. This shows that not only can STE be used to check more expressive logics in principle, but that it is feasible to do so.

Keywords: symbolic trajectory evaluation, quaternary logic, model checking, temporal logic, bilattices

1 Introduction

Model-checking is a well-known automatic verification method that can determine whether a system has a certain set of properties. The nature of the model that represents the system and the type of logic used to express properties are choices open to the verifier. Different choices have

This research was supported by operating grant OGPO 109688 from the Natural Sciences and Engineering Research Council of Canada, a fellowship from the B.C. Advanced Systems Institute, and by equipment grants from Sun Microsystems Canada Ltd. and Digital Equipment of Canada Ltd. Support was also received from the Semiconductor Research Corporation. This paper has been submitted for review for publication, and so may be subject to copyright.

different advantages and disadvantages — generally the more powerful and expressive the formalism the easier it is to represent behaviour and properties, and the higher the computational cost in performing verification.

An underlying problem of all model checking approaches is the state explosion problem — the number of states of a system increases exponentially with respect to the number of components. One solution to the problem is representing the state space as a partial order. This allows partial information to be used effectively and reduces the size of representations of models.

The purpose of this paper is to present a general temporal logic suitable for model checking partially-ordered state spaces. The basis of this logic is a quaternary valued logic for expressing properties. Section 2 justifies the use of partially-ordered state space and why the use of partial information leads to the use of a quaternary-valued logic rather than a two-valued logic.

Section 3 introduces the quaternary logic. Using this as a foundation, Section 4 defines the syntax and semantics of an extended temporal logic, TL. Formulas of the logic are built up from simple basic blocks, and the meaning of the formulas is defined by a satisfaction relation between formulas and sequences of the underlying state space.

Symbolic trajectory evaluation (STE) is a model checking algorithm which has successfully exploited partially-ordered state spaces representations [32]. Moreover, STE does not use fix-point computations of the next-state relation to determine sets of reachable states (symbolic simulation of the model is used to explore the model's behaviour). Although these properties of STE have important performance advantages, the price paid for the advantages is that STE algorithms presented so far use a restricted temporal logic to express properties of systems (for example, negation and disjunction are not supported). One reason for using this restricted logic (called trajectory formulas) is that there are efficiency benefits to be gained by sacrificing expressiveness. A fundamental reason, however, is that the technical framework used in previous work is unsatisfactory for a richer logic. As a more expressive logic is often useful or necessary, and can be efficiently used, this is an issue which needs exploration.

Although there may be several model checking algorithms for partially-ordered state spaces, since in a more restricted setting, symbolic trajectory evaluation proved to be a successful method, this paper explores how STE can be extended to support the richer logic. Section 5 presents a decision procedure for a class of formulas of the extended logic. This decision procedure is a generalised form of symbolic trajectory evaluation presented in earlier work [7, 32].

This STE algorithm is used as the basis for the decision procedures for TL and Section 6 briefly describes a number of STE-based model-checking algorithms. Section 7 gives two example verifications, and section 8 compares the logic, method and results to other work. Section 9 concludes the paper.

2 Motivation

2.1 Model Checking and state explosion

In traditional symbolic model checking described by Burch *et al.*, a number of boolean variables are used to describe the state of the system [9]. For example, in a circuit, each state holding component of the circuit could be represented by a boolean variable or value; the state of the circuit is then naturally represented as the cross-product of the states of the components.

Suppose at time t, a system is in state $\langle x_0, \ldots, x_n \rangle$, and at time $t+1$ the system is in state

$\langle x'_0, \ldots, x'_n \rangle$. The behaviour of the system is represented by a next state relation, R, which describes how the x_i and x'_i are related. By suitable manipulation of R it is possible to determine the set of reachable states and to perform model checking. See [9, 13] for details.

Although there has been considerable success with this approach, there are limitations on the size of the circuit that can be dealt with effectively. The underlying technology — ordered binary decision diagrams — can only deal with hundreds of boolean variables. Even moderate size circuits can be beyond model checking using this approach if the circuit has a non-trivial data path, especially when an accurate model of time is used. The next state relation may be too large, or even where the next state relation can easily be computed and stored, computing the set of reachable states may not be tractable.

Dealing with this problem is one of the key issues in model checking. A number of suggestions have been made for dealing with the problem (e.g., see discussion in [9, 11, 27]).

The underlying motivation of symbolic trajectory evaluation is that incomplete information is often sufficient for successful model checking. Computing the exact state of large parts of the circuit for successive time instants may be irrelevant for checking many formulas. STE allows a 'don't care' value to be given to state holding elements which the verifier believes are not relevant to the computation[1]. This can reduce by orders of magnitude the size of the data structures needed to represent the behaviour of the circuit.

2.2 Partially-ordered state spaces

Formally, the state space of the model is represented by $\langle \mathcal{S}, \sqsubseteq \rangle$, a complete lattice under the partial order $\sqsubseteq$, and the behaviour of the model is represented by the next-state function $\mathbf{Y}$ which is monotonic with respect to the partial order. Sequences of states can also be (partially) ordered by extending the partial order on states element-wise to sequences.

Consider 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 (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. The partial ordering of the lattice is an information ordering: the higher up in the ordering we are in the model, 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. $\mathbf{Y}$ can be extended to operate on all states of the lattice. State s_9 represents a state that has contradictory information.

For circuit models, the lattice representing the state space has a natural and intuitive basis. Each state holding component has a value drawn from the set $\{\mathbf{X}, \mathbf{L}, \mathbf{H}, \mathbf{Z}\}$. $\mathbf{L}$ and $\mathbf{H}$ represent low and high voltages; $\mathbf{X}$ represents an unknown value; and $\mathbf{Z}$ represents an overconstrained value. These values can be represented efficiently. There is a natural partial order between these values: $\mathbf{X}$ contains less information than $\mathbf{L}$ and $\mathbf{H}$ which in turn have less information than $\mathbf{Z}$. The state of the circuit is the cross-product of the states of the individual state components.

[1] In practice, this is done the other way round. State holding components which are relevant are given boolean values, and the rest are automatically assigned the 'don't care' values

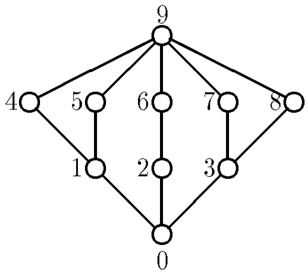

Figure 1: Example Lattice State Space

Through the judicious use of **X** 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. Since STE computes the next state function and the reachable states using symbolic simulation, this type of mathematical structure can be effectively used.

2.3 Motivation for a quaternary logic

The purpose of model checking is to determine whether a model has a certain property. Ideally, we would like our verification method to 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 we can express 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.

Furthermore, state s_9 plays an important role too. A state like s_9 represents states about which inconsistent information is known. Although such states don't occur in 'reality', they are some times 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. 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 previous work using STE [32], a simple two valued temporal logic was used. No distinction was made between a property being false and there being too little information to know whether it is true. Among others, disjunction and negation of formulas was not allowed. Although its expressiveness was limited, it was expressive enough for many problems (e.g., see [3, 17]). The advantage of the simplicity of this logic is that the model-checking technique is very efficient, and the theory of compositionality of results simple [25]. A particular advantage of this temporal logic is that formulas have the useful property that for each formula there exists a unique weakest sequence of states satisfying the formula. This leads to a very efficient model-checking algorithm.

The disadvantage is that the logic is not as expressive as other logics. For many systems, the simple logic is adequate. However, there are some properties which, while possible to cast as trajectory formulas, would be far easier to describe in a richer logic. And there are useful properties which cannot be expressed as trajectory formulas. Examples of where the richer logic is needed include the verification of an IEEE floating point multiplier [1] and current work of ours verifying some circuits in the IFIP WG10.2 Benchmark-Circuits Suite for Hardware-Verification. Simple examples will be given later in this paper. That STE can support a richer logic has been known for some time, although no work has been published on the topic. This paper examines how STE can be used to model check a richer logic.

Representing the state space as a lattice means that checking whether a property holds of a state can yield four results: true (the property does hold), false (the property does not hold), unknown, and inconsistent (the property holds and does not hold). A two-valued logic cannot express this, whereas a four-valued logic can. This is a major motivation for moving to a four-valued domain.

This important philosophical point is supplemented by an important pragmatic one. The introduction of negation into a two-valued temporal logic violates monotonicity constraints — a four valued logic *has* the right technical properties for a richer logic. The next section introduces the mathematical foundations for such a logic. As shown later, this richer logic can be checked using algorithms based on symbolic trajectory evaluation.

3 The quaternary logic $\mathcal{Q}$

This section describes the four-valued logic. The four values represent truth, falsity, undefined (or unknown) and overdefined (or inconsistent). Such a logic was proposed by Belnap, and has since been elaborated upon and different application areas discussed in a number of other works [21, 34]. This section first gives some mathematical background, based on [20, 30], and then definitions are given and justified.

An *interlaced 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, and 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 $\mathcal{Q} = \{\perp, \mathbf{f}, \mathbf{t}, \top\}$ where the partial orders are shown in Figure 2. $\mathbf{f}$ 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 $\mathcal{B} \subset \mathcal{Q}$). $\preceq$ and $\leq$ are of type $\mathcal{Q} \times \mathcal{Q} \to \mathcal{B}$. The partial order $\preceq$ represents an information ordering (on the truth domain), and the partial order $\leq$ represents a truth ordering. (Note, the symbol $\sqsubseteq$ is used for comparing *states* and the symbol $\preceq$ is used to compare *truth values*).

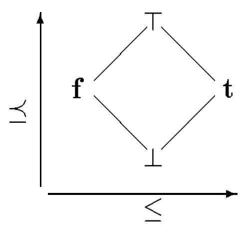

Figure 2: The bilattice $\mathcal{Q}$

Note that $\mathcal{Q}$ is a complete lattice with respect to each partial order, and that the natural distributivity laws hold with respect to meets and joins of the partial order. The technical properties

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

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

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

Figure 3: Conjunction, disjunction and negation operators for $\mathcal{Q}$

of $\mathcal{Q}$ make it suitable for model-checking partially-ordered state spaces.

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 [20].

These definitions are shown in Figure 3, and have the following pleasant properties.

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

The definition of $\mathcal{Q}$ is not without problems, but it is the 'classical' definition, and is convenient. Other definitions are possible too. In this paper, $\Longrightarrow$ is used to mean implication in the traditional, boolean sense; $\Rightarrow$ is reserved as a derived $\mathcal{Q}$-operator.

4 An Extended Temporal Logic

The extended temporal logic, TL, allows one symbolic formula to represent a large collection of scalar formulas concisely; this, together with the representation of the state space as a lattice makes model checking efficient for a class of formulas. The following sections introduce the scalar and symbolic versions of TL. A specialised version TL_n which is suitable for a circuit model is also described.

4.1 Scalar version of TL

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. $\mathcal{R} \subseteq \mathcal{S}$, 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. $\mathcal{R}$ must be a lower semi-lattice which is downward closed (if $x \in \mathcal{R}$, and $y \sqsubseteq x$ then $y \in \mathcal{R}$). $\mathbf{Y}: \mathcal{S} \to \mathcal{S}$ is a monotonic next state function. Let $\mathbf{X}$ be the least element in $\mathcal{S}$.

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 $s \sqsubseteq t \implies p(s) \preceq p(t)$ (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 paper.

Example 4.1 Take as an example, the state space $\mathcal{S}$ given in Figure 1. Define $g, h : \mathcal{S} \to \mathcal{Q}$ by:

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

Figure 4 depicts these definitions graphically. g and h are Q-predicates. The same state space and functions will be used in subsequent examples.

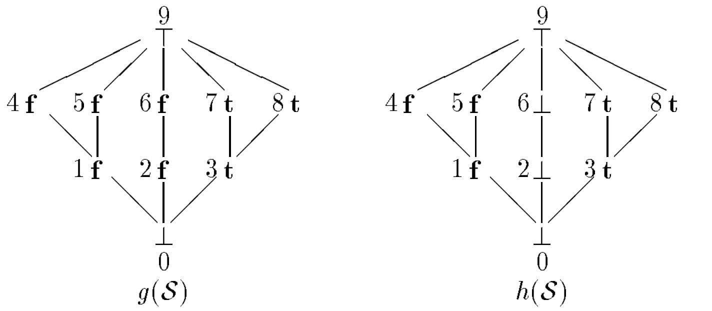

Figure 4: Definition of g and h

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 the weakest state for which $p(s) = q$.

Definition 4.1 $(s_q, q) \in \mathcal{S} \times \mathcal{Q}$ is a *defining pair* for a predicate g if $g(s_q) = q$ and $\forall s \in \mathcal{S}, g(s) = q$ implies that $s_q \sqsubseteq s$.

Example 4.2 $(s_3, \mathbf{t})$ 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 4.2 If $g : \mathcal{S} \to \mathcal{Q}$ then $D(g) = \{(s_q, q) \in \mathcal{S} \times \mathcal{Q} | (s_q, q) \text{ is a defining pair for } g\}$, is the *defining set* of g.

Example 4.3 ()
$D(g) = \{(s_0, \bot), (s_3, \mathbf{t}), (s_9, \top)\}$
$D(h) = \{(s_0, \bot), (s_1, \mathbf{f}), (s_3, \mathbf{t}), (s_9, \top)\}$

If a monotonic predicate has a defining pair for every element in its range, then its defining set uniquely characterises it (see Lemma 4.1). Such monotonic predicates are called *simple* predicates and form the basis of our temporal logic.

Definition 4.3 A monotone predicate $g: \mathcal{S} \to \mathcal{Q}$ is *simple* if $\forall q \in g(\mathcal{S}), \exists (s_q, q) \in D(g)$.

Example 4.4 In our example, h is simple since every element in the range of h has a defining pair. g is not simple since there is no defining pair $(s_{\mathbf{f}}, \mathbf{f})$. Informally, it is not simple since we can't use a single element of $\mathcal{S}$ to characterise the values for which $g(s) = \mathbf{f}$.

Note that simple predicates need not be surjective; we only require 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.

Lemma 4.1 If $g, h: \mathcal{S} \to \mathcal{Q}$ are simple, then $D(g) = D(h)$ implies that $g \equiv h$.

Proof: See Section A.1 □

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

Definition 4.4 Let G be the set of simple predicates.

We now use G to construct the temporal logic.

Definition 4.5 (The Scalar Extended Logic — TL) The scalar version of TL is defined by

$$\mathrm{TL} ::= \; G \mid \mathrm{TL}\ \texttt{and}\ \mathrm{TL} \mid \texttt{not}\ \mathrm{TL} \mid \texttt{Next}\ \mathrm{TL} \mid \mathrm{TL}\ \texttt{Until}\ \mathrm{TL}$$

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

Suppose g, h are TL formulas. Informally, if g is simple a sequence satisfies g if g holds of the initial state of the sequence. Conjunction has a natural definition. A sequence satisfies $\texttt{not}\, g$ 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 definition below, the definitions of conjunction and negation are the operations in the quaternary logic.

Notation: Let $\tilde{s} = s_0 s_1 s_2 \ldots$ be a sequence in $\mathcal{S}$. For convenience, let $\tilde{s}_i = s_i s_{i+1} \ldots$.

Definition 4.6 *Semantics of* TL

1. If $g \in G$ then $Sat(\tilde{s}, g) \; = \; g(s_0)$.
2. $Sat(\tilde{s}, g\ \texttt{and}\ h) \; = \; Sat(\tilde{s}, g) \wedge Sat(\tilde{s}, h)$
3. $Sat(\tilde{s}, \texttt{not}\, g) \; = \; \neg Sat(\tilde{s}, g)$

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

4. $Sat(\tilde{s}, \texttt{Next}\, g) \;=\; Sat(\tilde{s}_1, g)$
5. $Sat(\tilde{s}, g\, \texttt{Until}\, h) \;=\; \vee_{i=0}^{\infty}(Sat(\tilde{s}_0, g) \wedge \ldots \wedge Sat(\tilde{s}_{i-1}, g) \wedge Sat(\tilde{s}_i, h))$

$\square$

Note that we have the strong version of the until operator: g need never hold, and h must eventually hold.

Using these operators we can define other operators as shorthand.

Definition 4.7 (Other operators) *Some that we shall use are:–*

- *Disjunction:* g `or` h = `not` ((`not` g) `and` (`not` h)).
- *Sometime:* `Exists` g = `True` `Until` g. (Some suffix of the sequence satisfies g.)
- *Always:* `Global` g = `not` (`Exists` `not` g). (No suffix of the sequence does not satisfy g, hence all must satisfy g).
- *Weak until:* g `UntilW` h = (g `Until` h) `or` (`Global` g). (This doesn't demand that h ever be satisfied.)

We also have bounded versions of `Global`, `Exists`, `UntilW` and `Until`, a generalised version of `Next` and a periodic operator `Periodic` which we 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.

$\square$

If $q = Sat(\sigma, g)$ then we say that σ satisfies g with truth value q. If $q \preceq Sat(\sigma, g)$, 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 4.2 The satisfaction relation is monotonic. For all $\tilde{s}, \tilde{t} \in \mathcal{S}^{\omega}$, if $q \;=\; Sat(\tilde{s}, g)$ and $\tilde{s} \sqsubseteq \tilde{t}$, then $q \preceq\; Sat(\tilde{t}, g)$

Proof: If g is simple, this follows since g is monotonic. Otherwise the result follows from the monotonicity of the operators of $\mathcal{Q}$. $\square$

Although the basis of the logic is G, the set of simple predicates, Lemma 4.3 shows that all monotonic predicates can be expressed in TL.

Lemma 4.3 For all monotonic predicates $p : \mathcal{S} \to \mathcal{Q}$, $\exists p' \in$ TL such that $p \equiv p'$.

Proof: See Section A.1. $\square$

4.2 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 since a symbolic formula represents the set of all possible instantiations of that symbolic formula.

TL is extended to symbolic domains by allowing variables to appear in the formulas. Let $\mathcal{V}$ be a set of variable names $\{v_1, \ldots, v_n\}$.

Definition 4.8 (The Extended Logic — $\dot{\mathrm{T}}$L) The syntax of the set of symbolic TL formulas, $\dot{\mathrm{T}}$L, is defined by:–

$$\dot{TL} ::= G \mid \mathcal{V} \mid \dot{TL}\ \texttt{and}\ \dot{TL} \mid \texttt{not}\,\dot{TL} \mid \texttt{Next}\,\dot{TL} \mid \dot{TL}\,\texttt{Until}\,\dot{TL}$$

The derived operators are defined in a similar way to Definition 4.7. For convenience, where there is little chance of confusion, we write TL rather than $\dot{\mathrm{T}}$L.

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 boolean values $\{\mathbf{f}, \mathbf{t}\}$. 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 $\phi \in \Phi$,

$\phi(g) = g$ if $g \in G$
$\phi(\texttt{not}\, g) = \texttt{not}\, \phi(g)$
$\phi(g_1\ \texttt{and}\ g_2) = \phi(g_1)\ \texttt{and}\ \phi(g_2)$
$\phi(\texttt{Next}\, g) = \texttt{Next}\, \phi(g)$
$\phi(g_1\,\texttt{Until}\,g_2) = \phi(g_1)\,\texttt{Until}\,\phi(g_2)$

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

The symbolic satisfaction relations, SAT_q, determine, for different degrees of truth, for which interpretations of variables a sequence satisfies a formula.

Definition 4.9 (Satisfaction relations for $\dot{\mathrm{T}}$L) A number of satisfaction relations are defined.

- For $q = \mathbf{f}, \mathbf{t}, \top$,
  $SAT_q(\tilde{s}, g) \ = \ \{\phi \in \Phi | q = Sat(\tilde{s}, \phi(g))\}$.
- For $q = \mathbf{f}, \mathbf{t}, \top$,
  $SAT_{q\uparrow}(\tilde{s}, g) \ = \ \{\phi \in \Phi | q \preceq Sat(\tilde{s}, \phi(g))\}$.

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

- $SAT_{\top\uparrow}(\tilde{s}, g)$ is the set of interpretations for which g and $\neg g$ hold. Such results are undesirable and verification algorithms should detect and flag them. $SAT_{\top\uparrow}(\tilde{s}, g) = SAT_{\top}(\tilde{s}, g)$.
- $SAT_{\mathbf{t}}(\tilde{s}, g)$ is the set of interpretations for which g is (sensibly) true. $SAT_{\mathbf{t}\uparrow}(\tilde{s}, g) = SAT_{\top}(\tilde{s}, g) \cup SAT_{\mathbf{t}}(\tilde{s}, g)$.
- $SAT_{\mathbf{f}}(\tilde{s}, g)$ is the set of mappings for which g is (sensibly) false. $SAT_{\mathbf{f}\uparrow}(\tilde{s}, g) = SAT_{\top}(\tilde{s}, g) \cup SAT_{\mathbf{f}}(\tilde{s}, g)$.

Thus each satisfaction relation defines a set of interpretations for which a desired relationship holds. Sets of interpretations can be represented efficiently using ordered binary-decision diagrams (BDDs) [6].

4.3 Circuit models as state spaces

In practice, the model-checking algorithms described here 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 (**X**) and inconsistent or over-defined (**Z**) values. The set $\mathcal{C} = \{\mathbf{X}, \mathbf{L}, \mathbf{H}, \mathbf{Z}\}$ forms a lattice, the partial order given in Figure 5.

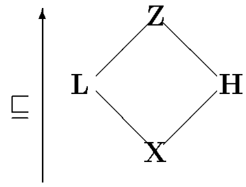

Figure 5: The partial order for $\mathcal{C}$

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

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

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

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,

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

Note that all members of G_n are simple and hence monotonic. The semantics of TL_n is the definition of the semantics of TL (Definition 4.6), replacing G with G_n; this is reproduced below for completeness.

Definition 4.10 *Semantics of* TL_n

1. If $g \in G$ then $Sat(\tilde{s}, g) = g(s_0)$;
2. $Sat(\tilde{s}, g \,\texttt{and}\, g) = Sat(\tilde{s}, g) \wedge Sat(\tilde{s}, h)$;
3. $Sat(\tilde{s}, \texttt{not}\, g) = \neg Sat(\tilde{s}, g)$;
4. $Sat(\tilde{s}, \texttt{Next}\, g) = Sat(\tilde{s_1}, g)$;
5. $Sat(\tilde{s}, g \,\texttt{Until}\, h) = \vee_{i=0}^{\infty}(Sat(\tilde{s}_0, g) \wedge \ldots \wedge Sat(\tilde{s}_{i-1}, g) \wedge Sat(\tilde{s}_i, h))$.

These definitions are useful because in practice properties of interest are built up from the set of predicates which say things about individual state components. Lemma 4.4 shows that restricting the basis of TL_n to G is not a real restriction, since using the operators such as conjunction we can construct any simple predicate.

Lemma 4.4 (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 Lemma 4.3 and Lemma 4.4 is that the logic TL_n is powerful enough to describe all monotonic state predicates over $\mathcal{Q}$, something which is not true for trajectory formulas.

The definition of the symbolic version of TL_n is exactly the same as the general definitions (Definitions 4.8 and 4.9), substituting G_n for G.

Note that if g is a formula of TL_n in which $\top$ does not syntactically appear, $Sat(\sigma, g) = \top$ only if there exists i, j such that $\sigma_i[j] = \mathbf{Z}$. Thus, if g is a formula with this restriction, and $\mathbf{Z}$ does not appear in σ then $SAT_{\mathbf{t}\uparrow}(\sigma, g) = SAT_{\mathbf{t}}(\sigma, g)$. This property is important since we are most interested in the $SAT_{\mathbf{t}}$ relation. As shown in the next section, there is a good decision procedure for the relation $SAT_{\mathbf{t}\uparrow}$; using this property we can extend the result to $SAT_{\mathbf{t}}$.

5 Symbolic Trajectory Evaluation – STE

Symbolic trajectory evaluation (STE) is a model checking algorithm for checking partially-ordered state spaces. It was first presented in [8] and a full description of STE can be found in [32]. In these presentations, the algorithm is applied only to trajectory formulas, a restricted, two-valued temporal logic. In practice it has been applied successfully in hardware verification. This paper generalises STE in two important respects:

1. It presents the theory for the quaternary logic.

2. It presents the theory for the full class of TL. In particular it deals with disjunction and negation.

For readability, proofs are deferred to the appendix.

5.1 Preliminary

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. Informally, the *trajectories* are all the possible runs of the system; formally, a *trajectory*, σ, is a sequence compatible with the next state function: $\forall i \geq 0, \mathbf{Y}(\sigma_i) \sqsubseteq \sigma_{i+1}$. The partial order on $\mathcal{S}$ is extended point-wise to sequences. Let $\mathcal{S}_\mathcal{T}$ be the set of trajectories and, $\mathcal{R}_\mathcal{T} = \{\sigma \in \mathcal{R}^\omega \cap \mathcal{S}_\mathcal{T}\}$ is the set of realisable trajectories. $\mathcal{R}_\mathcal{T}$ represents those trajectories corresponding to real behaviours of a system.

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

Definition 5.1 (Notation) If A and B are subsets of a lattice $\mathcal{L}$ on which a partial order $\sqsubseteq$ is defined, then $A \sqcup B = \{a \sqcup b : a \in A, b \in B\}$. If $g: \mathcal{L} \to \mathcal{L}$, then $g(A) = \{g(a) : a \in A\}$, and similarly, $g(\langle A, B\rangle) = \langle g(A), g(B)\rangle$.

Definition 5.2 If $\mathcal{S}$ is a lattice with partial order $\sqsubseteq$ and $A, B \subseteq \mathcal{S}^\omega$, then $A \sqsubseteq_\mathcal{P} B$ if $\forall b \in B, \exists a \in A$ such that $a \sqsubseteq b$. Where the types can be inferred readily, we write $A \sqsubseteq B$ rather than $A \sqsubseteq_\mathcal{P} B$.

Informally, the motivation for this definition is that formulas of the logic can be represented by the set of sequences which satisfy them. $\sqsubseteq_\mathcal{P}$ can be used to represent a type of logical implication. Suppose that A is a set of sequences such that any sequence larger than any sequence in A has property g, and that B is a set of sequences such that any sequence larger than any sequence in B has property h. If $A \sqsubseteq_\mathcal{P} B$ then every sequence which has property h also has property g since if σ is larger than some sequence in B it must also be larger than some sequence in A. How this is used is a major result of this section.

Lemma 5.1 If $\mathcal{S}$ is a lattice with partial order $\sqsubseteq$, then $\sqsubseteq_\mathcal{P}$ is a preorder (i.e., it is reflexive and transitive).

5.2 Scalar Trajectory Evaluation

Recall the definition of defining pair and defining set from Section 4.1. The defining set of a simple predicate characterises that predicate. We can use this as a building block to find a characterisation of all temporal predicates.

Some formulations of temporal logic give the semantics of a formula by the giving the set of sequences which satisfy it. In practice it may not be possible to use this definition directly since such sets are likely to be extremely large. The advantage of our approach is that since sequences are partially ordered, the minimal sequences which satisfy a formula can be used to represent the entire set, which in turn means that the set of minimal sequences can be used to characterise a formula. These sets are called defining sequence sets. In our experience with verification using STE, there are many formulas which 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 $\mathbf{t} \preceq \mathit{Sat}(\sigma, g)$, and the second set contains those sequences for which $\mathbf{f} \preceq \mathit{Sat}(\sigma, g)$. 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.

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 which satisfy g_1 $\texttt{and}$ 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, $q \preceq \mathit{Sat}(\sigma^J, g_1 \texttt{ and } g_2)$. Moreover, suppose $q \preceq \mathit{Sat}(\sigma', g_1 \texttt{ and } g_2)$, 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 $\texttt{and}$ g_2.

What about characterising sequences which satisfy g_1 $\texttt{or}$ 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 which satisfy it, taking the meets will lose information. While it will be the case that if $q \preceq \mathit{Sat}(\sigma', g_1 \texttt{ or } g_2)$ then $\sigma_1 \sqcap \sigma_2 \preceq \sigma'$, the converse does not hold in general. This means that to characterise g_1 $\texttt{or}$ 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 which satisfy a predicate with value at least $\mathbf{t}$ and those that satisfy a predicate with value at least $\mathbf{f}$.

Definition 5.3 (Defining sequence set) Let $g \in \mathrm{TL}$. 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, $\Delta^q(g) = \{s\mathbf{XX}\ldots : (s, q) \in D_g, \text{ or } (s, \top) \in D_g\}$. 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. $\Delta(g_1 \texttt{ or } g_2) = \langle \Delta^{\mathbf{t}}(g_1) \cup \Delta^{\mathbf{t}}(g_2), \Delta^{\mathbf{f}}(g_1) \sqcup \Delta^{\mathbf{f}}(g_2) \rangle$

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

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

   This case is symmetric to the preceding one.

4. $\Delta(\texttt{not}\, g) = \langle \Delta^{\mathbf{f}}(g), \Delta^{\mathbf{t}}(g) \rangle$

   This is motivated by the fact that for $q = \mathbf{f}, \mathbf{t}$, σ satisfies g with truth value at least q if and only if it satisfies $\texttt{not}\, g$ with truth value at least $\neg q$.

5. $\Delta(\texttt{Next}\, g) = \langle \{\mathbf{X}\sigma : \sigma \in \Delta^{\mathbf{t}}(g)\}, \{\mathbf{X}\sigma : \sigma \in \Delta^{\mathbf{f}}(g)\} \rangle$

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

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

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

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 effect what is presented below.

An important consequence of this definition is that for each formula g of TL, $\Delta(g)$ characterises g: all sequences which 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 5.2 Let $g \in$ TL, and let $\sigma = \sigma_0\tilde{\sigma}$. For $q = \mathbf{t}, \mathbf{f}$, $q \preceq \mathit{Sat}(\sigma, g)$ iff $\exists \delta_g \in \Delta^q(g)$ with $\delta_g \sqsubseteq \sigma$.

The defining sequence sets contains the set of the minimal sequences which satisfy the formula. These sequences are not necessarily trajectories. Given an arbitrary sequence it is possible to find the weakest trajectory larger than it.

Definition 5.4
Let $\sigma = s_0s_1s_2\ldots$. Let $\tau(\sigma) = t_0t_1t_2\ldots$ where

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

$t_0t_1t_2\ldots$ is the smallest sequence larger than σ. s_0 is a possible starting point of a trajectory, so $t_0 = s_0$. Any run of the machine which 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 which satisfy a formula, there is a set of minimal trajectories which satisfy a formula. A set which 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 5.5 (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 which satisfy g. This is formalised in the following lemma which is proved in Section A.2.

Lemma 5.3 Let $g \in$ TL, and let $\sigma = \sigma_0\tilde{\sigma}$ be a trajectory. For $q = \mathbf{t}, \mathbf{f}$, $q \preceq \mathit{Sat}(\sigma, g)$ 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 properties which can be phrased as:

> Do all trajectories that satisfy formula g also satisfy formula 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. The verification approach is formalised in Theorem 5.4 (which is proved in Section A.2). Section 5.3 shows how this result is used in practice. These antecedent, consequent pairs are called assertions.

Theorem 5.4 Let $g, h \in \mathrm{TL}, q \in \mathcal{B}$.
$\Delta^q(h) \sqsubseteq T^q(g)$ if and only if for every trajectory σ with $q \preceq \mathit{Sat}(\sigma, g)$ it is the case that $q \preceq \mathit{Sat}(\sigma, h)$

The proof is given in the appendix.

Definition 5.6 If $g \in \mathrm{TL}$, and $\exists \delta^g \in \Delta^{\mathbf{t}}(g)$ such that $\forall \delta \in \Delta^{\mathbf{t}}(g), \delta^g \sqsubseteq \delta$, 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.

Finite formulas with defining sequences are known as trajectory formulas. Seger and Bryant characterise these syntactically.

There are two useful special cases of Theorem 5.4. First, if A is a formula of TL with a well-defined defining sequence δ^A, and $h \in \mathrm{TL}$, then $\forall \delta \in \Delta^{\mathbf{t}}(h), \delta \sqsubseteq \tau^A$ if and only if, for every trajectory σ for which $\mathbf{t} \preceq \mathit{Sat}(\sigma, A)$ it is the case that $\mathbf{t} \preceq \mathit{Sat}(\sigma, h)$

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 which $q \preceq \mathit{Sat}(\sigma, A)$ it is the case that $q \preceq \mathit{Sat}(\sigma, C)$. This is essentially the result of Seger and Bryant generalised to the four valued logic.

5.3 Verification with STE

We first have to decide how to deal with inconsistent information, i.e. whether to use the $SAT_{\mathbf{t}}$ or $SAT_{\mathbf{t}\uparrow}$ relations for verification. Ideally, given an antecedent g and consequent h we want to know whether every realisable trajectory that 'properly' satisfies g also 'properly' satisfies h: $\forall \sigma \in \mathcal{R}_{\mathcal{T}}, \mathbf{t} = \mathit{Sat}(\sigma, g)$ implies that $\mathbf{t} = \mathit{Sat}(\sigma, h)$. In general g and h are symbolic and so we want to know for which interpretations of variables the result holds. Formally this is put as:

Definition 5.7 $[g \Longrightarrow\!\!\!\!\triangleright\, h] = \{\phi \in \Phi : \forall \sigma \in \mathcal{R}_{\mathcal{T}}, \mathbf{t} = \mathit{Sat}(\sigma, \phi(g))$ implies that $\mathbf{t} = \mathit{Sat}(\sigma, \phi(h))\}$.

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

Definition 5.8 $\models_{\mathcal{M}} [g \Longrightarrow\!\!\!\!\triangleright\, h] = ([g \Longrightarrow\!\!\!\!\triangleright\, h] = \Phi)$

Note that $\models_{\mathcal{M}} [g \Longrightarrow\!\!\!\!\triangleright\, h]$ if and only if $\forall \sigma \in \mathcal{R}_{\mathcal{T}}, SAT_{\mathbf{t}}(\sigma, g)$ implies that $SAT_{\mathbf{t}}(\sigma, h)$. 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 5.9 $[g \Longrightarrow\!\!\!\!> h] = \{\phi \in \Phi : \forall \sigma \in \mathcal{S}_{\mathcal{T}}, \mathbf{t} \preceq \mathit{Sat}(\sigma, \phi(g))$ implies that $\mathbf{t} \preceq \mathit{Sat}(\sigma, \phi(h))\}$

and

Definition 5.10 $\models_{\mathcal{M}} [g \Longrightarrow\!\!\!\!> h] = ([g \Longrightarrow\!\!\!\!> h] = \Phi)$

Note that $\models_{\mathcal{M}} [g \Longrightarrow\!\!\!> h]$ if and only if $\forall \sigma \in \mathcal{S}_{\mathcal{T}}, SAT_{\mathbf{t}\uparrow}(\sigma, g) \subseteq SAT_{\mathbf{t}\uparrow}(\sigma, h))$.

While the $\Longrightarrow\!\!\!>$ relation does not capture exactly the notion of correctness that we want, Theorem 5.4 provides a decision procedure for this: $\models_{\mathcal{M}} [g \Longrightarrow\!\!\!> h]$ exactly when $\Delta^{\mathbf{t}}(h) \sqsubseteq_{\mathcal{P}} T^{\mathbf{t}}(g)$. And, for circuit models we can use this as a basis for proving the more refined notion of satisfaction.

When $\mathcal{S} = \mathcal{C}^n$ and $\mathcal{R} = \{\mathbf{X}, \mathbf{L}, \mathbf{H}\}^n$, and only formulas of TL_n not syntactically containing $\top$ are considered, computing these verification results is simplified. In the rest of this section we only consider this class of TL_n. We use the following two facts:

1. $\sigma \in \mathcal{R}_{\mathcal{T}}$ if and only if, $\mathbf{Z}$ does not appear in σ (for all i, j, $\sigma_i[j] \neq \mathbf{Z}$).
2. If $\sigma \in \mathcal{R}_{\mathcal{T}}$, $SAT_{\top}(\sigma, g) = \emptyset$ and $SAT_{\mathbf{t}}(\sigma, g) = SAT_{\mathbf{t}\uparrow}(\sigma, g)$

We compute $\models_{\mathcal{M}} [g \Longrightarrow\!\!\triangleright\, h]$ as follows. First, compute $T^{\mathbf{t}}(g)$. It is easy to determine whether $T^{\mathbf{t}}(g) \subseteq \mathcal{R}_{\mathcal{T}}$ using Fact 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 $T^{\mathbf{t}}(g) \subseteq \mathcal{R}_{\mathcal{T}}$.

$$
\begin{array}{lll}
\models_{\mathcal{M}} [g \Longrightarrow\!\!\!> h] & & \\
\quad = & \forall \sigma \in \mathcal{S}_{\mathcal{T}}, (SAT_{\mathbf{t}\uparrow}(\sigma, g) \subseteq SAT_{\mathbf{t}\uparrow}(\sigma, h)) & \text{By definition} \\
\quad \Longrightarrow & \forall \sigma \in \mathcal{R}_{\mathcal{T}}, (SAT_{\mathbf{t}\uparrow}(\sigma, g) \subseteq SAT_{\mathbf{t}\uparrow}(\sigma, h)) & \mathcal{R}_{\mathcal{T}} \subseteq \mathcal{S}_{\mathcal{T}} \\
\quad \Longrightarrow & \forall \sigma \in \mathcal{R}_{\mathcal{T}}, (SAT_{\mathbf{t}}(\sigma, g) \subseteq SAT_{\mathbf{t}}(\sigma, h)). & \text{By Fact 2.} \\
\quad = & \models_{\mathcal{M}} [g \Longrightarrow\!\!\triangleright\, h] &
\end{array}
$$

This result is useful because in this important special case, efficient STE-based algorithms can be used. The rest of this paper uses this result implicitly. The main computational task is to determine $\models_{\mathcal{M}} [g \Longrightarrow\!\!\!> h]$. By placing sensible restrictions on the logic used and checking for inconsistency in the defining trajectory set of the antecedent, we can then deduce $\models_{\mathcal{M}} [g \Longrightarrow\!\!\triangleright\, h]$ from $\models_{\mathcal{M}} [g \Longrightarrow\!\!\!> h]$.

6 Model Checking Algorithms

Section 5 presents a theoretical decision procedure for the logic. This section outlines practical model checking algorithms based on existing symbolic trajectory evaluation algorithms [32]. The algorithms presented below model check results of the form $\models_{\mathcal{M}} [A \Longrightarrow\!\!\!> h]$ where A is a trajectory formula and $h \in \mathrm{TL}$.

Scope of verification algorithms

Theorem 5.4 presents a general verification methodology based on STE. However, there is a trade-off between the power of the logic and the computational cost of decision procedures. For the purpose of this paper, we restrict antecedents to be trajectory formulas, which has very important practical implications for efficiency. As our experience with STE shows that the main need is for enriched consequents, this is a justifiable trade-off; enriching the antecedents is left for future research.

This section sketches three decision procedures for assertions of the form $\models_{\mathcal{M}} [A \Longrightarrow\!\!\triangleright\, g]$. In each case, A must be a trajectory formula. The first procedure is the most general — g can be any formula of TL. The two other procedures deal with finite subsets of TL (not allowing the operators `Global` and `Exists`). In the examples shown in the Section 7, all three approaches are shown to be feasible. More experimental evidence is needed to determine their advantages and disadvantages.

6.1 Direct Modification of STE

For trajectory formulas, using the method of Bryant and Seger verifying $\models_{\mathcal{M}} [A \Longrightarrow C]$ entails computing τ^A and δ^C and comparing the two. In principle, exactly the same approach could be taken to verify $\models_{\mathcal{M}} [g \Longrightarrow h]$ for arbitrary g and h. Note, that in principle the fact that both g and h could contain 'infinite' temporal operators is not a problem. Since the state space is finite, all sequences must contain repeated elements. Hence two infinite sequences can be compared by comparing suitably long finite prefixes. In practice, this would be too expensive and probably no better than other model checking approaches.

Verifying $\models_{\mathcal{M}} [A \Longrightarrow h]$ where A is a trajectory formula is much easier since $T^{\mathbf{t}}(A)$ has exactly one element, and because A is finite. Checking whether $\Delta^{\mathbf{t}}(h) \sqsubseteq_{\mathcal{P}} T^{\mathbf{t}}(A)$ means checking whether $\exists \delta \in \Delta^{\mathbf{t}}(h)$ such that $\delta \sqsubseteq \tau^A$; this can often be checked efficiently, particularly when h is finite.

6.2 Use of testing machines

The second approach transforms the problem of determining whether $\models_{\mathcal{M}} [A \Longrightarrow h]$ is true into the problem of determining whether $\models_{\mathcal{M}'} [A' \Longrightarrow C]$ holds of $\mathcal{M}'$ where A' and C are trajectory formulas. A', C and $\mathcal{M}'$ are all computed automatically from A, h and $\mathcal{M}$. Intuitively, extra testing circuitry is added to the circuit to be tested to observe the values in the circuit being verified and compute whether h holds of its observation. Once M' and C have been found, this can be done by using standard STE algorithms. This idea is very similar to the idea of using satellites or observers [4].

This method is applicable to cases where the consequent is finite. Its great advantage is that the simplicity of the approach allows a uniform way for constructing the testing machines. The cost of the method is the construction of the testing circuitry, composing it with the model, and then the extra cost in performing trajectory evaluation on $\mathcal{M}'$ (which is bigger than $\mathcal{M}$).

6.3 Use of mapping information

The third approach determines whether $\models_{\mathcal{M}} [A \Longrightarrow h]$ holds by computing $\varphi = [A \Longrightarrow C]$ where C is a trajectory formula. This works by constructing C in such a way as to extract enough information from τ^A by using extra boolean variables in the consequent. Performing trajectory evaluation determines for which interpretations of these extra variables the verification holds. From the set of interpretations, we can compute whether h holds. If for each interpretation of variables in A there is an interpretation of variables in C such that the composed interpretation is in φ and if this interpretation satisfies h, then the verification condition holds.

This method may entail the use of many extra boolean variables, creating larger BDDs and thereby slowing down trajectory evaluation considerably. This needs experimental testing.

7 Examples

The focus of this paper is theoretical – the introduction of the new logic in Sections 3 and 4 and a decision procedure for the logic in Section 5. To illustrate the use of the logic, two small examples are given. The three different approaches discussed in the previous section were used. Extensive experimental work needs to be done to assess the advantages, disadvantages and limitations of all three approaches. Work is currently under way on this and optimising the implementations of the

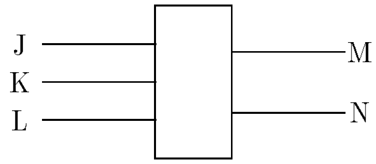

Figure 6: A CSA Adder

algorithms. The results are presented only to show that all three approaches are feasible, and the times quoted should be viewed in that light.

The Voss tool

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 efficient manipulation of boolean expressions. Using BDDs, boolean expressions have canonical forms making comparison of expressions very efficient. Though BDDs have practical limitations, the use of BDD-based methods has extended by orders of magnitude the size of systems that can be tackled by model-checkers.

The Voss system [31], a formal hardware verification system developed at the University of British Columbia, consists of three major components: an efficient implementation of BDDs; an event driven symbolic simulator with comprehensive delay and race analysis capabilities; and a general purpose, functional language. The language, called FL, is strongly typed, polymorphic, and fully lazy. Every object of type boolean in the system is internally represented as a BDD. Consequently, FL is a very convenient language for developing prototype verification methodologies that require BDD manipulations. Voss has been used to perform STE efficiently on large, sophisticated circuits [1, 3, 17].

7.1 Example 1

The first example shows the verification of the carry-save adder (CSA) [22] shown in Figure 6.

Define the formulas A and h by:

$$A = (J = j) \wedge (K = k) \wedge (L = l)$$

$$h = \texttt{Next}\,(M + N = j + k + l)$$

We wish to verify that $\models_{\mathcal{M}} [A \Longrightarrow\!\!\triangleright\, h]$. This was verified using all three algorithms presented in the previous section. Table 1 summarises the performance of the algorithms in verifying a 64-bit CSA (approximately 360 gates). The time given was obtained by running the example on a DEC Alpha 3000. In all cases, the time spent actually performing trajectory evaluation is approximately 10–15% of the overall time — the rest of the time was spent in reading in the model to be verified and other computation involved in the algorithm.

The current implementations of the algorithms are fairly crude. We need to gather more experience on larger and more varied examples before being able to determine the advantages and disadvantages of the different approaches. The purpose of these experiments was to perform some

	Algorithm	Time (s)
1	Direct	3.8
2	Testing Machine	3.6
3	Mapping information	2.6

Table 1: Experimental results

preliminary evaluation of these approaches, and we believe the experimental results shows that all of them are promising.

7.2 Example 2

The second example shows the verification of a B8ZS encoder. This is a very simple circuit but this is an example which it would be very difficult to do in traditional STE and illustrates some points about the style of verification.

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 '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 [2]. 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 gives an external view of the encoder. 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. All input and output lines are serial.

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.

2. Does the implementation have the properties that we expect? (Specification validation) In particular is it the case that:

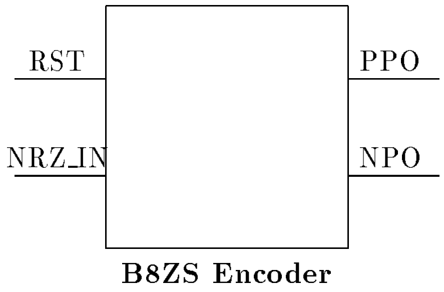

Figure 7: B8ZS Encoder

- 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 k-th output bit pair (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 $(k-6)$-th bit 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, we define a set of reachable states and show that the encoder enters one of the reachable states after it is reset. We show that if the encoder is in a reachable state and runs one time unit it enters another reachable state. Then we show that if the encoder starts in a reachable state then the output of the ninth input bit is correctly interpreted. 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 specification. 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 a trajectory formula. 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). The direct method could verify the general case.

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.

8 Related Work

Temporal logics are now well-established formalisms for specifying correctness properties of systems. Since automatic model checking algorithms were first proposed in the early 1980s [12], much progress has been made and there are now many successful techniques based on BDDs (e.g. [9]) and tableaux (e.g. [5]).

The key distinguishing feature of our temporal logic is that it is a four-valued logic. Although four-valued logics have been used in other areas of computer science, we believe having a four-valued temporal logic is a novel contribution. Its utility and justification is that it is the appropriate technical setting for model checking partially-ordered state spaces.

This use of a multi-valued logic is very different to the research in the use of multi-valued logics for circuits where the focus is on circuit components which can take on many values (unlike current technology where components typically take on one of two values). Our use of a lattice to represent the state space is a representational convenience for modelling purposes. Furthermore, we carefully distinguish between the lattice used to represent the state space, and the lattice used to represent the logic in which we reason about the state space.

Syntactically and semantically, TL looks much like logics such as CTL [13]. However it does not have the $\forall$ and $\exists$ operators. Since the next state function is deterministic, there is no need to reason about different paths. In a deterministic system, there is no distinction between linear-time and branching-time. This doesn't quite close the issue, since as alluded to earlier, we can use a lattice structure to represent non-determinism (and with circuit models we successfully represent input non-determinism). This entails converting a non-deterministic model with a flat state space into a deterministic lattice model. What then are the semantics in the original model? This is a subtle point and all the examples that STE has been used in, this issue has not arisen. However, it raises an important question for future research.

The new techniques of model checking have greatly increased the size of the models that can be checked. However, the state explosion problem cannot be avoided and other techniques are needed to deal with many real problems. Abstraction is one popular idea — 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.

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 [15, 19, 23, 24, 28].

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 (which is difficult to do automatically, but see [14, 33]) and prove that the abstraction is conservative (in which progress has been made in automatically checking). For an example of this type of proof see [10].

A great advantage of STE-based approaches is that the lattice structure of the state space allows simple and implicit use of abstractions of the model. Essentially the choice of antecedent determines the amount of information about the behaviour of the model which will be used. This type of abstraction is different to the abstractions described above — the methods are orthogonal. These abstractions are potentially very powerful, but require a higher degree of sophistication of use. STE also has the advantage that the abstractions are tailored to the particular verification assertions being checked: for each assertion a different abstraction is used. Furthermore, the use of abstraction in STE does not affect accuracy of timing. For low level verification, this may be very important.

The use of a four-valued logic complements this: in traditional conservative approximation approaches proving that a property does not hold of an abstraction says nothing about the underlying model. In our approach we can use **f** and $\perp$ to distinguish between a property not holding and the abstraction being too weak to prove the property we are interested in.

9 Conclusion

Representing large state spaces with a lattice is an effective way of ameliorating the state explosion problem. The four-valued temporal logic TL is the appropriate technical framework for expressing properties of models which have a partially-ordered state space. Symbolic trajectory evaluation can be used to model check these models. This paper is primarily theoretical and has presented the underlying theory, and only sketched the practical algorithms which can be used for verification.

This work is part of a larger project to use symbolic trajectory evaluation for verification. Important issues are combining the use of theorem-proving and model checking, and the use of a compositional theory for generalised STE [1, 26].

Other areas for research include:

- Practical experience with the logic and the development of efficient tools for verification.
- The development of a compositional theory for the logic.
- Non-determinism. First, we need a deeper understanding of the process of converting a non-deterministic model with a flat state space into a deterministic model with a lattice state space. Second, it would be interesting to investigate the theoretical and practical effects of allowing the lattice-structured model to deal with non-determinism explicitly through the use of a next state relation rather than a next state function.
- Determining how applicable other model-checking approaches would be for this framework of lattice-structured models and truth domains.

Acknowledgement: We thank Mark Aagaard and Mark Greenstreet for their helpful comments and suggestions.

References

[1] M. Aagaard and C.-J.H. Seger. The Formal Verification of a Piplelined Double-Precision IEEE Floating-Point Multiplier. ACM/IEEE International Conference on Computer-Aided Design (*to appear*), 1995.

[2] Advanced Micro Devices. *PAL Device Handbook.* Advanced Micro Devices, Inc., 1988.

[3] D.L. Beatty. *A Methodology for Formal Hardware Verification with Application to Microprocessors*. PhD thesis, Carnegie-Mellon University, School of Computer Science, 1993.

[4] I. Beer, S. Ben-David, D. Geist, R. Gewirtzman, and M. Yoeli. Methodology and system for practical formal verification of reactive hardware. In Dill [18], pages 182–193.

[5] J.C. Bradfield. *Verifying Temporal Properties of Systems*. Birkhäuser, Boston, 1992.

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

[7] R.E. Bryant, D.L. Beatty, and C.-J. H. Seger. Formal Hardware Verfication by Symbolic Ternary Trajectory Evaluation. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pages 397–407. 1991.

[8] R.E. Bryant and C.-J. H. Seger. Formal Verfication 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.

[9] 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.

[10] J.R. Burch and D.L. Dill. Automatic Verification of Piplelined Microprocessor Control. In Dill [18], pages 68–80.

[11] G. Cabodi and P. Camurati. Advancements in symbolic traversal techniques. In Milne and Pierre [29], pages 155–166.

[12] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach. In *Proceedings of the 10th ACM Symposium on the Principles of Programming Languages*, New York, 1983. ACM.

[13] 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.

[14] E.M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In Courcoubetis [16], pages 450–462.

[15] E.M. Clarke, O. Grumberg, and D.E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5), September 1994.

[16] C. Courcoubetis, editor. *Proceedings of the 5th International Conference on Computer-Aided Verification*, number 697 in Lecture Notes in Computer Science, Berlin, July 1993. Springer-Verlag.

[17] M. Darwish. Formal Verification of a 32-Bit Pipelined RISC Processor. MASc Thesis, University of British Columbia, Department of Electrical Engineering, 1994.

[18] 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.

[19] M. Donat. Verification Using Abstract Domains. Unpublished paper, Department of Computer Science, University of British Columbia, April 1993.

[20] M. Fitting. Bilattices and the Theory of Truth. *Journal of Philosophical Logic*, 18(3):225–256, August 1989.

[21] M. Fitting. Bilattices and the Semantics of Logic Programming. *The Journal of Logic Programming*, 11(2):91–116, August 1991.

[22] 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.

[23] S. Graf and C. Loiseaux. A tool for symbolic program verification and abstraction. In Courcoubetis [16], pages 71–84.

[24] N.A. Harman and J.V. Tucker. Algebraic models and the correctness of microprocessors. In Milne and Pierre [29], pages 92–108.

[25] S. Hazelhurst and C.-J. H. Seger. Composing symbolic trajectory evaluation results. In Dill [18], pages 273–285.

[26] S. Hazelhurst and C.-J.H. Seger. A Simple Theorem Prover Based on Symbolic Trajectory Evaluation ad BDD's. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(4):413–422, April 1995.

[27] A.J. Hu and D.L. Dill. Efficient Verification with BDDs using Implicitly Conjoined Invariants. In Courcoubetis [16], pages 3–14.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] C.-J.H. Seger and R.E. Bryant. Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories. *Journal of Formal Methods in Systems Design*, 6:147–189, March 1995.

[33] P.A. Subrahmanyam. Towards Verifying Large(r) Systems: A strategy and an experiment. In Milne and Pierre [29], pages 135–154.

[34] A. Visser. Four Valued Semantics and the Liar. *Journal of Philosophical Logic*, 13(2):181–212, May 1984.

A Technical results

A.1 Auxiliary results

Proof of Lemma 4.1

Lemma A.1 If $g, h: \mathcal{S} \to \mathcal{Q}$ are simple, then $D(g) = D(h)$ implies that $g \equiv h$.

Proof: To emphasise that $D(g) = D(h)$, we set $D = D(g)$. Let $s \in \mathcal{S}$.

Let $E = \{q \in \mathcal{Q} : (s_q, q) \in D \wedge s_q \sqsubseteq s\}$
$e = \sqcup E$

1. $g(s) \preceq e$
   Since:
   g is simple, $\exists s_p \in \mathcal{S}$ such that $(s_p, g(s)) \in D$.
   $s_p \sqsubseteq s$ by definition of defining pair.
   $\Longrightarrow g(s) \in E$ by definition of E
   $\Longrightarrow g(s) \preceq \sqcup E$ by definition of join.
2. $e \preceq g(s)$
   Since:
   By monotonicity of g, $\forall (s_q, q) \in E, s_q \sqsubseteq s \Longrightarrow q = g(s_q) \preceq g(s)$
   $\Longrightarrow \sqcup E \preceq g(s)$

Thus $g(s) = e$. Similarly, $h(s) = e$.
$\Longrightarrow g(s) = h(s)$

As s was arbitrary $g \equiv h$. □

Note that the proof does not rely on the particular structure of $\mathcal{Q}$; it only relies on $\mathcal{Q}$ being a complete lattice.

Proof of Lemma 4.3

First, equivalence must be formally defined. If p is a monotonic predicate, then $p: \mathcal{S} \to \mathcal{Q}$. If $p' \in$ TL, then its meaning is given by the satisfaction relation. However, where p' contains no temporal operator it can also be considered as a function from $\mathcal{S}$ to $\mathcal{Q}$ since its meaning is given by the degree to which the first elements of sequences satisfy it. Formally $p \equiv p'$ if $\forall s \in \mathcal{S}, p(s) = \mathit{Sat}(s\mathbf{XX}\ldots, p')$.

Let $p: \mathcal{S} \to \mathcal{Q}$ be an arbitrary monotonic predicate. Partition $\mathcal{S}$ according to the value of p:
$S_\perp = \{s \in \mathcal{S} : p(s) = \perp\}$.
$S_\mathbf{f} = \{s \in \mathcal{S} : p(s) = \mathbf{f}\}$.
$S_\mathbf{t} = \{s \in \mathcal{S} : p(s) = \mathbf{t}\}$.
$S_\top = \{s \in \mathcal{S} : p(s) = \top\}$.

Some of these sets may be empty. Now, for each $s \in \mathcal{S}$, we define $\chi_s: \mathcal{S} \to \mathcal{Q}$ as follows:

$$\chi_s(t) = \begin{cases} \mathbf{t}, & s \sqsubseteq t \\ \perp, & s \not\sqsubseteq t \end{cases} \qquad A.1$$

Note that each χ_s is simple. We define as auxiliary operators, $\rhd x = x \wedge \top$ and $\lhd x = \neg(\rhd x)$. Recall that $\top \wedge \perp = \mathbf{f}$ so these definitions yield:

$$\rhd x = \begin{cases} \mathbf{f}, & \text{when } x = \mathbf{f}, \perp \\ \top, & \text{when } x = \mathbf{t}, \top \end{cases}$$
$$\lhd x = \begin{cases} \mathbf{t}, & \text{when } x = \mathbf{f}, \perp \\ \top, & \text{when } x = \mathbf{t}, \top \end{cases}$$

Further define $\chi_{\mathbf{f}}(t) = \texttt{ or }_{(s \in S_{\mathbf{f}})} \chi_s(t)$, $\chi_{\mathbf{t}}(t) = \texttt{ or }_{(s \in S_{\mathbf{t}})} \chi_s(t)$ and $\chi_{\top}(t) = \texttt{ or }_{(s \in S_{\top})} \chi_s(t)$. For the purpose of this lemma, define $\texttt{ or } \emptyset = \perp$. Thus, $\lhd(\texttt{ or } \emptyset) = \mathbf{t}$.

Define $p'(t)$ by $p'(t) = \chi_{\mathbf{t}}(t) \texttt{ and } \lhd \chi_{\mathbf{f}}(t) \texttt{ and } \lhd \chi_{\top}(t) \texttt{ or } \rhd \chi_{\top}(t)$. Intuitively, the first terms checks whether $p(s)$ is a least $\mathbf{t}$ and not at least $\mathbf{f}$ and not at least $\top$. The second term asks checks whether $p(s)$ is $\top$.

Finally, the lemma we want to prove.

Lemma A.2 Let $p : \mathcal{S} \to \mathcal{Q}$ be a monotonic predicate. Then $\exists p' \in \mathrm{TL}$ such that $p \equiv p'$.

Proof: Define $p'(t)$ by

$$p'(t) = \chi_{\mathbf{t}}(t) \texttt{ and } \lhd \chi_{\mathbf{f}}(t) \texttt{ and } \lhd \chi_{\top}(t) \texttt{ or } \rhd \chi_{\top}(t)$$

- Suppose $t \in S_{\perp}$. Thus $p(t) = \perp$. By definition, $\chi_{\mathbf{f}}(t) = \chi_{\mathbf{t}}(t) = \chi_{\top}(t) = \perp$. Therefore $p'(t) = \perp \wedge \mathbf{t} \wedge \mathbf{t} \vee \mathbf{f} = \perp = p(t)$,
- Suppose $t \in S_{\mathbf{f}}$. Thus $p(t) = \mathbf{f}$. By definition, $\chi_{\mathbf{f}}(t) = \mathbf{t}, \chi_{\mathbf{t}}(t) = \chi_{\top}(t) = \perp$. Therefore $p'(t) = \perp \wedge \top \wedge \mathbf{t} \vee \mathbf{f} = \mathbf{f} = p(t)$.
- Suppose $t \in S_{\mathbf{t}}$. Thus $p(t) = \mathbf{t}$. By definition, $\chi_{\mathbf{t}}(t) = \mathbf{t}, \chi_{\mathbf{f}}(t) = \chi_{\top}(t) = \perp$. Therefore $p'(t) = \mathbf{t} \wedge \mathbf{t} \wedge \mathbf{t} \vee \mathbf{f} = \mathbf{t} = p(t)$.
- Suppose $t \in S_{\top}$. Thus $p(\top) = \top$. By definition, $\chi_{\top}(t) = \mathbf{t}, \lhd \chi_{\top}(t) = \rhd \chi_{\top}(t) = \top$.

$$\begin{array}{rcl} \Longrightarrow p'(t) & = & \chi_{\mathbf{t}}(t) \wedge (\lhd \chi_{\mathbf{f}}(t)) \wedge \top \vee \top \\ & \succeq & \perp \wedge \perp \wedge \top \vee \top \\ & = & \mathbf{f} \vee \top = \top \\ & = & p(t) \\ \Longrightarrow p'(t) & = & p(t). \end{array}$$

All the χ_s are simple as are the constant predicates $\perp, \mathbf{f}$ and $\mathbf{t}$, so 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. $\Box$

Proof of Lemma 4.4

Lemma A.3 (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: For each $(s_q, q) \in D(p)$, let $s_q = \langle p_{q,1}, \ldots, p_{q,n} \rangle$.
For each $x \in \mathcal{C}, i = 1, \ldots, n$, define $\chi(x, [i]) : \mathcal{C}^n \to \mathcal{Q}$ by

$$\chi(x, [i]) = \begin{cases} \mathbf{t}, & \text{when } x = \mathbf{X} \\ \perp \texttt{ or not } [i], & \text{when } x = \mathbf{L} \\ \perp \texttt{ or } [i], & \text{when } x = \mathbf{H} \\ \perp \texttt{ or } (\texttt{not } [i] \texttt{ and } [i]), & \text{when } x = \mathbf{Z} \end{cases}$$

Note that $\chi(\mathbf{L}, [i]) \texttt{ and } \chi(\mathbf{H}, [i]) = \chi(\mathbf{Z}, [i])$. Given a state s, $\chi(p_{x,i}, [i])(s)$ returns $\mathbf{t}$ if s's i-th component is at least as large as the i-th component of the p's defining value for x, and $\perp$ otherwise.

Now define $\chi_x(s) = \texttt{and}_{i=1}^{n}(\chi(x,[i])(s))$.

$$\Longrightarrow \chi_x(s) = \begin{cases} \mathbf{t}, & \text{if } p_x \sqsubseteq s \\ \perp, & \text{otherwise} \end{cases}$$

This is exactly the definition of Equation A.1. Thus, adopting the definitions of the previous section, if we define $p'(t) = \chi_{\mathbf{t}}(t)$ and $(\lhd \chi_{\mathbf{f}}(t))$ and $(\lhd \chi_{\top}(t))$ or $\rhd \chi_{\top}(t)$, we have by Lemma 4.3 that $p' \equiv p$. □

A.2 Proofs from Section 5

Proof of Lemma 5.2

First, an auxiliary result.

Lemma A.4 If $g \in \text{TL}, \delta \in \Delta^q(g)$, then $q \preceq \mathit{Sat}(\delta, g)$.

Proof: Let $g \in \text{TL}, (\delta = s_0 s_1 s_2 \ldots) \in \Delta^q(g)$. Proof by structural induction.

1. If g is simple, $\mathit{Sat}(\delta, g) = q$ or $\mathit{Sat}(\delta, g) = \top$ by definition of *Sat*.

2. Let $g = g_1 \,\texttt{and}\, g_2$. By definition $\mathit{Sat}(\delta, g) = \mathit{Sat}(\delta, g_1) \wedge \mathit{Sat}(\delta, g_2)$.

   Suppose $q = \mathbf{t}$, i.e. $\delta \in \Delta^{\mathbf{t}}(g)$. Then, $\delta = \delta_1 \sqcup \delta_2$ for some $\delta_1 \in \Delta^q(g_1)$, $\delta_2 \in \Delta^q(g_2)$. By induction, $q \preceq \mathit{Sat}(\delta_1, g_1), q \preceq \mathit{Sat}(\delta_2, g_2)$. By monotonicity $q \preceq \mathit{Sat}(\delta, g_1), q \preceq \mathit{Sat}(\delta, g_2)$. Thus, by definition $q \preceq \mathit{Sat}(\delta, g)$.

   Suppose $q = \mathbf{f}$, i.e. $\delta \in \Delta^{\mathbf{f}}(g)$. Then, either (or both) $\delta \in \Delta^q(g_1)$ or $\delta \in \Delta^q(g_2)$. Suppose (without loss of generality) that $\delta \in \Delta^q(g_1)$. By induction $\mathbf{f} \preceq \mathit{Sat}(\delta_1, g_1)$. Trivially, $\perp \preceq \mathit{Sat}(\delta_2, g_2)$. Thus, $\mathbf{f} \wedge \perp \preceq \mathit{Sat}(\delta_1, g_1) \wedge \mathit{Sat}(\delta_2, g_2) = \mathit{Sat}(\delta, g)$. But $\mathbf{f} \wedge \perp = \mathbf{f}$ which concludes the proof.

3. Let $g = \texttt{not}\, g_1$. Then $\delta \in \Delta^{\neg q}(g_1)$. By induction $\neg q \preceq \mathit{Sat}(\delta, g_1)$. Since $\mathit{Sat}(\delta, g) = \neg \mathit{Sat}(\delta, g_1)$. this implies that $\neg q \preceq \neg \mathit{Sat}(\delta, g)$. Hence $q \preceq \mathit{Sat}(\delta, g)$.

4. Let $g = \texttt{Next}\, g_1$. Then by construction of Δ it must be that $s_0 = \mathbf{X}$ and that $s_1 s_2 \ldots \in \Delta^q(g_1)$. By induction $q \preceq \mathit{Sat}(s_1 s_2 \ldots, g_1)$. Thus $q \preceq \mathit{Sat}(\mathbf{X} s_1 s_2 \ldots, \texttt{Next}\, g_1)$.

5. Suppose $g = g_1 \,\texttt{Until}\, g_2$.
   By definition $\mathit{Sat}(\tilde{s}, g_1 \,\texttt{Until}\, g_2) = \vee_{i=0}^{\infty}(\mathit{Sat}(\tilde{s}_0, g_1) \wedge \ldots \wedge \mathit{Sat}(\tilde{s}_{i-1}, g_1) \wedge \mathit{Sat}(\tilde{s}_i, g_2))$. Let $\delta \in \Delta^q(g)$ be given.

   (a) Suppose $q = \mathbf{t}$, i.e. $\delta \in \Delta^{\mathbf{t}}(g_1 \,\texttt{Until}\, g_2)$.
   Then $\exists i$ such that $\delta \in \Delta^{\mathbf{t}}(\texttt{Next}\, 0\, g_1) \sqcup \ldots \sqcup \Delta^{\mathbf{t}}(\texttt{Next}\,(i-1)\, g_1) \sqcup \Delta^{\mathbf{t}}(\texttt{Next}\, i\, g_2)$.
   Thus, $\forall j = 0, \ldots, i-1, \exists \delta_i \in \Delta^{\mathbf{t}}(\texttt{Next}\, j\, g_1)$ such that $\delta_j \sqsubseteq \delta$.
   By induction, $\mathbf{t} \preceq \mathit{Sat}(\delta_j, \texttt{Next}\, j\, g_1)$
   Therefore, $\mathbf{t} \preceq \mathit{Sat}(\delta, \texttt{Next}\, j\, g_1)$
   Similarly $\mathbf{t} \preceq \mathit{Sat}(\delta, \texttt{Next}\, i\, g_2)$
   Thus, $\mathbf{t} \preceq \mathit{Sat}(\delta, (\texttt{Next}\, 0\, g_1) \wedge \ldots \wedge (\texttt{Next}\,(i-1)\, g_1) \wedge (\texttt{Next}\, i\, g_2))$.
   So, $\mathbf{t} \preceq \mathit{Sat}(\delta, g_1 \,\texttt{Until}\, g_2)$.

(b) Suppose $q = \mathbf{f}$, i.e. $\delta \in \Delta^{\mathbf{f}}(g_1\, \mathtt{Until}\, g_2)$.
Then $\forall i = 0, \ldots, \exists \delta_i$ with

$\delta_i \sqsubseteq \delta$
$\delta_i \in \Delta^{\mathbf{f}}(\mathtt{Next}\,0\, g_1) \cup \ldots \cup \Delta^{\mathbf{f}}(\mathtt{Next}\,(i-1)\, g_1) \cup \Delta^{\mathbf{f}}(\mathtt{Next}\, i\, g_2)$

By induction $\mathbf{f} \preceq \mathit{Sat}(\delta_i, \mathtt{Next}\,0\, g_1\ \mathtt{and}\ \mathtt{Next}\,(i-1)\, g_1\ \mathtt{and}\ \mathtt{Next}\, i\, g_1)$.
Therefore, by definition of $g_1\, \mathtt{Until}\, g_2$, $\mathbf{f} \preceq \mathit{Sat}(\delta_i, g_1\, \mathtt{Until}\, g_2)$

□

Lemma A.5 Let $g \in$ TL, and let $\sigma = \sigma_0\tilde{\sigma}$. 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.

1. Suppose g is simple. Then $q \preceq g(\sigma_0)$. Since g is simple there exists $q' \in \{q, \top\}$ with $(s_{q'}, q') \in D(g)$ and $s_{q'} \sqsubseteq \sigma_0$. Thus, $s_{q'}\mathbf{X} \ldots \sqsubseteq \sigma$. But $s_{q'}\mathbf{X} \ldots \in \Delta^q(g)$ by definition of $\Delta^q(g)$.

2. Suppose $g = g_1\ \mathtt{and}\ g_2$.

   (a) Suppose $q = \mathbf{t}$. Then $\mathbf{t} \preceq \mathit{Sat}(\sigma, g_1)$ and $\mathbf{t} \preceq \mathit{Sat}(\sigma, g_2)$. By induction, $\exists \delta_1 \in \Delta^{\mathbf{t}}(g_1), \delta_2 \in \Delta^{\mathbf{t}}(g_2)$ with $\delta_1, \delta_2 \sqsubseteq \sigma$. Therefore, $\delta_1 \sqcup \delta_2 \sqsubseteq \sigma$. But, by definition of $\Delta(g)$,$\delta_1 \sqcup \delta_2 \in \Delta^{\mathbf{t}}(g)$.

   (b) Suppose $q = \mathbf{f}$. Then either (or both) $\mathbf{f} \preceq \mathit{Sat}(\sigma, g_1)$ or $\mathbf{f} \preceq \mathit{Sat}(\sigma, g_2)$. Without loss of generality assume $\mathbf{f} \preceq \mathit{Sat}(\sigma, g_1)$. By induction, $\exists \delta_1 \in \Delta^{\mathbf{f}}(g_1)$ with $\delta_1 \sqsubseteq \sigma$. By definition of $\Delta(g)$,$\delta_1 \in \Delta^{\mathbf{f}}(g)$.

3. Suppose $g = \mathtt{not}\, g_1$. If $q \preceq \mathit{Sat}(\sigma, g)$, then $\neg q \preceq \mathit{Sat}(\sigma, \mathtt{not}\, g_1)$. By induction $\exists \delta \in \Delta^{\neg q}(g_1)$ with $\delta \sqsubseteq \sigma$. But by definition of $\Delta(g)$, $\delta \in \Delta^q(g)$.

4. Suppose $g = \mathtt{Next}\, g_1$. If $q \preceq \mathit{Sat}(\sigma, g)$, then $q \preceq \mathit{Sat}(\tilde{\sigma}, g_1)$. By induction $\exists \delta \in \Delta^q(g_1)$ with $\delta \sqsubseteq \tilde{\sigma}$. By construction of $\Delta(g)$, $\mathbf{X}\delta \in \Delta^q(g)$. Since $\mathbf{X} \sqsubseteq \sigma_0$, $\mathbf{X}\delta \sqsubseteq \sigma$.

5. Suppose $g = g_1\, \mathtt{Until}\, g_2$

   (a) Suppose $q = \mathbf{t}$. Then $\exists i$ such that $\mathbf{t} \preceq \mathit{Sat}(\sigma, \mathtt{Next}\,0\, g_1) \wedge \ldots \wedge \mathit{Sat}(\sigma, \mathtt{Next}\,(i-1)\, g_1) \wedge \mathit{Sat}(\sigma, \mathtt{Next}\, i\, g_2))$
   Thus, $\mathbf{t} \preceq \mathit{Sat}(\sigma, \mathtt{Next}\, i\, g_2)$ and
   $\forall j = 0, \ldots, i-1, \mathbf{t} \preceq \mathit{Sat}(\sigma, \mathtt{Next}\, j\, g_1)$
   Thus, by induction:
   (1) $\exists \delta_i \in \Delta^{\mathbf{t}}(\mathtt{Next}\, i\, g_2)$ such that $\delta_i \sqsubseteq \sigma$
   (2) $\forall j = 0, \ldots, i-1, \exists \delta_j \in \Delta^{\mathbf{t}}(\mathtt{Next}\, j\, g_1)$ such that $\delta_j \sqsubseteq \sigma$
   Hence, $\delta = \delta_0 \sqcup \ldots \sqcup \delta_i \in \Delta^{\mathbf{t}}(\mathtt{Next}\,0\, g_1) \sqcup \ldots \sqcup \Delta^{\mathbf{t}}\mathtt{Next}\,(i-1)\, g_1 \sqcup \Delta^{\mathbf{t}}\mathtt{Next}\, i\, g_2$ and $\delta \sqsubseteq \sigma$.
   Thus, $\delta \in \Delta^{\mathbf{t}}(g_1\, \mathtt{Until}\, g_2)$ and $\delta \sqsubseteq \sigma$.

   (b) Suppose $q = \mathbf{f}$. Then $\forall i, \mathbf{f} \preceq \mathit{Sat}(\sigma, \mathtt{Next}\,0 g_1) \wedge \ldots \wedge \mathit{Sat}(\sigma, \mathtt{Next}\,(i-1) g_1) \wedge \mathit{Sat}(\sigma, \mathtt{Next}\, i g_2))$.
   Thus $\forall i$, either $\mathbf{f} \preceq \mathit{Sat}(\sigma, \mathtt{Next}\, i\, g_2)$ or $\exists j \in 0, \ldots, i-1$ such that $\mathbf{f} \preceq \mathit{Sat}(\sigma, \mathtt{Next}\, j\, g_1)$.
   Thus by induction $\forall i$, either $\exists \delta_i'$ such that $\delta_i' \in \Delta^{\mathbf{f}}(\mathtt{Next}\, i\, g_2)$ with $\delta_i' \sqsubseteq \sigma$, or $\exists \delta_j \in \Delta^{\mathbf{f}}(\mathtt{Next}\, j\, g_1)$ and $\delta_j \sqsubseteq \sigma$.

In either case, $\forall i$ by construction

$\exists \delta_i \in \Delta^{\mathbf{f}}(\texttt{Next}\,0\,g_1) \cup \ldots \cup \Delta^{\mathbf{f}}(\texttt{Next}\,(i-1)\,g_1) \cup \Delta^{\mathbf{f}}(\texttt{Next}\,i\,g_2)$

with $\delta_i \sqsubseteq \sigma$. Hence as $\mathcal{S}$ is a complete lattice,

$\delta = \sqcup_{i=0}^{\infty} \delta_i \in \Delta^{\mathbf{f}}(g_1\,\texttt{Until}\,g_2)$ and $\delta \sqsubseteq \sigma$.

($\Longleftarrow$) Let $g \in \mathrm{TL}, \sigma \in \mathcal{S}^\omega$, and assume that $\exists \delta_g \in \Delta^q(g)$ such that $\delta_g \sqsubseteq \sigma$.
By Lemma A.4, $q \preceq \mathit{Sat}(\delta_g, g)$. By the monotonicity of Sat, $q \preceq \mathit{Sat}(\sigma, g)$. □

Proof of Lemma 5.3

Lemma A.6 Let $g \in \mathrm{TL}$, and let $\sigma = \sigma_0\tilde{\sigma}$ be a trajectory. For $q = \mathbf{t}, \mathbf{f}$, $q \preceq \mathit{Sat}(\sigma, g)$ if and only if $\exists \tau_g \in T^q(g)$ with $\tau_g \sqsubseteq \sigma$.

($\Longrightarrow$) Suppose $q \preceq \mathit{Sat}(\sigma, g)$.
By lemma 5.2, $\exists \delta_g \in \Delta^q(g)$ 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 (we use $\sigma[i]$ to refer to the i-th state in σ).

1. $\tau_g[0] = \delta_g[0] \sqsubseteq \sigma[0]$
2. Assume $\tau_g[i] \sqsubseteq \sigma[i]$.
3. Since σ is a trajectory,
   $\mathbf{Y}(\tau_g[i]) \sqsubseteq \mathbf{Y}(\sigma[i]) \sqsubseteq \sigma[i+1]$

   Since $\delta_g \sqsubseteq \sigma$,
   $\delta_g[i+1] \sqsubseteq \sigma[i+1]$. Therefore, $\tau_g[i+1] = \delta_g[i+1] \sqcup \mathbf{Y}(\tau_g[i]) \sqsubseteq \sigma[i+1]$.

($\Longleftarrow$) Suppose $\exists \tau_g \in T^q(g)$ such that $\tau_g \sqsubseteq \sigma$.
By transitivity, $\delta_g \sqsubseteq \sigma$. As $\tau_g \in T^q(g), \exists \delta_g \in \Delta^q(g)$ such that $\delta_g \sqsubseteq \tau_g$.
By lemma 5.2, $q \preceq \mathit{Sat}(\delta_g, g)$.
By monotonicity $q \preceq \mathit{Sat}(\sigma, g)$. □

Proof of Theorem 5.4

Theorem A.7 Let $g, h \in \mathrm{TL}, q \in \mathcal{B}$.
$\Delta^q(h) \sqsubseteq T^q(g)$ if and only if for every trajectory σ with $q \preceq \mathit{Sat}(\sigma, g)$ it is the case that $q \preceq \mathit{Sat}(\sigma, h)$

Proof: ($\Longrightarrow$) Suppose $\forall \tau_g \in T^q(g), \exists \delta_h \in \Delta^q(h)$ with $\delta_h \sqsubseteq \tau_g$.
Suppose $q \preceq \mathit{Sat}(\sigma, g)$.
By lemma 5.3 $\exists \tau_g \in T^q(g)$ 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 5.2, $q \preceq \mathit{Sat}(\sigma, h)$.

($\Longleftarrow$) Suppose for all trajectories σ, $q \preceq \mathit{Sat}(\sigma, g)$ implies that $q \preceq \mathit{Sat}(\sigma, h)$.
Let $\tau_g \in T^q(g)$.

Then by lemma A.4 and monotonicity, $q \preceq Sat(\tau_g, g)$.
By assumption, $q \preceq Sat(\tau_g, h)$.
By lemma 5.2, $\exists \delta_h \in \Delta^q(h)$ such that $\delta_h \sqsubseteq \tau_g$.
As τ_g was arbitrary, the proof follows. $\square$