Composing Symbolic Trajectory Evaluation Results*

Scott Hazelhurst and Carl-Johan H. Seger

Department of Computer Science, University of British Columbia, Vancouver, Canada V6T 1Z4
E-Mail: {shaze, seger}@cs.ubc.ca

Abstract. Symbolic trajectory evaluation shows much promise as a method for verifying large scale VLSI designs with a high degree of automation. However, to verify today's designs, a method for composing partial verification results is needed. Consequently, we have proven a number of inference rules for the composition of symbolic trajectory evaluation results and developed a specialised 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 conclude with an example in which we verify a design that could not have been verified directly. In particular, the complete verification of a 64 bit multiplier takes under 15 minutes on a Sparc 10/51 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. There are a number of different approaches to hardware verification [10]. This paper presents a new hybrid technique based on theorem-proving and symbolic trajectory evaluation (STE).

1.1 Symbolic Trajectory Evaluation – STE

If the state space of a system (for example a circuit) can be embedded in 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. We define a partial order between sequences of states by extending the partial order on the state space in a natural way.

* 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.

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 ($\mathcal{S}$ being the state space and $\sqsubseteq$ a partial order on $\mathcal{S}$) and Y is a monotone successor function $Y: \mathcal{S} \rightarrow \mathcal{S}$. A sequence is a *trajectory* if and only if $Y(\sigma^i) \sqsubseteq \sigma^{i+1}$ for $i \geq 0$.

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, but expressive enough to capture many of the properties we need to check for.

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

1. **Simple predicates**: Every simple predicate over $\mathcal{S}$ is a trajectory formula.
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 a Boolean expression.
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. Whether a trajectory $\tilde{\sigma}$ satisfies a formula F (written as $\tilde{\sigma} \models_{\mathcal{M}} F$) is given by the following rules.

1. $\sigma^0 \tilde{\sigma} \models_{\mathcal{M}} p$ iff $p(\sigma^0) = true$.
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. $\sigma \models_{\mathcal{M}} (e \rightarrow F)$ iff $\phi(e) \Rightarrow \sigma \models_{\mathcal{M}} F$, for all ϕ mapping Boolean expressions to $\{true, false\}$.
4. $\sigma^0 \tilde{\sigma} \models_{\mathcal{M}} \mathbf{N}F$ iff $\tilde{\sigma} \models_{\mathcal{M}} F$.

Given a formula F there is a unique *defining sequence*, δ_F, which is the weakest sequence which satisfies the formula[2]. The defining sequence can usually be computed very efficiently. From δ_F a unique *defining trajectory*, τ_F, can be computed (often efficiently). This is the weakest trajectory which satisfies the formula — all trajectories which satisfy the formula must be greater than it in terms of the partial order.

If the main verification task can be phrased in terms of "for every trajectory σ that satisfies the trajectory formula A, verify that the trajectory also satisfies the formula C", verification can be carried out by computing the defining trajectory for the formula A and checking that the formula C holds for this trajectory. We write such a result as $\models_{\mathcal{M}} [A \Longrightarrow\!\!\gg C]$. The fundamental result of STE is given below.

Theorem 1. *Assume A and C are two trajectory formulas. Let τ_A be the defining trajectory for formula A and let δ_C be the defining sequence for formula C. Then $\models_{\mathcal{M}} [A \Longrightarrow\!\!\gg C]$ iff $\delta_C \sqsubseteq \tau_A$.*□

[2] "Weakest" is defined in terms of the partial order.

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

STE uses ordered binary-decision diagrams (OBDDs or BDDs) [2] for efficient manipulation of boolean expressions. Using OBDDs, boolean expressions have canonical forms making comparison of expressions very efficient. Though BDDs 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.

The Voss system, a formal hardware verification system developed at the University of British Columbia, consists of three major components: a highly efficient implementation of OBDDs; 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. Voss has been used to perform STE efficiently on large, sophisticated circuits. A full description of trajectory evaluation is given in [12]. Although many circuits can be verified very efficiently, there are limitations — these have have motivated this work.

1.2 Motivation

Our approach improves on STE by

1. Raising the level of abstraction, making verification more convenient for the user, and, more importantly, allowing us to overcome the limitations of BDDs.
2. Developing inference rules for the combination of STE results.
3. Using a specialised theorem-prover implementing these rules, allowing us to exploit the strengths of theorem-proving while still keeping the advantages of STE.

2 Related Work

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

Compositionality is the divide-and-conquer approach of verification, promoting usability and efficiency. It allows re-use of results and, most importantly, it greatly reduces the state space of systems that need to be explored. Compositionality has been used in a number of different verification techniques, for example process algebras [11], model checking [3, 9], and theorem-proving.

By hybrid methods, we mean the combined use of different verification methods in a rigorous and sound way. One of the first systems which combined different approaches rigorously was the HOL-Voss system [13, 7], linking the HOL

theorem-prover and the Voss STE system. This enables the proof of something within one system to be carried over into the other system. Although HOL-Voss is a much more powerful and general system, it is not clear that all the extra power is useful, and it is much more difficult to learn to use. Further, our system has the rules for composing symbolic trajectory evaluation results built in.

Kurshan and Lamport have combined the COSPAN model-checker with the TLP theorem prover [8]. The model checker proves properties of components of the system, which are then translated into a form suitable for the theorem-prover. In order to prove the overall result, a number of sub-results need to be proved (including a hand-checked step).

Hungar also links model checking and theorem-proving [6]. The model is given by a Kripke structure representing the semantics of an Occam program, and the properties are expressed in a variant of CTL. The results generated by model-checking are combined using the LAMBDA theorem-prover. Given an Occam program consisting of a number of processes, properties can be proven of each process using the model-checker. A number of rules — some analogous to the ones we propose in Section 3 — can be used by the theorem-prover to combine these sub-properties to prove properties of the entire program.

3 Inference Rules

The key result of STE is Theorem 1: from $\delta_C \sqsubseteq \tau_A$, we can infer $\models_{\mathcal{M}} [A \Longrightarrow\!\!\gg C]$. This section develops other rules for inferring these results, enabling re-use and composition of old results, and incorporation of domain knowledge. To verify a circuit, we use STE to prove the low level properties of the circuit, and then use these rules to combine the low-level results into higher-level results. Throughout the section we shall use the circuit shown in Fig. 1 as a simple example. We omit all proofs for space reasons — the proofs can be found in [5].

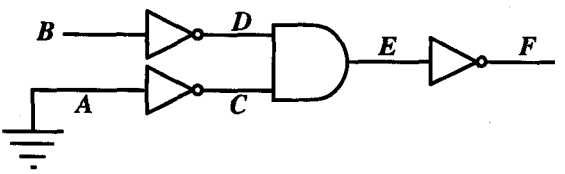

Fig. 1. Circuit C_2.

3.1 Identity Inference Rule

Although this theorem is very simple, it is useful.

Theorem 2. *If A is any trajectory formula then* $\models_{\mathcal{M}} [A \Longrightarrow\!\!\gg A]$. □

3.2 Time Shift Inference Rule

This result allows us to move the "base time" of a result: 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. This allows us to abstract details of time in certain places and promotes re-use of STE results.

Theorem 3. *For any trajectory formulas A and C, if $\models_{\mathcal{M}} [A \Longrightarrow\!\!\gg C]$ then $\models_{\mathcal{M}} [N^t A \Longrightarrow\!\!\gg N^t C]$ for any $t \geq 0$.* □

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

3.3 Post-condition Weakening and Pre-condition Strengthening

The following theorem is equivalent to the classical post-condition weakening and pre-condition strengthening. Importantly, semantic as well as syntactic information is used when deciding the ordering of two trajectories (shown later).

Theorem 4. *Let A, C, A_1 and C_1 be trajectory formulas. Suppose $\models_{\mathcal{M}} [A \Longrightarrow\!\!\gg C]$. If $\delta_A \sqsubseteq \delta_{A_1}$, then $\models_{\mathcal{M}} [A_1 \Longrightarrow\!\!\gg C]$. If $\delta_{C_1} \sqsubseteq \delta_C$, then $\models_{\mathcal{M}} [A \Longrightarrow\!\!\gg C_1]$.* □

Example
In the example circuit, STE shows that $\models_{\mathcal{M}} [(B \text{ is } 0) \wedge (\mathbf{N}(B \text{ is } 0)) \Longrightarrow\!\!\gg (\mathbf{N}^2(E \text{ is } 1) \wedge \mathbf{N}^3(E \text{ is } 1))]$. Using post-condition weakening we can prove $\models_{\mathcal{M}} [(B \text{ is } 0) \wedge \mathbf{N}(B \text{ is } 0)) \Longrightarrow\!\!\gg \mathbf{N}^2(E \text{ is } 1)]$.

3.4 Conjunction Rule

The following theorem is useful when properties about separate parts of the system have been proven and we now want to combine these results to reason about the combination. It is also useful when we have different results about the same part of the circuit and wish to combine them.

Theorem 5. *Let A_1, C_1, A_2, and C_2 be trajectory formulas. If $\models_{\mathcal{M}} [A_1 \Longrightarrow\!\!\gg C_1]$ and $\models_{\mathcal{M}} [A_2 \Longrightarrow\!\!\gg C_2]$, then $\models_{\mathcal{M}} [(A_1 \wedge A_2) \Longrightarrow\!\!\gg (C_1 \wedge C_2)]$.* □

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

3.5 Transitivity Inference Rule

This rule is analogous to the transitivity rule in logic: if $A \Rightarrow B$ and $B \Rightarrow C$ then $A \Rightarrow C$.

Theorem 6. *Let A_1, C_1, A_2, and C_2 be trajectory formulas.*
Suppose $\models_{\mathcal{M}} [A_1 \Longrightarrow\!\!\gg C_1]$ and $\models_{\mathcal{M}} [A_2 \Longrightarrow\!\!\gg C_2]$. If $\delta_{A_2} \sqsubseteq \text{lub}(\delta_{A_1}, \delta_{C_1})$ then $\models_{\mathcal{M}} [A_1 \Longrightarrow\!\!\gg C_2]$. □

Example
We can use STE to show of the example circuit that $\models_{\mathcal{M}} [B \text{ is } 0 \Longrightarrow\!\!\gg \mathbf{N}^2(E \text{ is } 1)]$ and that $\models_{\mathcal{M}} [\mathbf{N}^2(E \text{ is } 1) \Longrightarrow\!\!\gg \mathbf{N}^3(F \text{ is } 0)]$. Using transitivity, we can conclude, without having to compute anything else that $\models_{\mathcal{M}} [B \text{ is } 0 \Longrightarrow\!\!\gg \mathbf{N}^3(F \text{ is } 0)]$.

3.6 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 $\Longrightarrow\!\!\gg$ O is 2*a]** we may wish to generate **[M is 1 $\Longrightarrow\!\!\gg$ O is 2*1]** or **[M is a+b $\Longrightarrow\!\!\gg$ O is 2*(a+b)]**.

Definition 3.1 *A* simple substitution *σ is a function from a set of variables, $\tilde{I}$, to expressions over constants and these variables, $\mathcal{E}_{\tilde{I}}$.* □

Thus, if σ is a simple substitution and F is a trajectory formula, $\sigma(F)$ is F rewritten by replacing variables in F with the appropriate expressions given by the mapping.

Theorem 7 Simple Substitution Theorem. *Suppose $\models_{\mathcal{M}} [A \Longrightarrow\!\!\gg C]$, and σ is a simple substitution. Then $\models_{\mathcal{M}} [\sigma(A) \Longrightarrow\!\!\gg \sigma(C)]$.* □

A substitution defines what should be substituted for which variables throughout an assertion, and the simple substitution theorem shows that this is a valid result. A more sophisticated way of transforming an STE result is *specialisation.* Suppose we have two substitutions σ_1 and σ_2. Using the Simple Substitution Theorem we can derive the two results $\models_{\mathcal{M}} [\sigma_1(A) \Longrightarrow\!\!\gg \sigma_1(C)]$ and $\models_{\mathcal{M}} [\sigma_2(A) \Longrightarrow\!\!\gg \sigma_2(C)]$. Now using Theorem 5, we can derive the combined result $\models_{\mathcal{M}} [\sigma_1(A) \wedge \sigma_2(A) \Longrightarrow\!\!\gg \sigma_1(C) \wedge \sigma_2(C)]$. We can generalise this in the following way.

Let $\tilde{\sigma} = [(c_1, \sigma_1), \ldots, (c_n, \sigma_n)]$, where each c_i is a boolean expression, and each σ_i a simple substitution. Then $\tilde{\sigma}$ is a *specialisation*, and if F is a trajectory formula, then $\tilde{\sigma}(F) = \wedge_i (c_i \rightarrow \sigma_i(F))$. The following result holds.

Theorem 8 Specialisation Theorem. *Let $\hat{\sigma}$ be specialisation. If $\models_{\mathcal{M}} [A \Longrightarrow\!\!\gg C]$ then $\models_{\mathcal{M}} [\hat{\sigma}(A) \Longrightarrow\!\!\gg \hat{\sigma}(C)]$*

Specialisation helps gluing together the different building blocks of a proof — often before using transitivity it is necessary to perform specialisation. As a simple example, suppose we have a circuit which takes in two numbers x and y and outputs $2*\max(x,y)$. The circuit consists of two parts: one part compares x and y and outputs the greater; the second part doubles the output of the first part. Using STE, natural results we might prove are:

$$T_1 = \models_{\mathcal{M}} [(A \textbf{ is } x) \wedge (B \textbf{ is } y) \Longrightarrow\!\!\gg \mathbf{N}(((x > y) \rightarrow (T \textbf{ is } x)) \wedge (\neg(x > y) \rightarrow (T \textbf{ is } y)))]$$

and

$$T_2 = \models_{\mathcal{M}} [\mathbf{N}(T \textbf{ is } a) \Longrightarrow\!\!\gg \mathbf{N}^2(C \textbf{ is } (a + a))].$$

Now using specialisation, from the latter result we can obtain:

$$T_2' = \models_{\mathcal{M}} [\mathbf{N}(((x > y) \rightarrow (T \textbf{ is } x)) \wedge (\neg(x > y) \rightarrow (T \textbf{ is } y))) \Longrightarrow\!\!\gg \mathbf{N}^2((x > y) \rightarrow (C \textbf{ is } (x + x)) \wedge ((\neg(x > y)) \rightarrow (C \textbf{ is } (y + y))))].$$

Transitivity can now be applied between T_1 and T_2' to obtain the overall result which we want.

Specialisation also improves efficiency. In this example, it was faster to prove $T2$ than $T2'$ since the OBDDs needed in the verification of $T2$ are smaller than the ones in $T2'$. Although this example is so small that there is little real difference here, in larger examples, there can be an enormous practical difference. Thus, while it may be more indirect to prove a result using STE for a more general case (with simpler OBDDs) and then to specialise, it is far more efficient than directly using STE to obtain the desired result.

4 Implementing a useful tool

4.1 Implementation of Inference Rules

The theory described so far 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.

The system has defined abstract data types for representing and manipulating trajectories and the appropriate values (like integers). The inference rules return objects of type `Theorem`. Only the inference rules can manipulate these objects, guaranteeing soundness of 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.

All the inference rules described in the previous section have been implemented. In addition more complex rules which package the basic rules have been implemented so as to ease the job of the user. A very important part of the packaging is the provision of heuristics which help the user in finding specialisations and time-shifts. For example, before being able to use transitivity between two results T_1 and T_2, it may be necessary to specialise the latter or time-shift

one of them. Of course, the user can provide these transformations, but it would be better if these transformations could be automatically found. Although the heuristics we have provided are fairly simple, they are very useful and in the example verifications we have done have proven quite adequate. What should be emphasised is that the heuristics are integrated into the system in a secure way. If the heuristics are inadequate (or error-prone) it may not be possible to prove correct results; it will not be possible to prove incorrect results. For space reasons a discussion of the heuristics is beyond the scope of this paper.

4.2 Raising the Level of Abstraction

Besides helping the verification process by reducing the semantic gap between the user's understanding and the circuit being verified, abstraction has key efficiency advantages.

All verification results are represented by our tool as an abstract data type. Being able to translate from this abstract data type to OBDDs is essential, but using a more abstract representation means we no longer have to rely on OBDDs alone. Although there are disadvantages in this, we can exploit the strengths of OBDDs wherever possible, yet overcome their limitations where necessary.

There are two ways in which this is done. First, using a more abstract representation increases the usefulness of specialisation. General forms of a result can be proven using STE where the general form has an efficient OBDD representation. Specialisation can then yield the result we want, even if the specialised version of the result has no efficient OBDD representation.

Second, we no longer rely on OBDDs as the sole means of arguing about the equality of two expressions. In the HOL-Voss system it was shown how to prove in an integrated system that two expressions have the same OBDD (and hence are semantically identical) without having to construct the OBDDs [7]. We adopt a similar approach by supplying a simple integer arithmetic theorem prover. With the domain knowledge this prover provides and the inference rules described earlier, we are able to reason about circuits that we would not be able to reason about otherwise.

As a simple example, we may have proven that a certain node obtains some value e_1 where e_1 is a complicated arithmetic expression. Using domain knowledge and trivial postcondition weakening, we may be able to show that the node obtains the value e_2 where e_2 is a much simpler expression denoting the same value as e_1.

Additional abstraction is provided by allowing the user to describe a mapping between the names of the nodes in the physical circuit being verified and logical names used in the proof. This is useful in referring to a group of physical nodes by a group name, rather than by referring to all their individual names. Instead of specifying the values of all 64 bit-valued nodes representing some integer value, the proof can refer to one integer-valued node. This syntactic sugar makes proofs much pleasanter, and library routines are provided to aid the creation of the mapping. The user needs to bear in mind that the results obtained actually are about the low-level circuit and not some "abstract" machine.

5 Example

For space reasons, we only give one example; however, we have applied this method successfully to a number of other examples. The problem we concentrate on as an example here is the verification of a simple combinational multiplier circuit. (We have verified a Wallace-tree multiplier too — although the proof is not much more complicated than the example we show here, it is easier to describe this proof.)

Ideally, the verification would be single STE result relating the output to the inputs. However, properties concerning multiplication cannot be verified using BDD based tools alone, since the representation of multiplication by BDDs needs exponentially sized BDDs [1]. In this example we show how a multiplication circuit can be verified using our tool by breaking the proof into parts and composing the results, thereby achieving a result equivalent to the straightforward STE verification.

5.1 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, which is implemented as a VHDL program and consists of approximately 25 000 gates (in our case, $b = 64$). 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. The input of x and y is on nodes A and B respectively.

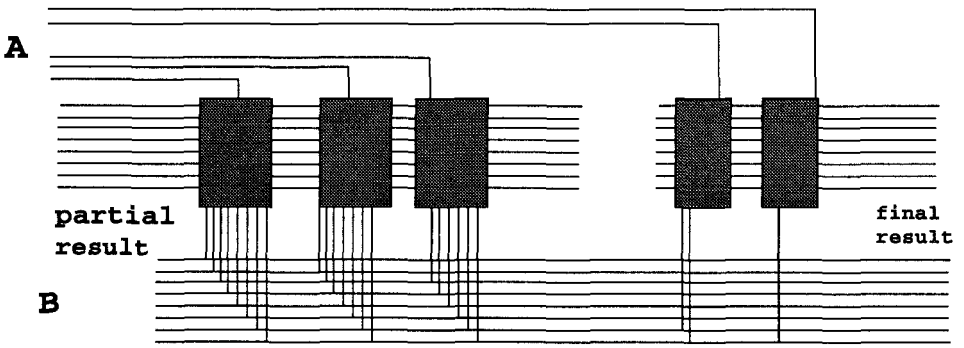

Fig. 2. Overview of Multiplier

5.2 Verification Strategy

The verification of the multiplier entails:

1. Verifying the individual adders;
2. Showing the individual adders are connected correctly;
3. The overall effect of the circuit is to perform multiplication.

The theory described earlier shows that the proof is equivalent to performing STE on the entire circuit. We gain significant performance improvements by using domain-knowledge (to show that two BDDs are the same without having to construct them), and composing a number of smaller STE results. The proof itself is a two page FL program which applies the proof rules described earlier: STE is used to prove local properties, and the composition rules show that that the parts are connected correctly and the overall result is multiplication. An outline of the proof follows, and an edited (for length and clarity – declarations are omitted) version of the proof appears in the appendix.

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. Secondly, once this check has been done, we compose this result with the result from the i-th stage. Specialising and time-shifting the new result[3] allows us to compose the transformed new result with the i-th stage result, yielding the $(i+1)$-st stage's output 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.

5.3 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.

[3] These transformations are found by the heuristics.

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 800 CPU seconds 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.

6 Conclusion

6.1 Summary

We have proposed a theorem-prover based on STE. This hybrid method is a powerful theory and tool for low-level hardware verification. 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.

We believe that we have shown that the use of hybrid methods give us flexibility and power without losing rigour. An important contribution is that domain knowledge can be incorporated — a very important factor in improving efficiency. From a usability and safety point of view it is important that a single, integrated system implement the hybrid method so that the user does not have to switch between systems or perform any translation.

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.

6.2 Problems and Extensions

This work is a prototype for future exploration. Besides issues of dealing with the rough edges, there are a number of problems and areas for future research.

As the use of domain knowledge is important, we need to see how this can be integrated in a cleaner fashion, and also extend the range and type of arguments we can make. We are also looking at enriching the logic in which we can express circuit properties. The importance of both of these points was illustrated in the verification of the Wallace-tree multiplier, where the proof would be made simpler and cleaner by such improvements.

One problem with theorem-provers is the "false implies everything problem". In our system, an inconsistent antecedent can be used to "prove" almost anything, leading to theorems which are mathematically valid but have no basis in

reality thereby lulling human users into a false sense of security. It would be good to provide automatic detection of such inconsistencies.

Systems which have little identifiable structure, or in which the interactions of the sub-parts of the system are very complex and fine-grained spatially and temporally will not be appropriate targets for this method.

Though the heuristics developed for the tool performed well, providing different and more powerful ones would improve the the system's usability.

STE applies to any type of system, but in practice it has only been applied to hardware systems. The use of abstraction and composition increases the range of systems for which STE will be tractable, and we are investigating how STE can be extended to more general systems.

References

1. R. E. Bryant. 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, Feb. 1991.
2. R. E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, Sept. 1992.
3. E. Clarke, D. Long, and K. McMillan. Compositional Model Checking. In *IEEE Fourth Annual Symposium on Logic in Computer Science*, Washington, D.C., 1989. IEEE Computer Society.
4. C. Courcoubetis, editor. *Proceedings of the 5th International Conference on Computer-Aided Verification*, Lecture Notes in Computer Science 697, Berlin, July 1993. Springer-Verlag.
5. S. Hazelhurst and C.-J. H. Seger. A Simple Theorem Prover Based on Symbolic Trajectory Evaluation and OBDDs. Technical Report 93-41, Department of Computer Science, University of British Columbia, Nov. 1993. Available by anonymous ftp as ftp.cs.ubc.ca:/pub/local/techreports/1993/TR-93-41.ps.gz.
6. H. Hungar. Combining Model Checking and Theorem Proving to Verify Parallel Processes. In Courcoubetis [4], pages 154–165.
7. J. J. Joyce and C.-J. H. Seger. Linking BDD-based Symbolic Evaluation to Interactive Theorem-Proving. In *Proceedings of the 30th Design Automation Conference*. IEEE Computer Society Press, June 1993.
8. R. Kurshan and L. Lamport. Verification of a Multiplier: 64 Bits and Beyond. In Courcoubetis [4], pages 166–179.
9. D. E. Long. *Model Checking, Abstraction, and Compositional Verification*. PhD thesis, Carnegie-Mellon University, School of Computer Science, July 1993. Technical report CMU-CS-93-178.
10. M. C. McFarland. Formal Verification of Sequential Hardware: A Tutorial. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(5):633–654, May 1993.
11. R. Milner. *Communication and Concurrency*. Prentice-Hall International, London, 1989.
12. C.-J. H. Seger and R. E. Bryant. Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories. Technical Report 93-8, Department of Computer Science, University of British Columbia, Apr. 1993.

13. C.-J. H. Seger and J. J. Joyce. A Mathematically Precise Two-Level Hardware Verification Methodology. Technical Report 92-34, Department of Computer Science, University of British Columbia, Dec. 1992.

A Proof of Multiplier

```
let Ainp="A";
let Bout= "B";
let Ground = "TC0";

// variable maps
....omitted for clarity
// Mathematical results
let theory = ...

// Preamble theorem-----------------------------------------------
let signal_length = 100000;

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

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

// Timing considerations -- ....
let answer_delay  = CPA_DELAY*bit_width;
let start_stage n = n*answer_delay;
let signal_len  n = signal_length-n*answer_delay;

let stage n  =
     let Cinp= "TC"^(num2str n) in
     let Cout= "TC"^(num2str (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  ('(n+1)) i) '* (POW2 ('n)) )))
                    FROM   answer_delay TO   signal_len n);

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

// Show partial result computed from the composition of stages 1..n is correct

let induc  n =
     let n1 = n+1 in
     let Cout= "TC"^(int2str 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 newthm2 = ALIGNSUB theory start (multhm n) in
      POSTWEAK theory  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 i sofar  =
         i=bit_width =>  POSTWEAK theory sofar postamble
                      |  do_proof (i+1) (inferencestep i sofar);

let  MultTheorem = do_proof 0 preambleThm;
```