A Simple Theorem Prover Based on Symbolic Trajectory Evaluation and OBDDs

Scott Hazelhurst
Carl-Johan H. Seger

Technical Report 93–41
12 November 1993

Department of Computer Science
University of British Columbia
Rm 201 - 2366 Main Mall
Vancouver, B.C.
CANADA V6T 1Z4

Telephone: (604) 822-3061
Fax: (604) 822-5485

A Simple Theorem Prover Based on Symbolic Trajectory Evaluation and OBDDs[1]

Scott Hazelhurst
Carl-Johan H. Seger
Department of Computer Science
University of British Columbia
Vancouver, B.C., Canada V6T 1Z4

Technical Report: 93–41

12 November 1993

Abstract

Formal hardware verification based on symbolic trajectory evaluation shows considerable promise in verifying medium to large scale VLSI designs with a high degree of automation. However, in order to verify today's designs, a method for composing partial verification results is needed. One way of accomplishing this is to use a general purpose theorem prover to combine the verification results obtained by other tools. However, a specialised purpose theorem prover is more attractive since it can more easily exploit symbolic trajectory evaluation (and may be easier to use). Consequently we explore the possibility of developing a much simpler, but more tailor made, theorem prover designed specifically for combining verification results based on trajectory evaluation. In the paper we discuss the underlying inference rules of the prover as well as more practical issues regarding the user interface. We finally conclude with a couple of examples in which we are able to verify designs that could not have been verified directly. In particular, the complete verification of a 64 bit multiplier takes approximately 15 minutes on a Sparc 10 machine.

1 Introduction

The verification of computer systems has become more important as computer systems grow in complexity and range of use. Verification — the proving under some mathematical theory of the existence (or non-existence) of properties in the system — comes at a cost: it is a difficult and computationally intensive task. Although significant advances have been made, all verification methods suffer from this problem in some way.

This paper presents a new technique based on trajectory evaluation which reduces the computational cost of verification in many cases. In this section, we present background material, and introduce and motivate the new method.

[1]This research was supported by operating grant OGPO 109688 from the Natural Sciences Research Council of Canada, a fellowship from the Advanced Systems Institute, and by research contract 92-DJ-295 from the Semiconductor Research Corporation.

1.1 Hardware Verification

The theory of all the verification methods discussed in this paper applies equally to software and hardware systems. However, the application of the theory has been more successful with hardware than software since the regular nature of design and the small number of types of components makes verification of hardware systems in general more tractable than more general systems.

A survey of the large number of different hardware verification techniques which have been proposed is beyond the scope of this paper (an introduction to the topic and a good set of references to the topic can be found in (McFarland, 1993)). The work presented here draws on three techniques.

1. *Theorem proving:* Theorem-provers are based on formal systems such as logic. For hardware verification, both the specification and implementation can be described in logic, and the task of verifying the system is to prove that the implementation entails the specification. One of the best known theorem-provers is the HOL system (Gordon, 1993), with which theories of different sorts can be built up in a rigorous way using a small number of primitive axioms and inference rules.

   The key advantage of the approach is that the proof that the implementation entails the specification can be checked mechanically. We can also have a high degree of confidence in the soundness of a system like HOL due to the small number of primitive axioms and inference rules. Another important advantage of theorem proving is that it can be used to argue at different levels of abstraction. Theorem proving is structural rather than behavioural, so that we can use the structure of the system to manage the complexity. Unfortunately, structure may not always be present in a system in the required degree (for example, optimisation of a circuit may transform a circuit with a high degree of structure into one with a low degree of structure).

   The most significant disadvantage of theorem proving by itself is that it can be extremely tedious to verify certain properties, particularly low-level properties of circuits. Unless simplifications are made, this approach can be very expensive.

   Besides HOL, there are a number of well-known theorem-provers, including PVS (Owre *et al.*, 1992) and the Boyer-Moore system (Boyer & Moore, 1988),

2. *Model checking:* In model checking, the behaviour of the system is described by some model (for example, a Kripke structure, a finite state machine, expressions in a process algebra like CCS). The desired properties of a system are expressed in some logic (for example, CTL, the modal μ-calculus) (Clarke *et al.*, 1983; Cleaveland *et al.*, 1989; Coudert *et al.*, 1990; Burch *et al.*, 1992). A model checking algorithm can be used to check whether the model satisfies such an expression. Depending on how restricted the model is (the state space of the model can be very large — even infinite), and the language we are allowed to write the logical expressions, model checking can be efficient. There are important limitations to what can be checked efficiently (as an example the data path of a circuit poses a computational challenge). Furthermore, while it may be very convenient to write expressions in a logic, it is not always clear that the collection of properties actually verified is what we want to verify. Model checking is based on the behavioural rather than structural properties of the system.

   Many recent model-checking algorithms use ordered binary-decision diagrams (OBDDs or BDDs) (Bryant, 1992) for the efficient manipulation of boolean expressions. One of the advantages of BDDs is that boolean expressions have canonical forms which makes comparison of expressions very efficient. Although BDDs do have practical limitations, the use of BDD-based method has extended by orders of magnitude the size of systems that can be tackled by model-checkers.

3. *Trajectory Evaluation:* If the state space of a system (for example a circuit) is a lattice, the behaviour of the system can be expressed as a *trajectory*, a sequence of points in the lattice determined by the initial state and the system functionality. Formulas in a simple temporal logic express properties of the circuit. Given a formula, we can derive bounds that trajectories with the desired property must obey. Symbolic trajectory evaluation efficiently checks whether all, or a specified class, of trajectories of the machine obey these bounds. Generally it checks that all trajectories of a machine $\mathcal{M}$ which have property A also have property C. We write this as $\models_{\mathcal{M}} [A \Longrightarrow\!\!\!> C]$, and say that $\mathcal{M}$ satisfies the trajectory assertion.

   Trajectory evaluation has several advantages. Verification is behavioural rather than structural, and thus symbolic trajectory evaluation is suitable for systems such as optimised circuits. Very accurate models, including timing, of the underlying model can be used, which increases our confidence in the meaningfulness of the results obtained. Symbolic trajectory evaluation also uses BDDs as a powerful method of manipulating boolean expressions.

   A full description of trajectory evaluation is given in a later section. Although many circuits can be verified very efficiently, there are, however, some limitations. It is these limitations which have motivated this work. Introductory references for symbolic trajectory evaluation include (Beatty *et al.*, 1991; Seger & Bryant, 1993).

1.2 Motivation

Although trajectory evaluation is a very successful approach for verifying many realistic circuits, two weaknesses have been noted of this method.

First, there can be a semantic gap between the trajectory evaluation verification and what the user has in mind to verify. Part of the problem is that the proof is structured at the bit-level, when it may be more natural to represent the proof in terms of a higher-level domain, for example arguing about the behaviour of the circuit in terms of what it does to integers rather than bit-vectors. And, unlike with theorem-proving, we cannot exploit any form of structure to guide the proof.

Second, it is still computationally expensive and approaches that would reduce this cost would be useful. In particular, due to the underlying data representation techniques used in symbolic trajectory evaluation and the inability to decompose the proof, there are some types of circuits which are in practice unverifiable using this approach solely — for example a multiplier circuit.

Our approach offers a number of advantages. We believe that the compositional approach makes the verification easier for the user, particularly for very large systems. Second, verifying the individual components separately is much more efficient for a variety of reasons (discussed in later sections). Third, composing results using domain-information makes composition efficient and practical. Fourth, different techniques can be used for verification of parts of the system, and for verification as a whole, thereby exploiting the strengths of different verification approaches and overcoming their limitations.

As an example, consider the circuit in Figure 1, which takes in three numbers x, y and z on nodes A, B and C and outputs $max(x, y) + z$ on node F. The circuit comprises three major parts: a comparator, a selector, and an adder. Although this is a simple example which can easily be verified by symbolic trajectory evaluation alone, it can illustrate some of these ideas. Each component is separately verified (with the advantages mentioned above). The next step is to verify that the comparator combined with the selector produces the greater of the two input values. The final step is to combine this with the adder to show the final result is correct.

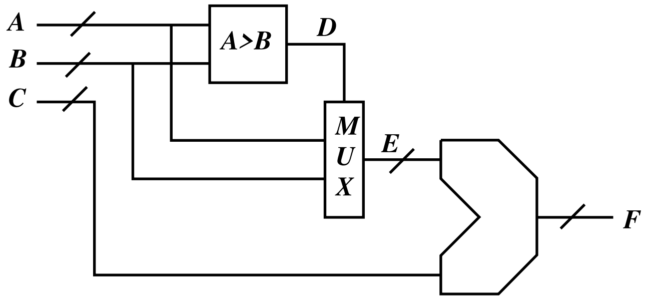

Figure 1: Illustration of symbolic trajectory evaluation

The verification of each component is done in the presence of the rest of the system, which means that any unintended interference can be detected. We can easily show that no matter what the rest of the system does we shall obtain the desired behaviour. This is a very useful property, since much of the work involved in other compositional approaches goes into ensuring that each component works correctly in all environments.

This paper explores combining theorem-proving and trajectory evaluation so as to gain the benefits of both approaches, and overcome the problems noted. An important objective in our work is to provide a practical tool without losing rigour.

The method that we propose is a theorem proving system in which the mathematical objects being manipulated are trajectory assertions. In later sections, the theory of trajectory evaluation and the theory of how they can be combined and manipulated will be discussed. Here we give a brief overview of theorem proving.

The work on theorem provers first started in the late 1960s and early 1970s with the work of de Bruijn and Milner. The core of a theorem prover is a set of axioms and inference rules. Using only these, the user can prove a theorem, with the system mechanically checking each step in a proof. While good theorem provers have tools which assist a user in the derivation of a proof, automatic derivation of proofs is usually not their goal.

There are different ways in which this assistance can be given. Perhaps the most important part of the utility of these tools is the way in which they can interact with the user. In HOL, for example, the user can write proof scripts in ML. This programmability alleviates some of the tedium of theorem proving and makes the tool much more flexible. In our tool too, the user has a fully programmable script language in which to specify proofs.

Proof management can also be accomplished with the use of what in HOL are called "tactics". This enables the decomposition of proofs: working backwards from the goal, the appropriate tactics are applied to the theorem to discover sub-results, which, if proven, can be used to prove the final result. It should be stressed that skilful human intervention is critical. Using the proof script language in our system, it is possible to create similar building blocks.

A third type of assistance which can be provided is the use of external, non-trusted programs. For example, in HOL a theory of differentiation has been built up. However, integration within HOL is extremely difficult, so one way of doing integration is to use a symbolic mathematics package to generate the answer. We do not trust this answer, however, because the result has not been proven within HOL. However, we can use differentiation within HOL to check the answer (Harrison & Thery, 1993).

We use these ideas in two ways. First, we use a simple auxiliary theorem prover about integers so as to incorporate domain knowledge. Secondly, our tool occasionally uses a heuristic to "guess" an answer, which is then automatically checked to see whether it is correct.

1.3 Outline

Our goal is to exploit the strengths of symbolic trajectory evaluation, to overcome some its weaknesses, and create a methodology for the composition and management of proof results.

The rest of this paper is structured as follows.

- Section 2 describes related work.
- In Section 3 , after the underlying theory of symbolic trajectory evaluation is described, an efficient method of performing trajectory evaluation is given.
- Section 4 describes the new theory — the rules of inference which allow the combination of symbolic trajectory evaluation results. Whereas symbolic trajectory evaluation is a purely behavioural method, for our new theory to be useful, it is important that some structure be readily identifiable. However, we are not moving to a purely structural model since our method deals more with the composition of the specification and the proof than the composition of the underlying circuit. So although we need to be able to use some structure in our system, we have neither to build a structural model of the system, nor describe the behaviour of the component parts.
- Earlier we noted the limitations of symbolic trajectory evaluation with respect to the low-level of abstraction, and the limitations of the underlying data representation. Section 5 discusses how these problems can be overcome in a rigorous and practical way.
- Section 6 describes the practical verification tool developed using the theory described in the earlier sections. With this tool, we are able to use symbolic trajectory evaluation to perform partial verification, and the combinational theory to combine these results.
- Section 7 presents several examples, and section 8 is a conclusion.

2 Related Work

There are two important influences on this work, compositionality and use of hybrid methods.

2.1 Compositionality

Compositionality is the divide-and-conquer approach of verification. It allows re-use of results and, most importantly, it greatly reduces the state space of systems that need to be explored. Since a human verifier is able to structure the verification, it makes verification easier to manage. Compositionality has been examined from a number of different aspects:-

- Process algebras such as CCS (Milner, 1989) and CSP (Brookes *et al.*, 1984) have as one of their major advantages the ability to define and prove properties of programs and specification using compositionality. Work continues on making verification through compositionality more efficient in process algebras and more general transition systems (Larsen & Thomsen, 1991; Groote & Moller, 1992)
- In model checking approaches too, there has been work in using compositionality to control the state space of the system (Coudert *et al.*, 1990; Clarke *et al.*, 1989; Shiple *et al.*, 1992; Long, 1993).
- For theorem-provers, composition at a structural level is very important for efficient and easy verification. The use of different forms of logic promotes the ease of verification.

This work shows how symbolic trajectory evaluation results can be composed.

2.2 Hybrid Methods and Proof Management

By hybrid methods, we mean the combined use of different verification methods. *Ad hoc* hybrid approaches where designers use different techniques to verify and test different parts of the system and use informal argument to glue the pieces together is not new and very common. What we are aiming at is something of greater rigour.

One of the first systems which combined different approaches rigorously was the HOL-Voss system (Seger & Joyce, 1992). The HOL theorem-prover and the Voss symbolic trajectory evaluation system were linked together in a formal, rigorous way. This enables the proof of something within one system to be carried over into the other system.

Although not a hybrid approach, the method of model-checking infinite state-spaces suggested by Bradfield illustrates the utility of providing a method of combining automatic verification and human-driven (but mechanically checked) verification (Bradfield, 1992). Although very different from our work, the common theme is that a practical yet rigorous proof management system is very useful in verification.

Recent work by Kurshan and Lamport is much closer to our approach (Kurshan & Lamport, 1993). They have combined the COSPAN model-checker with the TLP theorem prover. The model checker is used to prove properties of components of the system: these properties are then translated into a form suitable for the theorem-prover. In order to prove the overall result, a number of sub-results need to be proved (including a hand-checked step).

The common link between our approaches is the recognition that different techniques can be useful in different places to overcome the fundamental limitations of each individual approach. Beyond that it is difficult to compare our work since the two are very different. Some of the differences and difficulties in comparison are:

- In our approach, all verification results are expressed and have meaning within the theory of symbolic trajectory evaluation. The theorem-prover allows efficient inference of some of these results, but we do not have to integrate one theory in another, or translate one formalism into another (mechanically or manually).

- Related to this is that we have one integrated tool for verification, and all steps are automatic or machine checked.

- We can verify circuits at a much lower-level than in the COSPAN/TLP approach. This makes our approach much less dependent on the structural properties of the system and its specification. It also means that we can capture the timing constraints very accurately[2].

- As for usability, we believe that our approach is much easier to understand, since the mathematics a user must understand is much simpler. Of course, this is a very subjective issue, and much can be dealt with by providing a pleasant user interface.

- No performance figures are given for the COSPAN/TLP verification of the 64-bit multiplier which means we cannot compare performance. Furthermore, attractive as the fleeting glory of having the fastest 64-bit multiplier verification in the world may be, it is not clear that this is a useful benchmark — rather, it shows that a class of circuits unverifiable using forms of model-checking alone are now quite tractable.

Another recent proposal which combines model checking and theorem-proving is reported in (Hungar, 1993). Here, the model is given by a Kripke structure representing the semantics of an Occam program, and the properties are expressed in terms of a variant of CTL. The results generated by model-checking are combined using the LAMBDA theorem-prover. Given an Occam program consisting of a number of processes, using the model-checker, properties can be proven of each process. A number of rules — some analogous to the ones we propose in Section 4 — can then be used by the theorem-prover to combine these sub-properties to prove properties of the entire program. The work was reported as work in progress, so it is difficult to evaluate this properly in relation to our work, but the motivation behind the work is similar to ours.

3 The Theory Behind Trajectory Evaluation

In this section we shall give a brief introduction to the theory behind symbolic trajectory evaluation. For the complete theory, the interested reader is referred to (Seger & Bryant, 1993). We shall assume the reader has a working knowledge of elementary concepts from lattice theory. In particular, the concepts of partial orders, lattices, monotone functions, etc. will be used without further explanation.

The model we use of a system is simple and general. A *model structure* is a tuple $\mathcal{M} = [\langle \mathcal{S}, \sqsubseteq \rangle, Y]$, where $\langle \mathcal{S}, \sqsubseteq \rangle$ is a complete lattice and Y is a monotone successor function $Y: \mathcal{S} \rightarrow \mathcal{S}$. Intuitively, the successor function is used to express constraints on the sequence of states a system may go through. In other words, given that the system is in state $s \in \mathcal{S}$, we view $Y(s)$ as denoting the least specified state the system can be in one time unit later. Here, "least specified" is defined in terms of the partial order $\sqsubseteq$.

[2]This issue is not discussed in (Kurshan & Lamport, 1993) so we are not saying that they can not handle the timing, just that this is not done in this paper.

Let $\mathcal{S}^\omega$ denote the set of all infinite sequences of elements from $\mathcal{S}$. Sequences are useful when reasoning about model behaviours, but not all sequences represent possible behaviours of a model. The successor function generally restricts the possible sequences significantly. We formalise this property by introducing the concept of a trajectory. Given a model $\mathcal{M}$ and an arbitrary sequence $\sigma = \sigma^0\sigma^1 \ldots \in \mathcal{S}^\omega$ we say that the sequence is a *trajectory* if and only if

$$Y(\sigma^i) \sqsubseteq \sigma^{i+1} \text{ for } i \geq 0.$$

This rule for trajectories is consistent with our view of the successor function, i.e., a function computing a constraint on the possible value of the successor state. The set of all trajectories of model $\mathcal{M}$ is denoted $L(\mathcal{M})$.

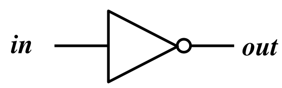

Figure 2: Unit delay inverter.

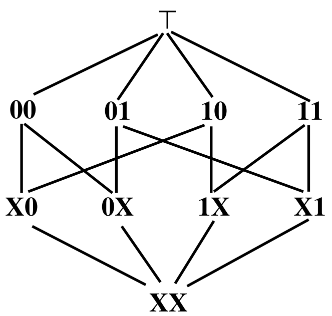

Figure 3: Poset for inverter circuit.

To illustrate the ideas introduced so far, consider the circuit shown in Fig. 2. If we assume a unit delay model of the inverter, the state of the system consists of the value of the input and the current value of the output. Consequently, the lattice $\mathcal{S}$ shown in Fig. 3 can be used to model the circuit behaviour. Since the inverter cannot impose any restrictions on its input, it follows that the next state function will always be X for the input node, i.e., $Y_{in}(i, o) = X$. For the output node, the next state function will be defined as:

$$Y_{out}(i, o) = \begin{cases} 0 & \text{if } i = 1 \\ 1 & \text{if } i = 0 \\ X & \text{if } i = X \\ \top & \text{if } i = \top \end{cases}$$

It is easy to convince oneself that $Y(i, o) = (Y_{in}(i, o), Y_{out}(i, o))$ is a monotone function. Thus, $[\langle \mathcal{S}, \sqsubseteq \rangle, Y]$ is a model structure. In Fig. 4 we illustrate the set of all trajectories ($L(\mathcal{M}^{\mathcal{C}})$) given this model structure.

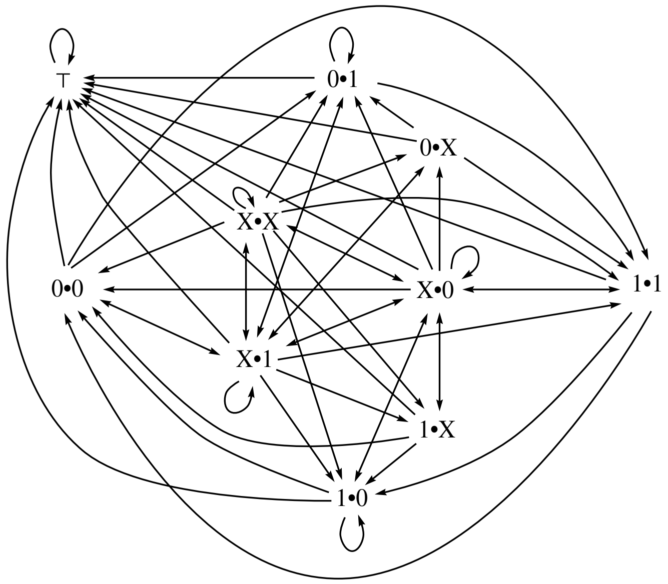

Figure 4: $L(\mathcal{M}^{\mathcal{C}})$ for a unit delay inverter.

The key to the efficiency of trajectory evaluation is the restricted language that can be used to phrase questions about the model structure. The basic specification language we use is very simple. Although at first glance it might appear as if it can only be used to specify rather trivial behaviours, this is not the case, particularly as we later extend the model structure to a symbolic domain. This will essentially allow us to represent in a very concise and efficient way a very large set of cases simultaneously. By keeping the underlying logic simple, we gain some important properties. The most important is that there is a unique weakest trajectory that satisfies a formula. By focusing initially on the scalar version, we avoid the added complexity of the symbolic case while building a foundation for this more general formulation.

Before describing trajectory formulas, we need to introduce the concept of a simple predicate. A *predicate* over $\mathcal{S}$ is a mapping from $\mathcal{S}$ to the complete lattice with elements *false* and *true*, with *false* as the lower bound and *true* as the upper bound. A predicate is said to be *simple* iff p is monotone and there is a unique element $\overline{p} \in \mathcal{S}$, called the *defining value of* p, such that $p(t) = \mathit{true}$ iff $\overline{p} \sqsubseteq t$ for all $t \in \mathcal{S}$. Another way of stating this property is that p is a simple predicate iff p is monotone and $p(\mathit{glb}(\{s \in \mathcal{S} | p(s) = \mathit{true}\})) = \mathit{true}$.

In circuit verification the natural simple predicates are of the following form:

1. (n_i **is** 0) where $n_i \in \mathcal{N}$, and

2. (n_i **is** 1) where $n_i \in \mathcal{N}$,

where $\mathcal{N}$ is the set of nodes that determines the state of the circuit (informally, a predicate describes a potential state of the system). It is easy to see that (n_i **is** 0) and (n_i **is** 1) are indeed simple with defining values

$$\langle X, \ldots, X, 0, X, \ldots, X\rangle$$

and

$$\langle X, \ldots, X, 1, X, \ldots, X \rangle,$$

where the 0 (1) is in position i.

Assume $\langle \mathcal{S}, \sqsubseteq \rangle$ is a lattice with universal lower bound $\bot$. Let $\mathcal{P}$ denote a set of simple predicates over $\mathcal{S}$. A *trajectory formula* is defined recursively as:

1. **Simple predicates**: p is a trajectory formula if $p \in \mathcal{P}$.
2. **Conjunction**: $(F_1 \wedge F_2)$ is a trajectory formula if F_1 and F_2 are trajectory formulas.
3. **Domain restriction**: $(e \rightarrow F)$ is a trajectory formula if F is a trajectory formula and e is either 0 or 1.
4. **Next time**: $(\mathbf{N}F)$ is a trajectory formula if F is a trajectory formula.

The truth semantics of a trajectory formula is defined relative to a model structure and a trajectory. In particular, given a model structure $\mathcal{M}$ and a trajectory σ, the truth of a trajectory formula F, written $\sigma \models_{\mathcal{M}} F$, is defined recursively. In the following, assume that both σ and $\sigma^0\tilde{\sigma}$ are members of $L(\mathcal{M})$.

1. $\sigma^0\tilde{\sigma} \models_{\mathcal{M}} p$ iff $p(\sigma^0) = \mathit{true}$.
2. $\sigma \models_{\mathcal{M}} (F_1 \wedge F_2)$ iff $\sigma \models_{\mathcal{M}} F_1$ and $\sigma \models_{\mathcal{M}} F_2$
3. (a) $\sigma \models_{\mathcal{M}} (1 \rightarrow F)$ iff $\sigma \models_{\mathcal{M}} F$
   (b) $\sigma \models_{\mathcal{M}} (0 \rightarrow F)$ holds for every σ.
4. $\sigma^0\tilde{\sigma} \models_{\mathcal{M}} \mathbf{N}F$ iff $\tilde{\sigma} \models_{\mathcal{M}} F$.

We can extend the definition of simplicity from predicates to formulas in the obvious way, i.e., given a model structure $\mathcal{M}$, a formula F is said to be *simple* iff there is a *defining trajectory* $\overline{\sigma} \in L(\mathcal{M})$ such that $\sigma \models_{\mathcal{M}} F$ iff $\overline{\sigma} \sqsubseteq \sigma$. The main observation behind trajectory evaluation is that trajectory formulas are simple. In fact, we shall give a construction that given a trajectory formula F constructs the defining sequence. The construction is direct and very efficient. As a result, if the main verification task can be phrased in terms of "for every trajectory σ that satisfies the trajectory formula A, verify that the trajectory also satisfies the formula C", it becomes obvious how the verification can be carried out: compute the defining trajectory for the formula A and check that the formula C holds for this trajectory.

Before giving the construction, it is convenient to introduce an infix "choice" function mapping $\{0,1\} \times \mathcal{S}^\omega$ to $\mathcal{S}^\omega$ and which is defined as:

$$e?\delta = \begin{cases} \delta & \text{if } e = 1 \\ \bot\bot\ldots & \text{otherwise} \end{cases}$$

We now show that given a trajectory formula F we can construct its *defining sequence* δ_F. This sequence is the weakest possible in the sense that $\sigma \models_{\mathcal{M}} F$ iff $\delta \sqsubseteq \sigma$. Note that δ_F is not necessarily a trajectory. We define δ_F recursively as follows:

1. $\delta_p = \overline{p}\,\bot\bot\ldots$ if $p \in \mathcal{P}$ is a simple predicate with defining value $\overline{p}$.

2. $\delta_{F_1 \wedge F_2} = lub(\delta_{F_1}, \delta_{F_2})$.

3. $\delta_{e \rightarrow F} = e?\delta_F$.

4. $\delta_{\mathbf{N}F} = \bot\,\delta_F$.

In general, we have the following result.

Lemma 3.1 *For any trajectory formula F let δ_F be constructed as above. Then for every $\sigma \in L(\mathcal{M})$*

$$\sigma \models_{\mathcal{M}} F \iff \delta_F \sqsubseteq \sigma$$

Proof: For a proof of this result, the reader is referred to (Seger & Bryant, 1993).

□

From this lemma we know that any trajectory satisfying F must be greater than or equal to the defining sequence δ_F. Thus computing δ_F and then determining if a trajectory is greater than or equal to δ_F allows us to quickly test whether the trajectory satisfies the formula F. However, δ_F is not necessarily itself a trajectory, so we combine the constraints on a state sequence implied by δ_F with those imposed by the system's next-state function to give a trajectory and show that the obtained trajectory is the weakest possible trajectory satisfying F.

Assume $\delta_F = \delta_F^0 \delta_F^1 \ldots$ is the defining sequence for a formula F. Define $\tau_F = \tau_F^0 \tau_F^1 \ldots$ inductively as follows:

$$\tau_F^i = \begin{cases} \delta_F^0 & \text{if } i = 0 \\ lub(\delta_F^i, Y(\tau_F^{i-1})) & \text{otherwise} \end{cases}$$

Given this construction, the main observation regarding trajectories and trajectory formulas, is captured in the following lemma.

Lemma 3.2 *Assume τ_F is defined as above, then:*

1. *$\tau_F \in L(\mathcal{M})$,*

2. *$\tau_F \models_{\mathcal{M}} F$, and*

3. *for every $\sigma \in L(\mathcal{M})$*

$$\sigma \models_{\mathcal{M}} F \iff \tau_F \sqsubseteq \sigma$$

Proof: For a proof of this result, the reader is referred to (Seger & Bryant, 1993). □

The above lemmas give a simple method for computing the defining trajectory and the defining sequence for a trajectory formula. Although both the defining trajectory and the defining sequence are theoretically infinite sequences, it is easy to show that if F is a trajectory formula and $\delta_F = \delta_F^0 \delta_F^1 \ldots$ is the defining sequence for F then $\delta_F^i = \bot$ for $i \geq d(F)$, where $d(F)$ denotes the maximum depth of nested next-time operators in F. In practice therefore, it is sufficient to compute a bounded prefix of the defining sequence and defining trajectory.

Our specification language describes a property of the system $\mathcal{M}$ as an *assertion* of the form $[A \Longrightarrow\!\!\!> C]$, where both A and C are trajectory formulas expressing constraints on the trajectory. Unlike a trajectory formula, however, an assertion is considered to hold only if it holds for all trajectories. That is, $[A \Longrightarrow\!\!\!> C]$ holds, written $\models_{\mathcal{M}} [A \Longrightarrow\!\!\!> C]$, when for every $\sigma \in L(\mathcal{M})$ we have that $\sigma \models_{\mathcal{M}} A$ implies that $\sigma \models_{\mathcal{M}} C$.

The following theorem forms the basis for verification based on trajectory evaluation.

Theorem 3.3 *Assume A and C are two trajectory formulas. Let τ_A be the defining trajectory for formula A and let δ_C be the defining sequence for formula C. Then $\models_{\mathcal{M}} [A \Longrightarrow\!\!> C]$ iff $\delta_C \sqsubseteq \tau_A$.*

Proof: Again, for a proof of this result the reader is referred to (Seger & Bryant, 1993). □

There is one major drawback with the verification method suggested above—a single trajectory formula is not very expressive. In order to verify any non-trivial property of a system we would have to write down and verify a very large number of assertions. By making the formulas and algorithms above operate over a symbolic domain we can efficiently express and verify this set of assertions in a concise and efficient way. The key idea is to preserve the symbolic structure of the formulas in the verification algorithm. By doing so, we can perform the same algebraic manipulations and rather than obtaining a true/false answer, we shall obtain a Boolean function over some variables. This Boolean function, given a truth assignment to the Boolean variables, evaluates to 1 iff the scalar assertion corresponding to this truth assignment holds.

More specifically, let $\mathcal{V}$ be a set of symbolic boolean variables. For convenience, let $\mathcal{B}$ denote the set $\{0,1\}$. An *assignment*, ϕ, is a mapping $\phi\colon \mathcal{V} \rightarrow \mathcal{B}$ assigning a binary value to each variable. Let Φ be the set of all possible assignments, i.e., $\Phi = \{\phi\colon \mathcal{V} \rightarrow \mathcal{B}\}$. A *domain constraint*, $\mathcal{D} \subseteq \Phi$, defines a restriction on the values assigned to the variables. We will denote such domain constraints by boolean expressions. That is, let E be a boolean expression over elements of $\mathcal{V}$[3]. This expression defines a Boolean mapping $e\colon \Phi \rightarrow \mathcal{B}$ and thus denotes the domain constraint $\mathcal{D} = \{\phi \mid e(\phi) = 1\}$. The set of all assignments Φ is denoted by the constant function $\dot{1}$. Expressing domain constraints by boolean expressions allows us to compactly specify many different circuit operating conditions with a single formula.

In general, if $\mathcal{D}$ is a scalar domain set we extend it to a symbolic domain set, written $\mathcal{D}(\mathcal{V})$, by defining

$$\mathcal{D}(\mathcal{V}) = \{f\colon \Phi \rightarrow \mathcal{D}\}.$$

In other words, $\mathcal{D}(\mathcal{V})$ denotes the set of functions mapping an assignment in Φ to $\mathcal{D}$.

We extend all operations from scalar to symbolic domains in a uniform way. Consider an operation $op\colon \mathcal{D}_1 \times \mathcal{D}_2 \rightarrow \mathcal{D}_3$, defined over scalar domains $\mathcal{D}_1$, $\mathcal{D}_2$, and $\mathcal{D}_3$. Its symbolic counterpart $\dot{op}\colon \mathcal{D}_1(\mathcal{V}) \times \mathcal{D}_2(\mathcal{V}) \rightarrow \mathcal{D}_3(\mathcal{V})$ is defined such that for all $\dot{a} \in \mathcal{D}_1(\mathcal{V})$ and $\dot{b} \in \mathcal{D}_2(\mathcal{V})$, we have $(\dot{a}\ \dot{op}\ \dot{b})(\phi) = \dot{a}(\phi)\ op\ \dot{b}(\phi)$.

There are several ways of extending trajectory formulas to the symbolic domain. The approach we have taken is very simple and consists simply of allowing the domain constraint to be an arbitrary boolean expression rather than only 1 or 0. For the case of circuit simulation, we also introduce the notation $(n_i\ \mathbf{is}\ E)$ as a shorthand for the formula $(E \rightarrow (n_i\ \mathbf{is}\ 1)) \wedge (\overline{E} \rightarrow (n_i\ \mathbf{is}\ 0))$. That is, we constrain node n_i to have the particular symbolic Boolean value denoted by the expression E.

Given a symbolic trajectory formula $\dot{F}$ and an assignment $\phi \in \Phi$, the corresponding trajectory formula, written $\dot{F}(\phi)$, is defined to be the trajectory formula obtained by replacing the Boolean expressions in the domain constraints with the value of the expression for this assignment.

Given the above, we can now derive a symbolic trajectory evaluation algorithm by simply extending the operators and functions used in the scalar trajectory evaluation algorithm to the symbolic domain. Thus, given a model structure $\mathcal{M}$ and a symbolic assertion $\left[\dot{A} \Longrightarrow\!\!> \dot{C}\right]$, the checking algorithm computes the Boolean function expressing the set of assignments under which

[3] For the sake of brevity, we omit a formal syntax of boolean expressions. Any standard expression syntax suffices.

the assertion is true. For most verification problems, this should simply be the constant function **1**, i.e., the assertion should hold under all variable assignments. Again, for a more detailed discussion and proof of correctness of this result the reader is referred to (Seger & Bryant, 1993).

3.1 The Voss System

The Voss system is a formal hardware verification system being developed at the University of British Columbia by the second author. Originally intended as a simple front-end for a symbolic trajectory evaluation system, the system has evolved significantly. In essence, the Voss system consists of three major components: a highly efficient implementation of Ordered Binary Decision Diagrams; an event driven symbolic simulator with very comprehensive delay and race analysis capabilities; and a general purpose, purely functional language. The language, called FL, is a strongly typed, polymorphic, and fully lazy language. Every object of type boolean in the system is internally represented as an OBDD. Consequently, FL is a very convenient language for developing prototype verification methodologies that require OBDD manipulations. Also, since the language supports abstract data types similar to ML (Milner, 1984), the type system can beneficially be used to safeguard the implementation (Gordon *et al.*, 1979). In fact, we use exactly this feature in the system we describe in Section 6.

To illustrate the practicality of the system, we used the built-in symbolic trajectory evaluation procedure to fully verify a simple 32-bit microprocessor (an extended version of Tamarack III (Joyce, 1989; Zhu *et al.*, 1993)) and the pipelined integer unit of a 32-bit RISC architecture with a 32 registers deep register file (the circuit is a switch-level implementation of the McMillan datapath described in (Burch *et al.*, 1992)). The circuits verified contained around 16 000 transistors each and the verification processes required less than 45 minutes on a Sparc 10/51 processor with 64Mbyte of memory.

Although circuits as large as 20 000 transistors modelled at the switch-level and as complex as pipelined integer RISC cores have been verified, it is clear that other methods are needed if circuits with several order of magnitude more transistors are to be verified. One method to accomplish is the topic of the next section.

4 Inference Rules

The previous section presented the theory of trajectory evaluation, and showed how properties of a circuit could be verified. Theorem 3.3 is the key result: from $\delta_C \sqsubseteq \tau_A$, we can infer $\models_{\mathcal{M}} [A \Longrightarrow\!\!\!> C]$. In this section, we develop other rules for inferring these results, enabling the re-use and composition of old results, and the incorporation of domain knowledge.

Throughout the section we shall use the circuit shown in Fig. 5 as a simple example (our new techniques are not needed for such a simple circuit, but it illustrates the theory).

4.1 Preliminary theory

Before we can introduce the theorems needed for combining verification results, some technical results will be needed.

The first technical lemma says that a defining sequence is less than the corresponding defining trajectory.

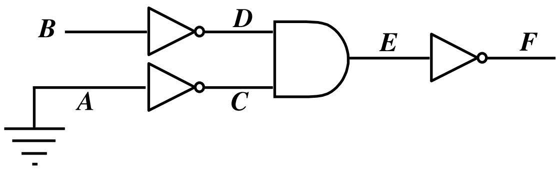

Figure 5: Circuit C_2.

Lemma 4.1 *If A is a defining formula then $\delta_A \sqsubseteq \tau_A$.*

Proof: We prove by induction on i that $\delta_A^i \sqsubseteq \tau_A^i$. For the basis, $i = 0$, the results follows trivially since $\tau_A^0 = \delta_A^0$. Hence assume inductively that $\delta_A^{i-1} \sqsubseteq \tau_A^{i-1}$ for $i \geq 1$. Since $\tau_A^i = lub(\delta_A^i, Y(\tau_A^{i-1}))$ it follows trivially from the properties of least upper bound that $\delta_A^i \sqsubseteq \tau_A^i$ and the induction goes through and the claim follows. □

The second lemma says that a defining sequence of a formula A is less than a defining trajectory of a formula C if and only if the defining trajectory of A is less than the defining trajectory of C. This result will be used when finding the equivalent of transitivity for verification results.

Lemma 4.2 (τ-δ lemma) *If A and C are trajectory formulas, then $\delta_C \sqsubseteq \tau_A \Leftrightarrow \tau_C \sqsubseteq \tau_A$.*

Proof:

- $\Leftarrow$ By Lemma 4.1, $\delta_C \sqsubseteq \tau_C$ and the result follows by the transitivity of the partial order.
- $\Rightarrow$ By induction. Suppose $\delta_C \sqsubseteq \tau_A$. Since $\tau_A^0 = \delta_A^0$ and $\tau_C^0 = \delta_C^0$ it follows trivially that $\tau_C^0 \sqsubseteq \tau_A^0$. Assume now inductively that $\tau_C^{j-1} \sqsubseteq \tau_A^{j-1}$ for some $j \geq 1$. Recall that $\tau_C^j = lub(\mathbf{Y}(\tau_C^{j-1}), \delta_C^j)$, and $\tau_A^j = lub(\mathbf{Y}(\tau_A^{j-1}), \delta_A^j)$ by definition. To conclude that $\tau_C^j \sqsubseteq \tau_A^j$ we must show that $\delta_C^j \sqsubseteq \tau_A^j$ and that $\mathbf{Y}(\tau_C^{j-1}) \sqsubseteq \tau_A^j$. The first claim follows trivially from the assumption that $\delta_C \sqsubseteq \tau_A$. Since $\mathbf{Y}$ is monotonic, using the induction hypothesis yields $\mathbf{Y}(\tau_C^{j-1}) \sqsubseteq \mathbf{Y}(\tau_A^{j-1})$. Thus $\mathbf{Y}(\tau_C^{j-1}) \sqsubseteq lub(\mathbf{Y}(\tau_A^{j-1}), \delta_A^j) = \tau_A^j$ and hence $\tau_C^j \sqsubseteq \tau_A^j$ and the result follows.

□

Our next lemma deals with the monotonicity imposed by the defining trajectory construction.

Lemma 4.3 *If δ_A, τ_A, δ_C, and τ_C are the defining sequences and defining trajectories for some trajectory formulas A and C, then $\delta_C \sqsubseteq \delta_A$ implies that $\tau_C \sqsubseteq \tau_A$.*

Proof: We prove the result by induction. Suppose $\delta_C \sqsubseteq \delta_A$. Since $\tau_A^0 = \delta_A^0$ and $\tau_C^0 = \delta_C^0$ it follows trivially that $\tau_C^0 \sqsubseteq \tau_A^0$.

Assume inductively that $\tau_C^{j-1} \sqsubseteq \tau_A^{j-1}$ for some $j \geq 1$. First recall that $\tau_C^j = lub(\mathbf{Y}(\tau_C^{j-1}), \delta_C^j)$, and $\tau_A^j = lub(\mathbf{Y}(\tau_A^{j-1}), \delta_A^j)$ by definition. Now, by assumption, $\delta_C^j \sqsubseteq \delta_A^j$. Furthermore, since $\mathbf{Y}$ is monotonic, using the induction hypothesis yields $\mathbf{Y}(\tau_C^{j-1}) \sqsubseteq \mathbf{Y}(\tau_A^{j-1})$. These two facts, together with the monotonicity of lub, show that $\tau_C^j \sqsubseteq \tau_A^j$ and the result follows. □

4.2 Identity Inference Rule

Our first general theorem is almost trivial. However, it is useful in a number of places.

Theorem 4.4 *If A is any trajectory formula then $\models_{\mathcal{M}} [A \Longrightarrow\!\!\!\!> A]$.*

Proof: By Theorem 3.3 it suffices to show that $\delta_A \sqsubseteq \tau_A$. This follows directly from Lemma 4.1. $\square$

4.3 Time Shift Inference Rule

The following results allow us to move the "base time" of a result. In essence, the final result says that if we can prove an assertion with all times relative to time zero, then the same result holds with all the times relative to some time t.

Lemma 4.5 *If A and C are some arbitrary trajectory formulas then*

$$\models_{\mathcal{M}} [A \Longrightarrow\!\!\!\!> C] \quad \textit{implies} \quad \models_{\mathcal{M}} [\mathbf{N}A \Longrightarrow\!\!\!\!> \mathbf{N}C].$$

Proof: Assume $\models_{\mathcal{M}} [A \Longrightarrow\!\!\!\!> C]$ holds. By Theorem 3.3 it follows that $\delta_C \sqsubseteq \tau_A$. This, together with the definition of $\delta_{\mathbf{N}C}$, implies that $\delta_{\mathbf{N}C} = \bot\delta_C \sqsubseteq \bot\tau_A$. If we can establish that $\bot\tau_A \sqsubseteq \tau_{\mathbf{N}A}$ the claim of the theorem would follow. First, note that $\tau^0_{\mathbf{N}A} = \delta^0_{\mathbf{N}A} = \bot$. Hence, we only need to prove that $\tau^i_A \sqsubseteq \tau^{i+1}_{\mathbf{N}A}$ for $i \geq 0$. We prove this by induction on i. For the basis, $i = 0$, note that $\tau^0_A = \delta^0_A$. On the other hand, $\delta^0_A \sqsubseteq lub(\mathbf{Y}(\bot), \delta^0_A)$ by the properties of least upper bound. However, $lub(\mathbf{Y}(\bot), \delta^0_A) = \tau^1_{\mathbf{N}A}$, and thus the claim holds for the basis. Now assume the claim holds for some $i \geq 0$. First, $\tau^i_A = lub(\mathbf{Y}(\tau^{i-1}_A), \delta^i_A)$ and $\tau^{i+1}_{\mathbf{N}A} = lub(\mathbf{Y}(\tau^i_{\mathbf{N}A}), \delta^{i+1}_{\mathbf{N}A})$. However, since $\delta_{\mathbf{N}A} = \bot\delta_A$ it follows that $\delta^{i+1}_{\mathbf{N}A} = \delta^i_A$. Furthermore, by the monotonicity of $\mathbf{Y}$ and the induction hypothesis, we can conclude that $\mathbf{Y}(\tau^{i-1}_A) \sqsubseteq \mathbf{Y}(\tau^i_{\mathbf{N}A})$. These two facts, together with the monotonicity of least upper bound, implies that $\tau^i_A \sqsubseteq \tau^{i+1}_{\mathbf{N}A}$ and the induction step goes through and the lemma follows. $\square$

Theorem 4.6 *For any trajectory formulas A and C, if $\models_{\mathcal{M}} [A \Longrightarrow\!\!\!\!> C]$ then $\models_{\mathcal{M}} [N^t A \Longrightarrow\!\!\!\!> N^t C]$ for any $t \geq 0$.*

Proof: Follows trivially from Lemma 4.5 by induction on t. $\square$

As an example of using this theorem, consider the circuit shown in Fig. 5. It is easy to see that trajectory evaluation can be used to prove that $\models_{\mathcal{M}} [B \textbf{ is } 1 \Longrightarrow\!\!\!\!> \mathbf{N}(D \textbf{ is } 0)]$. Applying the above theorem allows us to deduce that if at time t B has the value 1 then at time $t+1$ D has the value 0.

Note that the converse of Theorem 4.6 is not true in general. Using the same circuit as above we can prove that $\models_{\mathcal{M}} \left[\mathbf{N}^1(D \textbf{ is } 0) \Longrightarrow\!\!\!\!> N^2(E \textbf{ is } 0)\right]$ since by time 1 node C will have the value 1. But it is not true that $\models_{\mathcal{M}} [D \textbf{ is } 0 \Longrightarrow\!\!\!\!> N(E \textbf{ is } 0)]$, since nothing is known about the value of node C at time zero.

4.4 Post-condition weakening and pre-condition strengthening

The following theorem is a direct equivalent of the classical post-condition weakening and pre-condition strengthening in Hoare logic.

Theorem 4.7 *Let A, C, A_1 and C_1 be trajectory formulas. Suppose $\models_{\mathcal{M}} [A \Longrightarrow C]$. If $\delta_A \sqsubseteq \delta_{A_1}$, then $\models_{\mathcal{M}} [A_1 \Longrightarrow C]$. If $\delta_{C_1} \sqsubseteq \delta_C$, then $\models_{\mathcal{M}} [A \Longrightarrow C_1]$.*

Proof: Suppose $\models_{\mathcal{M}} [A \Longrightarrow C]$. By Theorem 3.3 it follows that $\delta_C \sqsubseteq \tau_A$. By the assumptions in the claim, $\delta_{C_1} \sqsubseteq \delta_C$, and thus, by transitivity, $\delta_{C_1} \sqsubseteq \tau_A$. By Theorem 3.3 this implies that $\models_{\mathcal{M}} [A \Longrightarrow C_1]$ and the first part of the claim holds. On the other hand, since $\delta_A \sqsubseteq \delta_{A_1}$ it follows by Lemma 4.3 that $\tau_A \sqsubseteq \tau_{A_1}$. Thus $\delta_C \sqsubseteq \tau_A \sqsubseteq \tau_{A_1}$ and by Theorem 3.3 the second claim follows. □

Example
Consider once again the circuit shown in Fig. 5. Trajectory evaluation can easily be used to show that $\models_{\mathcal{M}} \Big[(B \textbf{ is } 0) \wedge (\mathbf{N}(B \textbf{ is } 0)) \Longrightarrow (\mathbf{N}^2(E \textbf{ is } 1) \wedge \mathbf{N}^3(E \textbf{ is } 1))\Big]$. Using post-condition weakening we can prove $\models_{\mathcal{M}} \Big[(B \textbf{ is } 0) \wedge \mathbf{N}(B \textbf{ is } 0)) \Longrightarrow \mathbf{N}^2(E \textbf{ is } 1)\Big]$.

4.5 Conjunction rule

The following theorem is particularly useful when properties about separate parts of the system have been proven and we now want to combine these results to reason about the combined part.

Theorem 4.8 *Let A_1, C_1, A_2, and C_2 be trajectory formulas. If $\models_{\mathcal{M}} [A_1 \Longrightarrow C_1]$ and $\models_{\mathcal{M}} [A_2 \Longrightarrow C_2]$, then $\models_{\mathcal{M}} [(A_1 \wedge A_2) \Longrightarrow (C_1 \wedge C_2)]$.*

Proof: First, by the assumptions and by Theorem 3.3 we have $\delta_{C_1} \sqsubseteq \tau_{A_1}$ and $\delta_{C_2} \sqsubseteq \tau_{A_2}$. This together with the definition of $\delta_{C_1 \wedge C_2}$ implies that $\delta_{C_1 \wedge C_2} = lub(\delta_{C_1}, \delta_{C_2}) \sqsubseteq lub(\tau_{A_1}, \tau_{A_2})$. By definition of δ, $\delta_{A_1} \sqsubseteq \delta_{A_1 \wedge A_2}$. Therefore, by lemma 4.1, $\tau_{A_1} \sqsubseteq \tau_{A_1 \wedge A_2}$. Similarly, $\tau_{A_2} \sqsubseteq \tau_{A_1 \wedge A_2}$. Thus $lub(\tau_{A_1}, \tau_{A_2}) \sqsubseteq \tau_{A_1 \wedge A_2}$.

Therefore, by transitivity of the partial order, $lub(\delta_{C_1}, \delta_{C_2}) \sqsubseteq \tau_{A_1 \wedge A_2}$. By Theorem 3.3 the theorem follows. □

Example
Using the circuit of Fig. 5, we can prove $\models_{\mathcal{M}} [A \textbf{ is } 0 \Longrightarrow \mathbf{N}(C \textbf{ is } 1)]$ and $\models_{\mathcal{M}} [B \textbf{ is } 0 \Longrightarrow \mathbf{N}(D \textbf{ is } 1)]$. Combining these two results using the conjunction rule gives
$\models_{\mathcal{M}} [(A \textbf{ is } 0) \wedge (B \textbf{ is } 0) \Longrightarrow (\mathbf{N}(C \textbf{ is } 1)) \wedge (\mathbf{N}(D \textbf{ is } 1))]$.

4.6 Transitivity Inference Rule

This rule is analogous to the transitivity rule in logic: if $A \Rightarrow B$ and $B \Rightarrow C$ then $A \Rightarrow C$. Suppose that we have two results $\models_{\mathcal{M}} [A_1 \Longrightarrow C_1]$, and $\models_{\mathcal{M}} [A_2 \Longrightarrow C_2]$. This rule informs us when we can conclude that $\models_{\mathcal{M}} [A_1 \Longrightarrow C_2]$. We begin with a couple of lemmas.

Lemma 4.9 *Let A_1, C_1, A_2, and C_2 be trajectory formulas. Suppose $\models_{\mathcal{M}} [A_1 \Longrightarrow C_1]$ and $\models_{\mathcal{M}} [A_2 \Longrightarrow C_2]$. If $\delta_{A_2} \sqsubseteq \delta_{C_1}$ then $\models_{\mathcal{M}} [A_1 \Longrightarrow C_2]$.*

Proof: By the assumptions in the theorem and by Theorem 3.3 it follows that $\delta_{C_2} \sqsubseteq \tau_{A_2}$ and that $\delta_{C_1} \sqsubseteq \tau_{A_1}$. Since, by assumption $\delta_{A_2} \sqsubseteq \delta_{C_1}$, it follows from Lemma 4.3 that $\tau_{A_2} \sqsubseteq \tau_{C_1}$. Furthermore, since $\delta_{C_1} \sqsubseteq \tau_{A_1}$ it follows, by Lemma 4.2, that $\tau_{C_1} \sqsubseteq \tau_{A_1}$. Hence, we have $\delta_{C_2} \sqsubseteq \tau_{A_2} \sqsubseteq \tau_{C_1} \sqsubseteq \tau_{A_1}$, and therefore by Theorem 3.3 that $\models_{\mathcal{M}} [A_1 \Longrightarrow\!\!> C_2]$. □

Theorem 4.10 *Let A_1, C_1, A_2, and C_2 be trajectory formulas. Suppose $\models_{\mathcal{M}} [A_1 \Longrightarrow\!\!> C_1]$ and $\models_{\mathcal{M}} [A_2 \Longrightarrow\!\!> C_2]$. If $\delta_{A_2} \sqsubseteq \mathrm{lub}(\delta_{A_1}, \delta_{C_1})$ then $\models_{\mathcal{M}} [A_1 \Longrightarrow\!\!> C_2]$.*

Proof:

1. By definition $\delta_{A_1 \wedge C_1} = \mathit{lub}(\delta_{A_1}, \delta_{C_1})$.
2. By Theorem 3.3, $\delta_{C_1} \sqsubseteq \tau_{A_1}$. By Lemma 4.1, $\delta_{A_1} \sqsubseteq \tau_{A_1}$. Hence $\mathit{lub}(\delta_{A_1}, \delta_{C_1}) \sqsubseteq \tau_{A_1}$. So, $\delta_{A_1 \wedge C_1} \sqsubseteq \tau_{A_1}$; consequently by Lemma 4.2. $\tau_{A_1 \wedge C_1} \sqsubseteq \tau_{A_1}$
3. By assumption $\delta_{A_2} \sqsubseteq \mathit{lub}(\delta_{A_1}, \delta_{C_1})$. Therefore, by Lemma 4.2, $\tau_{A_2} \sqsubseteq \tau_{A_1 \wedge C_1}$.
4. By Theorem 3.3, $\delta_{C_2} \sqsubseteq \tau_{A_2}$.

By transitivity of the partial order, $\delta_{C_2} \sqsubseteq \tau_{A_1}$, and the result follows by Theorem 3.3. □

Example
Consider once again the circuit shown in Fig. 5. We can use trajectory evaluation to show that $\models_{\mathcal{M}} \left[B \textbf{ is } 0 \Longrightarrow\!\!> \mathbf{N}^2(E \textbf{ is } 1)\right]$ and that $\models_{\mathcal{M}} \left[\mathbf{N}^2(E \textbf{ is } 1) \Longrightarrow\!\!> \mathbf{N}^3(F \textbf{ is } 0)\right]$. Using the transitivity rule, we can conclude, without having to compute anything else that $\models_{\mathcal{M}} \left[B \textbf{ is } 0 \Longrightarrow\!\!> \mathbf{N}^3(F \textbf{ is } 0)\right]$.

4.7 Specialisation Inference Rule

Specialisation is a rule which allows the generation from a general form of a trajectory assertion more specialised versions. For example from [**M is a** $\Longrightarrow\!\!>$ **O is 2*a**] we may wish to generate [**M is 1** $\Longrightarrow\!\!>$ **O is 2*1**] or [**M is a+b** $\Longrightarrow\!\!>$ **O is 2*(a+b)**]. There are two types of specialisation — a simple substitution, and a more general specialisation. We discuss each in turn.

It should be remembered that symbolic trajectory assertions are universally quantified by the free Boolean variables which appear in them. Usually, it is convenient to omit the quantification; however, when discussing substitutions, explicitly quantifying the assertion makes the dependencies clearer.

4.7.1 Simple substitution

Definition 4.1 *A* simple substitution *σ is a function from a set of variables, $\tilde{I}$, to expressions over constants and these variables, $\mathcal{E}_{\tilde{I}}$.* □

Theorem 4.11 (Simple Substitution Theorem) *Suppose $\forall \tilde{I}.\ \models_{\mathcal{M}} \left[A(\tilde{I}) \Longrightarrow\!\!> C(\tilde{I})\right]$, and $\sigma : \tilde{I} \longrightarrow \mathcal{E}_{\tilde{I}}$ is a simple substitution. Then $\forall \tilde{I}.\ \models_{\mathcal{M}} \left[A(\sigma(\tilde{I})) \Longrightarrow\!\!> C(\sigma(\tilde{I}))\right]$.*

Proof: To prove $\forall\tilde{I}.\ \models_{\mathcal{M}} \left[A(\sigma(\tilde{I}))\Longrightarrow\!\!\!> C(\sigma(\tilde{I}))\right]$, it suffices to show that $\models_{\mathcal{M}} [A(\sigma(\phi))\Longrightarrow\!\!\!> C(\sigma(\phi))]$ for every $\phi \in \Phi$. Consider an arbitrary assignment ϕ. Let $\phi' = \phi \circ \sigma$. Note that $\phi' \in \Phi$. On the other hand, $\forall\tilde{I}.\ \models_{\mathcal{M}} \left[A(\tilde{I})\Longrightarrow\!\!\!> C(\tilde{I})\right]$ implies that for every assignment $\phi'' \in \Phi$ we have $\models_{\mathcal{M}} [A(\phi'')\Longrightarrow\!\!\!> C(\phi'')]$. In particular, $\models_{\mathcal{M}} [A(\phi')\Longrightarrow\!\!\!> C(\phi')]$ and since ϕ is arbitrary, the claim follows. $\square$

As an example of where this might be useful, the following may have been proven about the circuit shown in Fig. 5: $\models_{\mathcal{M}} A$ **is** $a\Longrightarrow\!\!\!> \mathbf{N}^2(E$ **is** $\neg a)$ and $\models_{\mathcal{M}} \mathbf{N}^2(E$ **is** $b)\Longrightarrow\!\!\!> \mathbf{N}^3(F$ **is** $\neg b)$. Transitivity cannot be used here since it is not the case that $\delta_{\mathbf{N}^2(E \textbf{ is } b)} \sqsubseteq \delta_{\mathbf{N}^2(E \textbf{ is } \neg a)}$. However, if $\neg a$ is substituted in for b throughout the second theorem, then transitivity *can* be used.

For notational convenience, we make the following definition:

Definition 4.2 $\sigma(A(\tilde{I})) = A(\sigma(\tilde{I}))$.

Theorem 4.12 ("When" Theorem) *Let* $\models_{\mathcal{M}} [A\Longrightarrow\!\!\!> C]$ *and let* E *be any Boolean expression over some variables* $\tilde{I}$. *Then* $\forall\tilde{I}.\ \models_{\mathcal{M}} [E \rightarrow A\Longrightarrow\!\!\!> E \rightarrow C]$.

Proof: Consider any assignment $\phi \in \Phi$. When $E(\phi) = 1$, the result follows immediately. When $E(\phi) = 0$, the result follows since the defining trajectory of the consequent is the bottom sequence and thus less than everything. $\square$

4.7.2 Specialisation

Substitution by itself is too coarse a transformation method. Using our example circuit, let $T_1 = [B$ **is** $a\Longrightarrow\!\!\!> \mathbf{N}^2((a \rightarrow (E$ **is** $0)) \wedge (\neg a) \rightarrow (E$ **is** $1))]$ and $T_2 = [\mathbf{N}^2(E$ **is** $b)\Longrightarrow\!\!\!> \mathbf{N}^3(F$ **is** $\neg b)]$. We can prove using Voss that $\models_{\mathcal{M}} T_1$ and $\models_{\mathcal{M}} T_2$. Suppose we want to "modify" T_2 so that we can use transitivity between T_1 and T_2. Any single substitution here into T_2 will lose us information so that the substituted version of T_2 will not be of help.

A *specialisation* gives more fine-grained control over the modification of a trajectory assertion.

Definition 4.3 *Let* $\hat{\sigma} = [(g_1, \sigma_1), \ldots, (g_n, \sigma_n)]$, *where each* g_i *a Boolean expression, and each* σ_i *is a substitution. If* C *is a trajectory formula then define* $\hat{\sigma}(C) = \wedge_1^n(g_i \rightarrow \sigma_i(C))$.

A specialisation is a list of pairs, each pair consisting of a boolean expression (the guard) and a simple substitution. To use the specialisation, for each guard-substitution pair, we use the substitution, and then qualify the antecedent and consequent of the trajectory assertion by restricting the domain of all variables to when the guard is true. To obtain the specialised assertion, all such qualified substitutions are conjoined. In this example, the specialisation $[(a, [(b, 0)]), (\neg a, [(b, 1)])]$ could be used to specialise T_2, and we would obtain the convenient result

$$T_2' = \mathbf{N}^2((a \rightarrow (E \textbf{ is } 0)) \wedge ((\neg a) \rightarrow (E \textbf{ is } 1)))\Longrightarrow\!\!\!> (\mathbf{N}^3((a \rightarrow (F \textbf{ is } 1)), ((\neg a) \rightarrow (F \textbf{ is } 0)))).$$

By the theorem below, we will have $\models_{\mathcal{M}} T_2'$, and now will be in a position to use transitivity between T_1 and T_2'.

Theorem 4.13 (Specialisation Theorem) *Let $\hat{\sigma} = [(g_1, \sigma_1), \ldots, (g_n, \sigma_n)]$ be a specialisation, where each g_i is a Boolean expression, and each σ_i is a simple substitution. If $\models_{\mathcal{M}} [A \Longrightarrow\!\!\!> C]$ then*

$$\models_{\mathcal{M}} [\wedge_i(g_i \rightarrow \sigma_i(A)) \Longrightarrow\!\!\!> \wedge_i (g_i \rightarrow \sigma_i(C))].$$

Proof: $\models_{\mathcal{M}} [A \Longrightarrow\!\!\!> C]$ implies that $\models_{\mathcal{M}} [\sigma_i(A) \Longrightarrow\!\!\!> \sigma_i(C)]$ by the simple substitution theorem. Therefore, $\models_{\mathcal{M}} [g_i \rightarrow \sigma_i(A) \Longrightarrow\!\!\!> g_i \rightarrow \sigma_i(C)]$ (when lemma), and therefore, $\models_{\mathcal{M}} [\wedge_i(g_i \rightarrow \sigma_i(A)) \Longrightarrow\!\!\!> \wedge_i (g_i \rightarrow \sigma_i(C))]$ (repeated use of the conjunction theorem). $\square$

4.8 Overview

In this section, we have completed the proof of the inference rules which our theorem prover described in section 6 uses. The basic rules of inference are:

1. VOSS: Given a trajectory assertion, use symbolic trajectory evaluation to show that it is valid.
2. Time-shifting
3. Transitivity
4. Post-condition weakening
5. Pre-condition strengthening
6. Specialisation
7. Conjunction

The soundness of rule 1 was shown in section 3, and the soundness of the remaining rules were shown in this section.

For a practical tool, we need something more. For example, we may have $\models_{\mathcal{M}} [A \Longrightarrow\!\!\!> B]$ and $\models_{\mathcal{M}} [C \Longrightarrow\!\!\!> D]$. To derive a useful result, we may need to time-shift the former, specialise the latter, and use transitivity. While being able to manually specify the specialisations and time-shifting is important, for a useful tool we would like as much of this to be done automatically. Although complete automation is not possible, we have made progress in this regard. The theorem prover therefore provides additional rules of inference which use the basic ones as steps. The soundness of these compound rules follows from the soundness of the basic rules.

5 Boolean and other domains

While in principle the theory of symbolic trajectory evaluation can be extended to any domain, in practice symbolic trajectory evaluation is applied to more limited domains. In the Voss system, the underlying domain is a quaternary system $\{0, 1, \top, \bot\}$, and the lattice for the state space is an appropriate cross-product of this lattice.

This can make specifying the behaviour of systems tedious and error-prone: it becomes difficult to see that the formulas written down accurately represent our intuitive understanding of the system. For example, to verify a circuit which adds two numbers together, the specification must be given in terms of the signals which different bits in the circuit have, rather than in terms of integers being added together.

Further, BDDs are the sole means of checking properties. In practice, this is a significant problem. In proving properties, it is not uncommon to need domain-specific properties such as the commutativity of integer multiplication (i.e. is $xy = yx$, where x and y are integers?). This example is a simple result, but cannot be checked using BDDs due to the inherent limitations of BDDs.

Therefore there are two reasons to raise the level of abstraction: to make the specification of properties easier and more understandable; and to use domain-specific knowledge in proofs. Two possible ways of raising the level of abstraction are presented below.

5.1 Node/value mapping

The solution we have adopted is to provide library routines which simplify the process of specification. Specification is still at the bit-level, but the library routines provide a higher-level interface to do this. For example instead of saying:

```
(N7  IS a7)  & (N6 IS a6) & (N5 IS a5) & (N4 IS a4) &
(N3  IS a3)  & (N2 IS a2) & (N1 IS a1) & (N0 IS a0)
```

we can write

```
N ISINT a
```

There are two issues. First, there is a need to refer to elements of the circuit being verified. Instead of referring individually to members of a group of nodes, we may find it convenient to name them as a group, and thus view the system at a more abstract level. Or, it may be appropriate to refer to nodes with more meaningful names. Thus, the first part of mapping is to map a logical node name to a physical node or a vector of physical nodes. The user of the system must do this by providing the mapping from logical to physical names. This part of the mapping is purely syntactic sugar, and it is important for the verifier to realise that the verification really is in terms of the physical nodes, not the logical nodes.

The drawback of this syntactic mapping is that the user of the system must clearly understand what the physical level of the system looks like, *and* how the translation works. The user must give the mapping from each integer node to the b boolean nodes which represent it, and so on. The library routines do not hide the complexity of the underlying circuit, rather they provide a way in which the complexity can be managed. Whatever the appearance that the library routines and pretty-printing gives, it needs to be emphasised that all specifications and all symbolic trajectory evaluation results are in terms of the state of the physical nodes.

The second issue is being able to refer to values in terms of concepts such as Booleans and integers, rather than as bits or bit-vectors. To do this, we need a proper semantic mapping between the values we represent at the "higher" level, and their meaning given by bits or bit-vectors. This semantics is given by a mapping routine. Using this mapping, it is possible to specify values in a more abstract way, and incorporate domain-knowledge into the system in a flexible yet rigorous way.

In the HOL-Voss system (Joyce & Seger, 1993) it was shown how the HOL theorem prover could be linked to the Voss system in a rigorous way. There, a set of library routines was formally proven to be correct. For example, we can be sure that $bv(x+y) = bv(x)\,\texttt{add}\,bv(y)$, where bv is the library routine which translates from integers into bit-vectors, and `add` is addition defined on bit-vectors. Knowing this (and similar properties of bv) allows us to use properties of integers (proven in HOL) without having to construct the bit-vectors for the expressions. The proof of the correctness of bv assures us that given an equation proven true in HOL, were we to construct BDDs for both sides of the equation, the two BDDs would be the same.

Similarly, in the present system we have a mapping function which gives the semantics of integer and boolean expressions in terms of bits and bit-vectors (although we have not gone through the exercise of performing a mechanically checked correctness proof as was done in HOL-Voss). Domain knowledge is provided by a simple integer theorem-prover and decision procedures for integers. This results in a sound, practical method for expressing integer and boolean values, and using domain knowledge effectively. We explore the strengths and weaknesses of this approach in the conclusion.

The following example illustrates the advantage of this approach

```
N ISINT a*b;
```

- Instead of referring to a number of bit-level nodes individually, we can refer to them as a group.
- Multiplication is not possible to represent efficiently using BDDs. Here we have a practical way of doing the representation which is sound in principle.
- We can use domain knowledge (for example properties of natural numbers) in our proofs. Examples will be seen later.

5.2 Mapping between domains

The second approach to raising the level of abstraction is to allow mapping between domains. Specification is done in one domain, and verification in another. Abstraction occurs when details present in the one domain are not present in the other. Abstraction is an important tool in making verification more efficient, and there have been a number of proposals for doing so (Donat, 1993; Long, 1993; McIsaac, 1993). There is an important distinction: we are proposing abstraction as a way to make verification pleasanter for the verifier, rather than more efficient.

In terms of the work we present in this paper there is no clear advantage in performing domain mapping over performing the simpler mapping. There are a number of open issues in how this form of abstraction can be specified, used, and automated. In general, we believe that besides performance benefits discussed elsewhere (Long, 1993; McIsaac, 1993), this may lead to more secure systems since the proofs will be done in domains closer to users' intuitive understanding of systems.

6 Practical Tool

The theory described in the preceding two sections has been implemented, and integrated with Voss into a new system. This tool is a theorem prover with which a user can prove properties of circuits specified either in VHDL or netlist form. The verification proof is an FL program which uses the theorem prover at each step. The ability to use FL as a script language gives great versatility and power.

In this section we describe the inference rules which the theorem prover provides. There are a number of small examples. More extensive examples are given in the next section.

We use the term "trajectory assertion" to denote any claim made about the circuit, and the term "theorem" to denote a claim which has been proven valid by using the rules of inference.

6.1 Use of tool

FL is used as the script language for proofs. The system has defined abstract data types for representing and manipulating trajectories and the appropriate values (like integers).

Trajectory assertions are expressed as an abstract data type using the syntax below.

```
Trajectory ::=
      string IS Value |
      Trajectory WHEN BoolValue |
      Trajectory FROM IntValue TO IntValue |
      Trajectory _&_ Trajectory

Value     ::= INT IntValue | BOOL BoolValue
```

Note that the expression `v WHEN c` is used to represent $c \rightarrow x$. The shorthand `Node ISINT x` can be used in place of `Node IS (INT x)`, and the shorthand `Node ISBOOL x` can be used in place of `Node IS (BOOL x)`. Timing is expressed using `FROM` and `TO`. Saying T `FROM s TO f` is the same thing as saying: $\mathbf{N}^s T \wedge \mathbf{N}^{s+1} T \ldots \wedge \mathbf{N}^{f-1} T$.

The two data types supported are integer and boolean expressions:

```
 int_expr = ' int   | IVar  string | '+  int_expr int_expr       |
           '- int_expr  int_expr | '* int_expr  int_expr       |
           DIV2   int_expr int_expr  | BWID int_expr  int_expr |
           BIT2   int_expr  int_expr | POW2   int_expr;

 bool_expr=
           Var string | CTrue | CFalse |
           And bool_expr bool_expr  | Or  bool_expr bool_expr  |
           Xor bool_expr bool_expr  | Not bool_expr            |
           '>  int_expr int_expr    | '=  int_expr int_expr    |
           '<  int_expr int_expr    |  --> bool_expr bool_expr;
```

`DIV2 x n` represents $x/2^n$. `BWID n x` represents the n lower order bits of x, *viz.* $x \bmod 2^n$. `POW2 n` is 2^n. `BIT2 n x` is the n-th bit of x, *viz.* $(x \bmod 2^n)/2^{n-1}$.

Trajectory assertions are of the form:

```
AssertionPair ::= Trajectory ==>> Trajectory;
```

Using this syntax, we can represent trajectories and trajectory assertions. The trajectory assertions can be verified and generated using only using the inference rules described below. This style of system is copied from the later LCF systems, where ML was the meta-language for interacting with the system (Paulson, 1987). In HOL (which grew out of LCF, and uses ML too), only HOL's inference rules can be used to generate theorems, but the ML program is used to order and combine the proof steps (Gordon, 1987; Gordon, 1993).

The inference rules return objects of type `Theorem`. Besides the inference rules, only `PrThm` (pretty printing) can manipulate these objects. This limitation guarantees the soundness of the results; by restricting the operations allowed on this type of object, we can ensure that the system is safe, while still having the power and flexibility of a fully-programmable script language.

Abstraction is promoted by allowing the specification to be given either in terms of integers or Booleans, and allowing the user to specify a correspondence between logical and physical nodes. To incorporate domain knowledge, the user can provide a *theory* which is a list of properties about integers. Each property is given as a pair of expressions, such a pair stating that the two expressions are equal. Properties can be proven using a special-purpose integer theorem-prover built for this tool. Domain knowledge is also supported through a set of simple normalisation routines which try to keep integer and boolean expressions in a close to canonical form. The system can use the domain information in two ways. Where the normalisation is successfully done, comparison two expressions can be done by simply performing a syntactic check between the two expressions. Where that fails each property is used in turn by matching the pair of expressions in the property with the pair of expressions we wish to compare. If these matchings succeed then a positive answer can be given, otherwise the comparison fails.

An issue beyond the scope of the paper is the comparison (in the information ordering) of trajectories. Since BDDs are no longer used to represent trajectories, other algorithms were developed to compare trajectories.

6.2 Primitive inference rules

The primitive inference rules are implementations of the inference rules described in Section 3. The implementation of these rules, together with basic routines such as trajectory comparison, form the core of the system, the trusted part of the system.

6.2.1 The Identity rule

Form of rule: `IDENTITY trajectory`
This rule takes a trajectory, T and returns the theorem $[T \Longrightarrow\!\!\!> T]$.

6.2.2 The Voss inference rule

Form of rule: `VOSS varmap (Ant, Con)`
This rule takes a trajectory assertion (an antecedent, consequent pair), and verifies this assertion by using Voss to check that the circuit is consistent with this assertion. The antecedent and consequent must be translated into Voss format. This implies that integer expressions must be encoded as bit vectors. `varmap` specifies the relationship between the domain variables in the antecedent and consequent and the underlying bit-vector variables used in the encoding. This mapping is needed because the performance of BDD-based approaches depends on the variable ordering used. The

mapping is checked automatically for consistency, so the correctness of the rule is not affected if a wrong ordering is given, although the performance may suffer.[4] One of the big advantages of breaking down the proof into different parts is that the appropriate variable orderings can be used for each part. Where the entire circuit is verified as a whole, it is sometimes difficult or impossible to find a variable ordering which allows efficient verification.

6.2.3 Time shifting

Form of rule: `SHIFT Base t`
This takes a valid theorem, and a non-negative integer t, and derives a new rule using the time shift inference rule.

For example to derive the assertion: [N ISINT a FROM 10 TO 20 $\Longrightarrow\!\!\!\!\gg$M ISINT 2*a FROM 20 TO 30], the following code could be used.

```
let A1 = N ISINT a FROM 0 TO 10 in
let C1 = N ISINT ('2 '* a) FROM 10 TO 20 in
  let thm1 = VOSS varmap (A1, C1) in
    SHIFT  thm1 10;
```

6.2.4 Conjuncting two valid trajectory assertions

Form of rule: `CONJUNCT Theorem1 Theorem2`
This rule takes two valid theorems $[A_1 \Longrightarrow\!\!\!\!\gg C_1]$ and $[A_2 \Longrightarrow\!\!\!\!\gg C_2]$ and returns a new valid trajectory assertion $[A_1 \wedge A_2 \Longrightarrow\!\!\!\!\gg C_1 \wedge C_2]$.

6.2.5 Precondition strengthening and Postcondition weakening

Form of rules: `PRESTRONG theory Theorem NewAnt`, `POSTWEAK theory Theorem NewCon`
`PRECONDITION` takes a theory (which may be empty) a basis theorem $[A \Longrightarrow\!\!\!\!\gg C]$ and a new antecedent A'. It then checks whether $\delta_A \sqsubseteq \delta_{A'}$, and if so returns a new theorem $[A' \Longrightarrow\!\!\!\!\gg C]$, or otherwise fails. To answer the question of whether $\delta_A \sqsubseteq \delta_{A'}$, the theory may be used to do testing.

`POSTCONDITION` is an analogous rule for postcondition weakening.

6.2.6 Transitivity

Form of rule: `TRANS theory Thm1 Thm2)`
Transitivity is one of the most important rules and is often used, either directly or indirectly. `TRANS` takes a theory, and two valid theorems $[A_1 \Longrightarrow\!\!\!\!\gg C_1]$ and $[A_2 \Longrightarrow\!\!\!\!\gg C_2]$. It checks (possibly using the theory) whether $\delta_{A_2} \sqsubseteq lub(\delta_{C_1}, \delta_{A_1})$, and if so returns the theorem $[A_1 \Longrightarrow\!\!\!\!\gg C_2]$; if not, it fails.

6.2.7 Specialisation

Form of rule: `SPECIAL Theorem subfn`
This takes a theorem and a specialisation, and returns the theorem, specialised.

[4] We provide library routines for giving the variable ordering. In this paper, we do not discuss or show this any further as this is a really technical detail.

A specialisation is a list of qualified substitutions. A qualified substitution is a pair consisting of a boolean expression which specifies when the substitution is valid, and a simple substitution. A simple substitution is pair consisting of a substitution list for Booleans and a substitution list for integers. A substitution list is a list of variable name, expression pairs which shows for each variable what expression should be substituted for it.

As an example of where specialisation may be useful, consider a system comprising a selector and an inverter. We can prove

```
T1 = (N1 ISINT a) _&_ (N2 ISINT b)
       ==>>
         (P ISINT a WHEN (a '> b)) _&_  (P ISINT b WHEN (Not(a '> b)));
```

about the selector, and we can prove

```
T2 = (P ISINT d) ==>>  (Q ISINT (Not d))
```

about the inverter. We would like to put these two results together, but cannot in the present form. Simple substitution of a for d in $T2$ would not help since the antecedent of the new theorem still would not be less (in the information ordering) than the consequent of $T1$. What we want to do is to substitute a for d when $a > b$, and substitute b for d otherwise. In the notation developed here, the appropriate specialisation is $[(a/d, a > b), (b/d, \neg a > b)]$. For each specialisation, a substitution is done, and a qualification is made. So in this case we would get two theorems from $T2$:

```
T2_1 = (P ISINT a WHEN (a '> b)) ==>> (Q ISINT (NOT a) WHEN (a '> b));
T2_2 = (P ISINT b WHEN (NOT(a '> b)))
                          ==>>  (Q ISINT (NOT b) WHEN (NOT(a '> b)));
```

Then using conjunction, we get

```
(P ISINT a WHEN (a '> b)) _&_ (P ISINT b WHEN (NOT(a '> b)))
   ==>>
    ((Q ISINT a WHEN (a '> b)) _&_  (Q ISINT b WHEN (Not(a '> b))));
```

which is what we want[5] since the antecedent of this is related to the consequent of $T1$. Using transitivity results in

```
T1 = (N1 ISINT a) _&_ (N2 ISINT b)
       ==>>
        ((Q ISINT a WHEN (a '> b)) _&_  (Q ISINT b WHEN (Not(a '> b))));
```

6.2.8 Automatic Specialisation rule

Form of rule: `AUTOSPTRANS goal theoremList`
This rule is an implementation of the combination of the simple substitution theorem (Theorem 4.11) and the transitivity theorem (Theorem 4.10). It takes a goal trajectory assertion with antecedent $A(V)$ and consequent $C(V)$, and a list of proven theorems $\models_{\mathcal{M}} [A_1(V) \Longrightarrow\!\!\!\!> C_1(V)], \ldots, \models_{\mathcal{M}} [A_n(V) \Longrightarrow\!\!\!\!> C_n(V)]$, and attempts to prove $\models_{\mathcal{M}} [A(V) \Longrightarrow\!\!\!\!> C(V)]$.

[5] As can be seen, specialisation is not primitive in that it could be obtained by implementing a substitution rule and the "when theorem" and using these two new rules and the conjunction rule. It is convenient, however, to implement it this way.

If $\forall V.\delta_{A_1(\sigma_1(V))} \sqsubseteq \delta_{A(V)}$, then, by precondition strengthening, $\forall V. \models_{\mathcal{M}} [A(V) \Longrightarrow\!\!\!> C_1(\sigma_1(V))]$. Further, if $\forall V.\delta_{A_2(\sigma_2(V))} \sqsubseteq lub(\delta_{A_1(\sigma_1(V))}, \delta_{C_1(\sigma_1(V))})$, then by the transitivity rule, we have $\forall V. \models_{\mathcal{M}} [A(V) \Longrightarrow\!\!\!> C_2(\sigma_2(V))]$.

If we show for each $i = 3, \ldots, n$ that $\forall V.\delta_{A_{i+1}(\sigma_{i+1}(V))} \sqsubseteq lub(\delta_{C_i(\sigma_i(V))}, \delta_{A_i(\sigma_i(V))})$, then we shall have using transitivity that $\forall V. \models_{\mathcal{M}} [A(V) \Longrightarrow\!\!\!> C_n(\sigma_n(V))]$. Finally using post-condition weakening, if $\forall V.\delta_{C(V)} \sqsubseteq \delta_{C_n(\sigma_n(V))}$, then we shall have $\forall V. \models_{\mathcal{M}} [A(V) \Longrightarrow\!\!\!> C(V)]$. Therefore, to conclude this, we have to find substitutions $\sigma_1, \ldots, \sigma_n$ for which the specified relationships hold.

Putting this all together formally we need to test whether, B holds, where

$$\begin{aligned} B = \exists \sigma_1, \ldots, \sigma_n . \forall V[& \delta_{A_1(\sigma_1(V))} \sqsubseteq \delta_{A(V)} \text{ AND} \\ & \delta_{A_{i+1}(\sigma_{i+1}(V))} \sqsubseteq \delta_{C_i(\sigma_i(V)) \wedge A_i(\sigma_i(V))}, \ \forall i \in \{1, \ldots, n\} \text{ AND} \\ & \delta_{C(V)} \sqsubseteq \delta_{C_n(\sigma_n(V))}] \end{aligned}$$

Now, suppose there exists a $\widetilde{\sigma_1}$, such that $\delta_{A(V)} \sqsubseteq \delta_{A_1(\widetilde{\sigma_1})}$, then if we define $\sigma_1(V) = \widetilde{\sigma_1}$, we shall have that $\delta_{A(V)} \sqsubseteq \delta_{A_1(\sigma_1(V))}$. We shall get similar results if we define the other σ_i correspondingly. The converse of this holds too. Therefore, if we define B' by

$$\begin{aligned} B' = \forall V. \exists \widetilde{\sigma_1}, \ldots, \widetilde{\sigma_n} . [& \delta_{A_1(\widetilde{\sigma_1})} \sqsubseteq \delta_{A(V)} \text{ AND} \\ & \delta_{A_{i+1}(\widetilde{\sigma_{i+1}})} \sqsubseteq \delta_{C_i(\widetilde{\sigma_i}) \wedge A_i(\widetilde{\sigma_i})}, \ \forall i \in \{1, \ldots, n\} \text{ AND} \\ & \delta_{C(V)} \sqsubseteq \delta_{C_n(\widetilde{\sigma_n})}] \end{aligned}$$

then $B \iff B'$.

B' can be checked automatically using BDDs. It can be encoded as an FL fragment using the existential quantifiers. Executing this fragment then determines whether the substitutions exist or not. The existence of the substitutions implies that $[A \Longrightarrow\!\!\!> C]$ is a valid theorem. This does not tell us what the substitutions are, but we do not have to know what the substitutions are to know whether the result is valid. This is a pleasing result in that it shows the power of BDDs. It is also a powerful result, since it is not limited by the same constraints as the SPTRANS rule below is. However, it is likely to be very expensive in practice, particularly for non-boolean domains.

6.3 Additional inference rules

The rules described so far are all that are strictly necessary to implement the rules described in chapter 4. However, to be really useful, they should be packaged in some way. Suppose we have $\models_{\mathcal{M}} [A_1 \Longrightarrow\!\!\!> C_1]$ and $\models_{\mathcal{M}} [A_2 \Longrightarrow\!\!\!> C_2]$. Now it may be that before being able to use transitivity to deduce that $\models_{\mathcal{M}} [A_1 \Longrightarrow\!\!\!> C_2]$ that either the first or the second trajectory assertion needs to be time-shifted. Or, the second trajectory assertion may need to be specialised. The rules described before allow the user to specify the necessary time-shifting and/or specialisation. However, it would be better for the system to be able to derive these changes automatically. This is what the rules below do. These rules are not core rules in that the soundness of the results does not rely on their correct implementation. All these rules use the primitive rules to perform inference. Their utility is that they are packaged with heuristics which "guess" appropriate specialisations etc. If these heuristics are incorrect, or not powerful enough, it may not be possible to prove results which are true; it will not be possible to prove false results.

6.3.1 Specialise/Transitivity rule

Form of rule: `SPTRANS theory Theorem1 Theorem2`
This rule takes a theory, and two theorems $\models_{\mathcal{M}} [A_1 \Longrightarrow\!\!\!> C_1]$ and $\models_{\mathcal{M}} [A_2 \Longrightarrow\!\!\!> C_2]$. It then tries to derive a specialisation $\hat{\sigma}$ such that $\hat{\sigma}(A_2) \sqsubseteq lub(\delta_{A_1}, \delta_{C_1})$, and if successful it returns $\models_{\mathcal{M}} [A_1 \Longrightarrow\!\!\!> \hat{\sigma}(C_2)]$. Unlike AUTOSPTRANS, SPTRANS is not a direct implementation of the theory from Section 4; rather it uses TRANS and SPEC for inferences.

A description of the algorithm used to derive the specialisation is beyond the scope of the paper, but essentially, it matches the antecedent of the second theorem against the consequent of the second. The process is analogous to unification. The example given in Section 6.2.7 illustrates why we wish to do this. This rule is the combination of the TRANS and SPEC rules, packaged with a heuristic to find the appropriate specialisation.

A simple example is given here, and a more complicated one in the next section. Suppose the first theorem is `[A ==>> (N ISINT c WHEN d)]` and the second theorem is `[(N ISINT x) ==>> C]`. What is the transformation which will make the antecedent of the transformed second theorem less than the consequent of the first? In this case, $\hat{\sigma} = [([x \longrightarrow a+b], e), ([x \longrightarrow c], d)]$ is the appropriate transformation, since it is trivially true that `(N ISINT (a+b) WHEN e) _&_ (N ISINT c WHEN d)` is less than itself. Hence, this rule would derive as a theorem $\models_{\mathcal{M}} [A \Longrightarrow\!\!\!> \hat{\sigma}(C)]$.

6.3.2 Automatic time alignment

Form of rule: `AUTOTIME theory Theorem1 Theorem2`
This rule takes a theory, and two valid theorems $\models_{\mathcal{M}} [A_1 \Longrightarrow\!\!\!> C_1]$ and $\models_{\mathcal{M}} [A_2 \Longrightarrow\!\!\!> C_2]$. Using the theory, the rule determines whether there exists a suitable time shift for the second theorem so that transitivity can be used between the two theorems. Recall that theorems can only be shifted forward in time. So, if the algorithm returns a negative time-shift, we cannot shift the second theorem backwards in time; rather we must shift the first theorem forwards in time. Formally, the rule seeks an integer t such that $\delta_{\mathbf{N}^t A_2} \sqsubseteq \delta_{C_1}$. If such a t exists, then $\models_{\mathcal{M}} [A_1 \Longrightarrow\!\!\!> \mathbf{N}^t C_2]$ (or $\models_{\mathcal{M}} [\mathbf{N}^{-t} A_1 \Longrightarrow\!\!\!> C_2]$ if t is negative) is returned as a valid theorem. Note that the remark after Theorem 4.6 explains why we have to treat the cases of t being positive and negative as separate cases: while forward-shifting a theorem yields a valid theorem, backward shifting does not necessarily do so. A description of the algorithm that finds the time-shift is beyond the scope of this paper.

This rule will find a time-shift if it exists, subject to the limitations of the domain knowledge given in the theory.

6.3.3 Combined time-aligning and specialising

Form of rule: `ALIGNSUB theory Theorem1 Theorem2`

This rule takes a theory, and two theorems, $\models_{\mathcal{M}} [A_1 \Longrightarrow\!\!\!> C_1]$ and $\models_{\mathcal{M}} [A_2 \Longrightarrow\!\!\!> C_2]$. The rule then attempts to find a time shift t and a specialisation $\hat{\sigma}$ such that $\models_{\mathcal{M}} [A_1 \Longrightarrow\!\!\!> \hat{\sigma}(\mathbf{N}^t C_2)]$, if $t \geq 0$, or $\models_{\mathcal{M}} [\mathbf{N}^{-t} A_1 \Longrightarrow\!\!\!> \hat{\sigma}(C_2)]$, if $t < 0$.

In general, for a given pair of theorems there could be a number of different time-shifts and specialisations which could be used to combine the theorems. To enumerate all the possibilities would be too computationally expensive, therefore a simple approach is taken. The first step is to guess a time-shift which might be suitable. This is done based on examining when nodes are defined, but ignoring how they are defined. Depending on whether the time-shift is negative or positive, the appropriate theorem is time-shifted. Although we cannot be sure that the time-shifted theorem will be useful in the next step, the time-shifted theorem is valid. The next step is to apply the SPTRANS rule to try to find an appropriate specialisation. This implies that the ALIGNSUB rule is safe, since if the SPTRANS rule finds an appropriate specialisation, a valid theorem will be returned, and if it cannot, no theorem will returned.

7 Examples

The first three examples are contrived examples to show the use of some of the inference rules. The fourth example shows the complete verification of a 64-bit multiplier.

7.1 Simple example 1

The first example illustrates the use of the SPTRANS rule, allowing the two trajectory assertions to be composed by appropriately specialising the second assertion, and then using transitivity. For

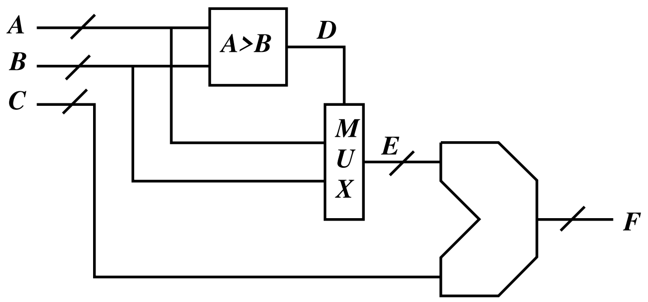

Figure 6: Circuit to illustrate use of the `SPTRANS` rule.

the first example, consider the circuit shown in Fig. 6. The overall purpose of the circuit is to take in three numbers x, y and z on nodes A, B and C, and produce $z + max(x, y)$ on F. There are three parts to the circuit. The first part compares the value on A with the value on B and produces true on D if the number at A is bigger than the number on B and produces false otherwise. The second part of the circuit takes the values at A, B and D and produces at node E the value at A if D is set to true, and produces B otherwise. The third part of the circuit takes the values at node E and C, adds them together, and produces the sum at node F. The proof script is below.

The theorem $T1$ is the proof that the comparator part of the circuit works. The theorem $T2$ is the proof that the selector part of the circuit works. The theorem $T3$ is the proof that the adder part of the circuit works. The proof of the correctness of the entire circuit contains two steps. First we use SPTRANS using $T1$ and $T2$. This results in specialising $T2$ (substitute $i \geq j$ for a) and then using transitivity to generate $G1$. SPTRANS is then used again, combining $G1$ and $T2$. The specialisation used on $T3$ is $[(([(''k'', i)], i > j), ([(''k'', j)], Not(i > j))]$. This shows quite nicely the power of general specialisation. Essentially it allows the proof of two cases to be done in one step.

```
// Information about BDD variable ordering omitted
let list1   = ....
let varmap1 = ....
let varmap2 = ....

let GlobalInput = ((A ISINT i) _&_ (B ISINT j) _&_ (C ISINT k)) FROM 0 TO 100;

let A1 = GlobalInput;
let C1 = D ISBOOL (i '> j) FROM 10 TO 100;
let T1 = VOSS varmap1  (A1 ==>> C1);

let A2 = GlobalInput _&_ ((D ISBOOL a) FROM 10 TO 100);
let C2 = GlobalInput _&_
         ((E ISINT i WHEN a) _&_ (E ISINT j WHEN (Not a)) FROM 20 TO 100);
let T2 = VOSS varmap2  (A2 ==>> C2);

let A3 = E ISINT l  _&_ C ISINT k   FROM 20 TO 100;
let C3 = FNode ISINT (l '+ k) FROM 50 TO 100;
let T3 = VOSS varmap1  (A3 ==>> C3);


let proof =
     let G1 = SPTRANS [] T1 T2 in
     let G2 = SPTRANS [] G1 T3 in
       G2;
```

Note that the verification of each component is done in the presence of the rest of the system, which means that any unintended interference can be detected. By considering the state of the rest of the system as unknown (i.e. having the nodes set to X) we can perform the component verification efficiently and still be sure that no matter what the rest of the system does we shall obtain the desired behaviour. This is a very useful property, since much of the work involved in other compositional approaches goes into ensuring that each component works correctly in all environments.

7.2 Simple example 2 – hidden weighted bit

The hidden weighted bit problem was one of the first to be proven to need exponential space to verify using traditional BDD-based methods (Bryant, 1991). A circuit for an 8-bit version is shown in Figure 7.

In this version, the global input $x_1, \ldots, x_n$ is copied to two buffers. The *Counter* part of the circuit computes the number of 1's on the input (i.e. $\Sigma_1^n x_i$). The *Chooser* part of the circuit takes the number j output on *CountNode* (the number is in binary form, hence if there are n input lines, *CountNode* comprises $\lfloor \lg n \rfloor + 1$ lines), and outputs the value x_j on *Result* and 0 on *Error* when $j > 0$. If $j = 0$ then *Error* is set to 1.

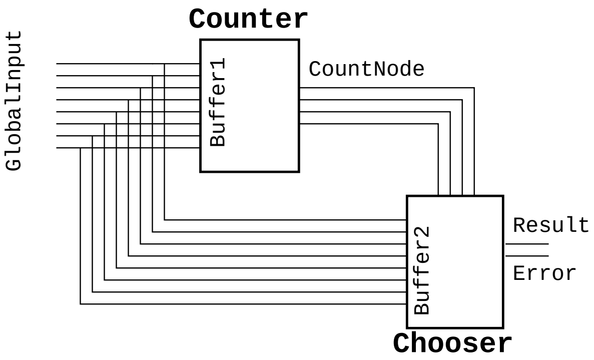

Figure 7: Circuit for the 8-bit hidden weighted bit problem

Intuitively, a verification of this circuit *as a whole* takes exponential time and space (in n) because the output value on *CountNode*, in terms of the boolean variables, is so complicated that no suitable variable ordering can be found so that the *Chooser* part of the circuit can be verified efficiently. The virtue of the the compositional approach is clearly illustrated: by decoupling the verification of the two parts of the circuit, we can choose suitable variable orderings for both parts of the circuit; moreover, it is more efficient to verify the circuit for an arbitrary input j (which only needs very simple BDDs to represent it), and then substitute for j the actual input, than to verify for the actual input (which needs more complicated BDDs to represent it).

The proof script is given below. There are five steps in the proof.

- The proof the copying of the input to the buffer is correct — *BufferTheorem*.
- The verification *Counter* part of the circuit — *CounterTheorem*.
- The composition of *BufferTheorem* and *CounterTheorem*. This is done in two steps: first, the AUTOTIME rule is used to shift the *CounterTheorem* along so that transitivity between *BufferTheorem* and *CounterTheorem* can be used; second, conjunct this with *BufferTheorem* so that we can use the value of *Buffer2* at a later stage. Call the result of this *stage1*.
- Verification of the *Chooser* part of the circuitry — *ChooserTheorem*.
- Composition of *stage1* and *ChooserTheorem* by using the ALIGNSUB rule. This shifts *ChooserTheorem* by an appropriate amount and specialises this so that transitivity can be used between *stage1* and *ChooserTheorem*.

```
//BDD variable ordering information omitted...
let varmap1 = ....
// Stage (1) Does the buffer pat of the circuit work
let  BufferTheorem =  VOSS varmap1
     ((InputNode ISINT x FROM 0 TO 1000)
      ==>>  ((BufferNode1 ISINT x) _&_ (BufferNode2 ISINT x) FROM 5 TO 1000));

// Stage (2) Does the counter part of the circuit work
```

```
let  count_of num =
     letrec add_bits x num =
                x = N => BIT2 ('N) num
                       | (BIT2 ('x) num) '+ (add_bits (x+1) num)
     in
         add_bits 1 num;


let CounterGoal = (BufferNode1  ISINT x FROM 0 TO 990) ==>>
                  (CountNode    ISINT (count_of x) FROM 400 TO 990);
let CounterTheorem = VOSS varmap1 CounterGoal;

// Stage (3)
let stage1  = CONJUNCT BufferTheorem
                       (AUTOTIME [] BufferTheorem CounterTheorem);

// Stage (4) Prove the chooser part
let seg x  =  BWID ('Nbit) x;
let kthBit k var = (BIT2 ('k) var) '= (BIT2 ('1) ('1));

letrec case_analysis var j =
       letrec case k =
         k=1 =>  Result ISBOOL  (kthBit k var) WHEN (j '= (seg ('k)))
              | (Result ISBOOL  (kthBit k var) WHEN (j '= (seg ('k))))
                _&_ (case (k-1) )
       in
           case N;

let  ChooserGoal=
     ((CountNode   ISINT j) _&_ (BufferNode2 ISINT x) FROM 0 TO 400)
          ==>>
        (((case_analysis x j) _&_ (Error ISBOOL (seg('0) '= j)))
                                                   FROM 300 TO 400);
let ChooserTheorem =  VOSS varmap2 ChooserGoal;

// Combined proof
let Proof = ALIGNSUB [] stage1 ChooserTheorem;
```

Results: We verified the circuit for different values of n (4, 8, 16, 32, 64, 128). For these values, verification takes roughly cubic time (and importantly, space was not an issue). The verification of the 128 bit problem took just under 27 minutes. Compared to this, verification of the system as one unit was not possible for $n = 64$ or larger.

7.3 Simple example 3

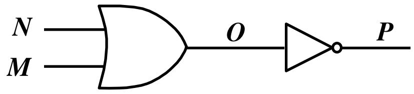

Figure 8: Circuit to illustrate use of AUTOSPTRANS rule.

This example shows off the AUTOSPTRANS rule. Consider the circuit shown in Fig. 8. Assume that the circuit we are verifying consists of two parts. We prove of the first part (using Voss) that if N has the value a, and M has the value b then O produces the value $a \vee b$. We prove of the second part that if O takes the value $\neg(c \wedge f)$ then P produces the value $c \wedge f$. Our overall goal is to show that if N has the value a, and M has the value b then P produces $\neg(a \vee b)$. The AUTOSPTRANS rule can be used to combine the primitive results. The existential quantification described in the previous section computes that there is indeed an appropriate substitution (in this case $\neg a$ for c, and $\neg b$ for f. Note also that the BDDs cope very well with different syntactic expressions for the same semantic object. The proof is expressed as follows:

```
let  list1 = .... bdd ordering omitted
let  list2 = ....
let  A1 = (N ISBOOL a) _&_ (M ISBOOL b)   FROM 0 TO 10;
let  C1 = O ISBOOL (a Or b)  FROM 4 TO 10;
let  T1    = VOSS  list1 (A1 ==>> C1);

let  A2    = O ISBOOL (Not(c And f)) FROM 4 TO 10;
let  C2    = P ISBOOL (c And f) FROM 8 TO 10;
let  T2    = VOSS list2  (A2 ==>> C2);

let  A = (N ISBOOL a) _&_ (M ISBOOL b)  FROM 0 TO 10;
let  C = P ISBOOL  (Not(a Or b)) FROM 8 TO 10;

let  GoalThm = AUTOSPTRANS (A ==>> C) [T1, T2];
```

Executing PrThm GoalThm; results in the appropriate theorem:

```
 |-  [N is (a from 0 to 10)]
     [M is (b from 0 to 10)]
        ==>>
              [P is (NOT((a OR  b)) from 8 to 10)]
```

7.4 Verifying a multiplication circuit

Properties concerning multiplication cannot be verified using BDD based tools alone, since the representation of multiplication by BDDs needs exponentially sized BDDs (Bryant, 1991). In this example we show how a multiplication circuit can be verified using our tool.

7.4.1 The multiplication circuit

The multiplication circuit consists of a series of adders with some additional circuitry. If the bit-width of the of the circuit is b, there will be b stages. The figure below shows an overview of the circuit. The algorithm used is the standard long multiplication algorithm $xy = \Sigma_{i=1}^{b}(2^{i-1}x_i y)$ where x and y are b bit numbers and x_i is the i-th least significant bit of x. The function of the i-th stage is to compute $(2^{i-1}x_i y)$ and add this to the result obtained so far. All arithmetic is done modulo 2^b. An implication of this is that the i-th stage does not use the i higher order bits of y. Therefore the i-th stage has as input b bits which will give the partial sum so far, one bit of x, and $b - i$ bits from y.

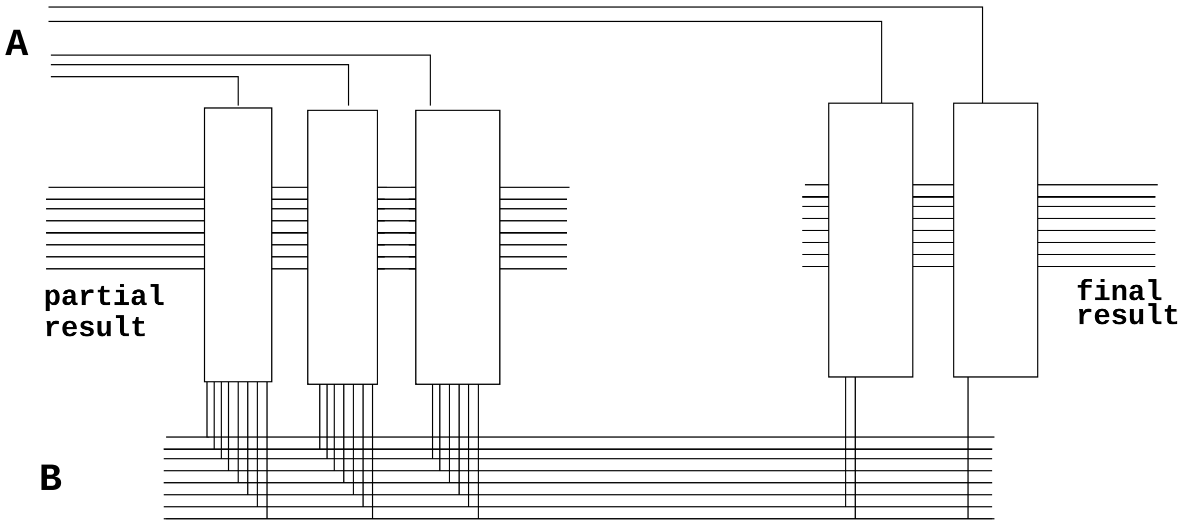

Figure 9: Overview of multiplier

7.4.2 Verification strategy

The key to the proof is to recognise that after the i-th stage, the result computed by the circuit is y multiplied by the i lower order bits of x. Suppose that at the end of the i-th stage we have proven that given the initial input for the circuit, the output of the i-th stage is indeed the i lower order bits of x multiplied by y. At the $(i+1)$-st step we do three things.

Firstly, the local property of the stage is checked – that it actually does the addition and so on. This proof is done using Voss, and, crucially for the efficiency of the checking, is done for arbitrary input values rather than the actual values the circuit will use when executing. Once this check has been done, we compose this result with the result from the i-th stage. Secondly, applying the ALIGNSUB rule (time alignment and specialisation for the new result, and composition of the i-th stage result with the transformed new result), we get what the $(i+1)$-st stage's output is in terms of the whole circuit's input. Finally, the consequent of this theorem is not quite in the form we want, and we use post-condition weakening to obtain that at the end of the $(i+1)$-st stage the partial result computed is the $(i+1)$ lower order bits of x multiplied by y.

This step is then repeated for $i = 1, \ldots, n$. To start off the process, we prove a simple theorem which just states that the input remains constant.

The proof script is given in the appendix. Here we work through a few steps of an eight bit multiplier to illustrate the proof. Assume that the circuit takes two integers on integer node A and integer node B. Each of these nodes is represented in the underlying circuit by 8 boolean nodes

- Prove the trivial theorem that if A has the value x and B has the value y, and $Ground$ has the value zero over a certain time interval then A has the value x and B has the value y, and $Ground$ has the value zero over the same time interval.[6] Using the IDENTITY rule, we obtain T_0 below.

```
|-   [B is (y from 0 to 1000)]
     [A is (x from 0 to 1000)]
     [TC0 is (0 from 0 to 1000)]
        ==>>
              [B is (y from 0 to 1000)]
              [A is (x from 0 to 1000)]
              [TC0 is (0 from 0 to 1000)]
```

- Prove that the first stage — basically an adder — does what it should. Using the VOSS rule we get T_1:

```
|-   [B is (j from 0 to 1000)]
     [A is (i from 0 to 1000)]
     [TC0 is (k from 0 to 1000)]
        ==>>
              [TC1 is ((k + ((i[1])*(j[1..8]))) from 24 to 1000)]
```

- We cannot use transitivity on T_0 and T_1 because they use different variables to describe the value of the nodes. However, if we specialised T_1 we would get the antecedent of T_1 being less than the consequent of T_0. So we use the ALIGNSUB rule to do this[7], getting T_2:

```
|-   [B is (y from 0 to 1000)]
     [A is (x from 0 to 1000)]
     [TC0 is (0 from 0 to 1000)]
        ==>>
              [TC1 is (((x[1])*(y[1..8])) from 24 to 1000)]
```

  Note how 0 is substituted for k.

[6] Introducing the ground nodes simplifies the presentation as in a real implementation the first stage would in fact be different to all the other stages.

[7] In this case we do not have to do any alignment, but in general we have to so that's why it is used. We could have used SPTRANS in this particular case.

- This is almost in the form that we want. The only problem is that in general, the i-th bit of x is not the same as the first i bits of x, and as this is going to be in a loop of the FL program constituting the proof, we use post-condition weakening to get, T_3:

```
|-  [B is (y from 0 to 1000)]
    [A is (x from 0 to 1000)]
    [TC0 is (0 from 0 to 1000)]
       ==>>
             [TC1 is ((y*(x[1..1])) from 24 to 1000)]
```

  In order to do this post-condition weakening, it had to be shown that $y * x[1] = y * x[1..1]$. This can easily be done in the simple integer prover provided. The rule *initrule* is supplied to the POSTWEAK procedure. We have now proven the correctness of the first stage.

- The first step in the second stage is to prove the correctness of the circuitry of the stage. We prove T_4:

```
|-  [B is (j from 0 to 976)]
    [A is (i from 0 to 976)]
    [TC1 is (k from 0 to 976)]
      ==>>
        [TC2 is ((k + (((j[1..7])*2^(1))*(i[2]))) from 24 to 976)]
```

- The second step is to combine T_3 and T_4 using ALIGNSUB. T_4 must be shifted 24 time units, and x, y and (y*(x[1..1])) must be substituted for i, j and k. Doing this we obtain:

```
|-  [B is (y from 0 to 1000)]
    [A is (x from 0 to 1000)]
    [TC0 is (0 from 0 to 1000)]
       ==>>
             [TC2 is (((y*(x[1..1])) + ((2^(1)*(y[1..7]))*(x[2])))
                  from 48 to 1000)]
```

- We want as an invariant after each step that:

```
TCn is y*x[1..n]
```

  Using post-condition weakening fails at first since the system needs to know that $y * x[1..1] + 2^1 * y[1..7] * x[2] = y * x[1..2]$. This is beyond the ability of our simple integer theorem prover to prove. Thus we give the system the general assumption :

```
let mulrule   =
     Assume (((y '* (BWID n x)) '+
              (POW2 n) '* (BWID ('bit_width '- n) y) '* (BIT2 (n '+ '1) x)),
             (y '* (BWID (n '+ '1) x)));
```

(note the substitution of n for 1 in appropriate places, so 1 becomes n, 2 becomes $n+1$, and 7 becomes $bit\_width - n$). Now use POSTWEAK to obtain:

```
 |-  [B is (y from 0 to 1000)]
     [A is (x from 0 to 1000)]
     [TC0 is (0 from 0 to 1000)]
        ==>>
              [TC2 is ((y*(x[1..2])) from 48 to 1000)]
  ASSUMING
((y*(x[1..n])) + ((2^(n)*(y[1..(8-n)]))*(x[(n + 1)])))]=(y*(x[1..(n + 1)]))
```

Note that any assumptions which are used are made explicit.

- We now repeat this process for each stage in the multiplier. The final stage (which we do not show for space reasons) is to use post-condition weakening to derive the final result. Assuming eight-bit integers, we get as the final result

```
 |-  [B is (y from 0 to 1000)]
     [A is (x from 0 to 1000)]
     [TC0 is (0 from 0 to 1000)]
        ==>>
              [TC8 is ((x*y) from 192 to 1000)]
  ASSUMING
((y*(x[1..n])) + ((2^(n)*(y[1..(8-n)]))*(x[(n + 1)])))]=(y*(x[1..(n + 1)]))
```

7.4.3 Results

By proving the properties of each stage of the circuit separately and using the rules of combination a number of advantages are gained. Firstly, since we are still using Voss for the low-level proof we keep the advantages of trajectory evaluation. Secondly, since we prove the partial results for arbitrary inputs rather that complicated specific cases, we can avoid the exponential growth of BDDs and thereby make the method tractable. All time-shifting and specialisations are derived automatically, simplifying the task of the user.

This process verifies the entire circuit, including ensuring that different parts of the circuit are correctly connected.

The verification of a 64-bit multiplier took just less than fourteen CPU minutes on a Sun Sparc 10/51 processor. The performance is roughly quadratic in the bit-width, so this problem can easily be dealt with.

It is difficult to estimate the amount of human effort involved in the proof, since the proof went hand-in-hand with system development (and circuit debugging!). It turned out that one of the most difficult parts of the verification was getting the timing constraints correct. Our estimate is that the proof itself took two days to get right.

8 Conclusion

8.1 Summary

We have proposed a theorem-prover based on symbolic trajectory evaluation and OBDDs. Combining these two methods of verification has resulted in a powerful theory and tool for low-level hardware verification and the combination of results. In our largest example, we verified the correctness of a 64-bit multiplier (a circuit consisting on the order of twenty-five thousand gates) using approximately fifteen minutes of CPU time.

This was obtained by exploiting the strengths of Voss, and providing (i) a means of overcoming its weaknesses, and (ii) a method for proof management.

The strength of Voss is that it allows the use of symbolic trajectory evaluation to verify low-level properties of circuits very efficiently. With Voss, we also have an accurate model of the circuit, including timing.

A weakness of Voss is that it relies on one rule for obtaining symbolic trajectory evaluation results. It proves results of the form $\models_{\mathcal{M}} [A \Longrightarrow\!\!\!> B]$ by checking whether $\delta_B \sqsubseteq \tau_A$. The combined task of computing and comparing trajectories can be very expensive (in general the problem is NP-hard). We have developed a theory and inference system which allows us to infer that $\delta_B \sqsubseteq \tau_A$ without computing τ_A. In the theorem prover implementing this inference system and providing a powerful and flexible proof management system, a key issue is the use of domain knowledge. For example, when proving properties of circuits which manipulate integers, using properties of integers obviates the need for representing everything using OBDDs, making the analysis of a large class of circuits tractable.

Although the tool also allows re-use of results, the examples did not show this off to its best advantage. In the multiplication example, instead of constructing it out of sixty-four stages, we could have implemented the multiplier in one stage, with the output being fed back into the input – slightly more complicated circuitry but not too different. In this case we could use Voss once to prove how it worked, and then use the time-shift rule 63 times. As the time-shift rule is one to two orders of magnitude faster than trajectory evaluation this would result in significant saving.

We believe that we have shown:

- The use of hybrid methods gives us flexibility and power without losing rigour.
- The use of domain knowledge is very important in gaining efficiency.
- Composition of results is important: whether we have proved results of different parts of the circuit, or want to combine smaller results of the same part of the circuit, or re-use results, composition is a very powerful technique. Verification is made easier to understand, and, for reasons discussed earlier, much more efficient.
- In particular, symbolic trajectory evaluation results can be composed. This enables the derivation of symbolic trajectory evaluation results without having to explicitly perform the trajectory evaluation.

8.2 Problems and extensions

8.2.1 Integrating domain knowledge

For the purposes of the work done so far, which we see as a proto-type for further work, we implemented a fairly *ad hoc* approach to incorporating domain knowledge. We implemented a simple theorem prover to make arguments about integers, and used simple decision procedures to keep integer expressions in some sort of normalised form. We saw the limitations of this. Firstly, the prover was not very powerful. Secondly, the interplay between the simple decision procedures and the theorem prover made the use of the system a bit clumsy in parts, due to the *ad hoc* nature of the approach. Finally, the security of an *ad hoc* approach must be questionable.

Therefore it is essential that there be a proper interface to a proper theorem prover in which arguments about domain knowledge can be made (a very important case is arguments about integers). Theorem-provers with appropriate libraries (like HOL), or decision procedures or reasoning tools for integers like PVS (Owre *et al.*, 1992) and Analytica (Clarke & Zhao, 1992) would be appropriate; criteria for choice include expertise of intended user, desired level of flexibility and security, and domain of application.

Arguments about the combination of theories may also need to be made. We need to explore work such as (Shostak, 1979; Shostak, 1984) to see how this can be done in a coherent way so as to make integrating new theories easy and rigorous.

8.2.2 Soundness of methods

While in principle we have a sound theory, the implementation has a number of weaknesses. The *ad hoc* implementation of the integer theorem prover is one hole. Another problem is that although we have separated out the trusted and non-trusted parts of the system, there is a fairly large, trusted core.

8.2.3 Completeness of rules

Let $\mathcal{A}$ be a set of trajectory assertions. Consider the class $\mathcal{U}$ of machines which satisfy all these assertions. A number of interesting theoretical questions arise:

- Does there exist a unique "weakest" machine in $\mathcal{U}$?
- Suppose for each $M \in \mathcal{U}, \models_M A, A \notin \mathcal{A}$. Can A be inferred from $\mathcal{A}$ using the inference rules above?

These are questions for future research.

8.2.4 Structural versus behavioural

One of the properties of symbolic trajectory evaluation is that verification is behavioural rather than structural. We move away from a purely behavioural model in two ways. Our model of the system is entirely behavioural. However, it is important for the usefulness of our system that sufficient meaningful structure be identifiable (there's nothing stopping the treating of any arbitrary eight binary nodes as an 8-bit integer; however, unless this corresponds to reality, this is unlikely to be very useful). Second, structural composition of the *specification* is a fundamental part of efficient proof management. This issue needs to be explored further to improve performance and allow composition of circuits.

8.2.5 False implies everything

In logic, false being in the protasis of an implication makes the implication true trivially. Theorems of the form *if pigs have wings, then this circuit works*, while logically valid are problematic since they may lull verifiers into a false sense of security. In Voss this problem manifests itself when top appears in the antecedent of a trajectory assertion. Thus in theory, nonsense theorems are possible. However, Voss checks whether there is at least one valid trajectory. In this tool, by moving away from the Voss representation, we lose this automatic checking of security. Furthermore, some of the inference rules admit the possibility of obtaining inconsistent antecedents (obvious cases being the conjunction and and time-shift rules, and specialisation). While mathematically this does not pose a problem, as a practical tool, we want any specification which has such an inconsistency to be rejected as it will indicate an error in the specification.

There are two factors which reduce the seriousness of the problem. Firstly, the greatest danger of such nonsense theorems is that they become buried — they get used in the middle of a proof and the final result does not reflect this. This does not occur here. If at any stage, we obtain an inconsistent antecedent, if that theorem is used by any combination rule, then the resulting theorem must also have an inconsistent antecedent. This means that in such a case, the final result of a proof would have such an inconsistency explicit in its antecedent. Second, in practice, the antecedent of a theorem is likely to be of a simple enough form that a BDD-representation would be reasonably efficient to obtain. This being the case, it may be reasonably efficient to allow the user to check a final result for consistency. This needs further research.

8.2.6 Heuristics and User-assistance

The usefulness of a tool like this depends in a large measure on the ease of use and assistance given to a user. The heuristics used to find appropriate time-shifts and specialisations are very important here. Although the heuristics performed well in the examples which we have used, an issue for further research is how these heuristics can be improved, and finding other heuristics or other kinds of assistance which could be provided. As an example, we found timing to be one of the most difficult aspects of getting the verification right — assistance here would be very valuable. There are also a range of user-interface issues which could be explored to make the use of such a tool pleasanter.

A Multiplication example

This section contains the proof of the verification of the 64-bit multiplication example shown in section 7. The multiplication circuit itself was generated by a VHDL program and consists of the order of twenty-five thousand gates. The FL program which verifies the circuit is shown below.

```
let Ground = "TC0";
// variable maps .....omitted for brevity....
let prevarmap = ...
let varmap n =  ...

// Mathematical results
let mulrule = Assume (((y '* (BWID n x)) '+
                           (POW2 n) '* (BWID ('bit_width '- n) y) '*
                           (BIT2 (n '+ '1) x)),
                        (y '* (BWID (n '+ '1) x)));

let  initrule = BDDVerify []
                  ((BIT2 one x) '* (BWID bwidth y), y '* (BWID one  x));

let  width2   = BDDVerify [] (BWID (' c_size) i, i);

// Preamble theorem--------------------------------------------
let signal_length = 1000;

let preamble = ("A" ISINT x) _&_
               ("B" ISINT y) _&_
               (Ground ISINT zero)
                            FROM 0 TO signal_length;
let preambleThm = IDENTITY preamble;


// GENERAL STEP------------------------------------------------
// This is the proof that the n-th stage in the multiplier works
// correctly

// Timing considerations -- each successive stage has the
// signal for less time
let answer_delay  = 3*bit_width;
let start_stage n = n*answer_delay;
let signal_len  n = signal_length-n*answer_delay;

let stage n  =
     let Ainp= "A" in
     let Binp= "B" in
     let Cinp= "TC"^(num2str n) in
     let Cout= "TC"^(num2str (n+1)) in
     let nplusone = '(n+1) in
     let jpart    = BWID ('(bit_width - n)) j in
      ((( Ainp ISINT i) _&_
        ( Binp ISINT j) _&_
        ( Cinp ISINT k)) FROM 0  TO signal_len n)
      ==>>
        (Cout ISINT
             (k '+ (jpart '* ((BIT2 nplusone i) '* (POW2 ('n)) )))
                      FROM   answer_delay
                        TO   signal_len n);
```

```
let multhm n = VOSS (varmap n)  (stage n);

// Wish to show that the partial result computed from the
// composition of stages 1..n is correct

let induc  n =
     let n1 = n+1 in
     let n2 = int2str n1 in
     let Ainp="A" in
     let Bout= "B" in
     let Cout= "TC"^n2 in
     let this_width = bit_width - n1 in
         Cout ISINT ((BWID ('n1) x) '*  y)
                       FROM start_stage n+answer_delay TO signal_length;

// Each step in the proof:
//    1. prove the n-th stage works
//    2. use ALIGNSUB to compose (1) with the proof that the
//       previous stage computed what it should be
//    3. Use POSTWEAK to show the "induction" step

let  inferencestep n start =
   let thisthm = multhm n in
   let newthm2 = ALIGNSUB [] start thisthm in
      POSTWEAK [initrule, mulrule]  newthm2 (induc n);

// Postamble
//   Use POSTWEAK to get the result in the form we want

let postamble=
     let prop_delay = start_stage bit_width in
     let gate = "TC" ^ (num2str c_size) in
       (gate ISINT ( x '* y))
                        FROM prop_delay TO signal_length;

// Proof
let  do_proof n  =
      let steps = gen (n-1)  in
      let proof = rev_itlist inferencestep steps preambleThm in
      let final = POSTWEAK [width2] proof postamble in
          final;
```

References

Beatty, D., Bryant, R.E., & Seger, C.-J. 1991 (June). Formal Hardware Verification by Symbolic Ternary Trajectory Evaluation. *In: Proceedings of the 1991 IEEE/ACM Design Automation Conference.*

Boyer, R.S., & Moore, J.S. 1988. *A Computational Logic Handbook.* Academic Press.

Bradfield, J.C. 1992. A Proof Assistant for Symbolic Model-Checking. *In:* (von Bochmann & Probst, 1992).

Brookes, S.D., Hoare, C.A.R., & Roscoe, A.W. 1984. A Theory of Communicating Sequential Processes. *Journal of the Association for Computing Machinery*, **31**(3), 560–599.

Bryant, Randal E. 1991. On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication. *IEEE Transactions on Computers*, **20**(2), 205–213.

Bryant, Randal E. 1992. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, **24**(3), 293–318.

Burch, J.R., Clarke, E.M., & McMillan, K.L. 1992. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, **98**(2), 142–170.

Clarke, Edmund, & Zhao, Xudong. 1992. Analytica – A Theorem Prover in Mathematica. *In:* (Kapur, 1992).

Clarke, E.M., Emerson, E.A., & Sistla, A.P. 1983. 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: ACM.

Clarke, E.M., Long, D.E., & McMillan, K.L. 1989. Compositional Model Checking. *In: IEEE Fourth Annual Symposium on Logic in Computer Science.* Washington, D.C.: IEEE Computer Society.

Cleaveland, R., Parrow, J., & Steffen, B. 1989. The Concurrency Workbench. *Pages 24–37 of:* Sifakis, J. (ed), *Lecture Notes in Computer Science 407: Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems.* Berlin: Springer-Verlag.

Coudert, O., Madre, J.C., & Berthet, C. 1990. Verifying Temporal Properties of Sequential Machines Without Building their State Diagram. *In:* Clarke, E.M., & Kurshen, R.P. (eds), *CAV '90: Proceedings of the 2nd International Workshop on Computer-Aided Verification.* Lecture Notes in Computer Science 531. Berlin: Springer-Verlag.

Courcoubetis, C. (ed). 1993. *Proceedings of the 5th International Conference on Computer-Aided Verification.* Lecture Notes in Computer Science 697. Berlin: Springer-Verlag.

Donat, Michael. 1993 (Apr.). *Verification Using Abstract Domains.* Unpublished paper, Department of Computer Science, University of British Columbia.

Gordon, M.J.C. 1987. HOL: A Proof Generating System for Higher-Order Logic. *In:* Birtwistle, G., & Subrahmanyam, P.A. (eds), *VLSI Specification, Verification and Synthesis.* Boston: Kluwer.

Gordon, M.J.C. (ed). 1993. *Introduction to HOL: a theorem proving environment for higher order logic.* Cambridge: Cambridge University Press.

Gordon, M.J.C., Wadsworth, C.P., & Milner, R. 1979. *Edinburgh LCF : a mechanised logic of computation*. Lecture Notes in Computer Science 78. Berlin: Springer-Verlag.

Groote, J.F., & Moller, F. 1992. Verification of Parallel Systems via Decomposition. *Pages 62–76 of:* Cleaveland, W.R. (ed), *CONCUR '92: Proceedings of the Third International Conference on Concurrency Theory*. Lecture Notes in Computer Science 630. Berlin: Springer-Verlag.

Harrison, J., & Thery, L. 1993 (Aug.). Extending the HOL Theorem Prover with a Computer Algebra System to Reason about the Reals. *In: Proceedings of the HOL User's Group Workshop*.

Hungar, Hardi. 1993. Combining Model Checking and Theorem Proving to Verify Parallel Processes. *In:* (Courcoubetis, 1993).

Joyce, Jeffrey J. 1989. *Multi-level Verification of Microprocessor-Based Systems*. Ph.D. thesis, University of Cambridge.

Joyce, Jeffrey J., & Seger, Carl-Johan H. 1993. Linking BDD-based Symbolic Evaluation to Interactive Theorem-Proving. *In: Proceedings of the 30th Design Automation Conference*. IEEE Computer Society Press.

Kapur, Deepak (ed). 1992. *Proceedings of the 11th International Conference on Automated Deduction — CADE-11*. Lecture Notes in Computer Science 607. Berlin: Springer-Verlag.

Kurshan, R.P., & Lamport, L. 1993. Verification of a Multiplier: 64 Bits and Beyond. *In:* (Courcoubetis, 1993).

Larsen, Kim, & Thomsen, Bent. 1991. Partial Specifications and Compositional Verification. *Theoretical Computer Science*, **88**(1), 15–32.

Long, David E. 1993 (July). *Model Checking, Abstraction, and Compositional Verification*. Ph.D. thesis, Carnegie-Mellon University, School of Computer Science. Technical report CMU-CS-93-178.

McFarland, Michael C. 1993. Formal Verification of Sequential Hardware: A Tutorial. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **12**(5), 633–654.

McIsaac, Anthony. 1993 (Aug). A Formalization of Abstraction in LAMBDA. *Pages 229–240 of: Proceedings of the 1993 HOL Users' Group Workshop*.

Milner, Robin. 1984. A Proposal for Standard ML. *Pages 184–197 of: Proceedings of the ACM Conference on LISP and Functional Languages*. New York: ACM.

Milner, Robin. 1989. *Communication and Concurrency*. London: Prentice-Hall International.

Owre, S., Rushby, J.M., & Shankar, N. 1992. PVS: A Prototype Verification System. *In:* (Kapur, 1992).

Paulson, L.C. 1987. *Logic and computation : interactive proof with Cambridge LCF*. Cambridge: Cambridge University Press.

Seger, Carl-Johan H., & Bryant, Randal E. 1993 (Apr.). *Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories.* Technical Report 93-8. Department of Computer Science, University of British Columbia.

Seger, Carl-Johan H., & Joyce, Jeffrey J. 1992 (Dec.). *A Mathematically Precise Two-Level Hardware Verification Methodology.* Technical Report 92-34. Department of Computer Science, University of British Columbia.

Shiple, T.R., Chiodo, M., Sangiovanni-Vincentelli, A.L., & Bryton, R.K. 1992. Automatic Reduction in CTL Compositional Model Checking. *In:* (von Bochmann & Probst, 1992).

Shostak, Robert E. 1979. A Practical Decision Procedure for Arithmetic with Function Symbols. *Journal of the Association for Computing Machinery*, **26**(2), 351–360.

Shostak, Robert E. 1984. Deciding Combinations of Theories. *Journal of the Association for Computing Machinery*, **31**(1), 1–12.

von Bochmann, G., & Probst, D.K. (eds). 1992. *CAV '92: Proceedings of the Fourth International Workshop on Computer Aided Verification.* Lecture Notes in Computer Science 663. Berlin: Springer-Verlag.

Zhu, Z., Joyce, J., & Seger, C. 1993 (Aug). Verification of the Tamarack-3 Microprocessor in a Hybrid Verification Environment. *Pages 255–268 of: HOL User's Group Workshop.*