ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

**Mary Sheeran, Chalmers Technical University, Sweden, and Satnam Singh,
University of Glasgow, UK (Eds)**

Designing Correct Circuits

Proceedings of the 3rd Workshop on Designing Correct Circuits
(DCC96), Båstad, Sweden, 2-4 September 1996

ISBN: 3-540-76102-0

Paper:

An Integrated Approach to Verifying Large Circuits: A Case Study

Scott Hazelhurst and Carl-Johan Seger

Published in collaboration with the
British Computer Society

An Integrated Approach to Verifying Large Circuits : A Case Study

S. Hazelhurst
Department of Computer Science
University of the Witwatersrand,
Johannesburg, South Africa
scott@cs.wits.ac.za

C.J.-H Seger
Department of Computer Science
University of British Columbia
Vancouver, B.C, Canada
seger@cs.ubc.ca

Abstract

Through the use of compositionality and abstraction, it is possible to extend automatic model checking techniques so that large circuits can be verified. This paper presents a case study verification of Benchmark 22 of the IFIP WG10.5 Benchmark Suite for Hardware Verification (a systolic array multiplier containing 115 000 gates). Both the timing and functionality of the circuit are verified (a significant error was discovered in the original benchmark). This illustrates that an appropriate logical framework can support an efficient, integrated tool for verification that incorporates a number of different verification techniques. A specialised theorem prover implements a compositional theory based on symbolic trajectory evaluation (STE). STE, an efficient model checking technique that can support large state spaces because of its natural and easily used method of abstraction, provides the underlying computational engine. The rest of the compositional theory allows a human verifier to use knowledge of the structure of the circuit to overcome some of the computational limitations of model checking. Using STE with its compositional theory, large circuits can be verified in detail using reasonable computational resources.

1 Introduction

Automatic model checking algorithms for verification are attractive because of their success in verifying many problems and because they are automatic. Unfortunately, the state explosion problem means that many real problems are out of the range of naïve approaches to model checking. This is particularly the case for circuits with relatively large datapaths, and where detailed analysis of timing may be necessary.

Considerable work has been done to extend model checking to support larger verification tasks. Some of these approaches are:

- *Abstraction*: Given a model to check, an abstract version of the model is constructed, and the abstract version of the model is checked. Typically, the abstract model is a *conservative abstraction* — false negatives may result, but not false positives [3].
- *Compositional model checking*: Using divide-and-conquer strategies to break the problem down into smaller problems and then combining smaller proofs into larger ones can be very successful. See for example [6, 11].
- *‘Hybrid’ approaches*: Theorem proving and automatic model checking algorithms can be combined so that these techniques complement each other. Some examples are [10, 12, 15].

How can a practical verification system exploit these ideas? There may be a danger of a Babel-like environment and approach to verification, leading to *ad hoc* solutions, which not only makes verification more difficult to do, but may unnecessarily compromise the soundness of the verification.

This paper is a case study of how automatic model checking, compositionality, abstraction, and theorem-proving can be combined into one integrated verification system that gives a verifier a consistent and flexible interface.

Section 2 briefly describes the theory underlying the approach. It presents a framework for model checking circuits with very large state spaces, briefly introduces symbolic trajectory evaluation (STE), the compositional theory for verification based on STE, and the practical tool used to accomplish verification. Section 3 introduces the methodology by giving an introductory outline of the verification of an integer multiplier (used in the major case study). The major part of this paper reports on the verification of the IFIP WG10.5 Benchmark 22 — a systolic array multiplier, which is approximately 115 000 gates in size. The circuit is verified completely, proving both functionality and timing properties. Section 4 discusses this verification, and the lessons learned in the verification.

2 Logical framework

This section presents the logical framework of our verification methodology. Section 2.1 presents symbolic trajectory evaluation (STE). Section 2.2 describes a compositional theory based on STE that can be used effectively to overcome computational limitations of fully-automatic methods. Section 2.3 discusses the practical aspects of a tool that implements the theory.

2.1 Symbolic trajectory evaluation

STE is an automatic model checking algorithm originally proposed by Bryant and Seger [2, 17], and later generalised and extended by us [7, 8]. The essential components of STE are described below.

- Circuits are represented by a model structure, comprised among other things by a state space $\mathcal{S}$ and next state function $\mathbf{Y}$ — this model structure is extracted automatically from the circuit description.
- Properties which the verifier wants to check are described by formulas of a temporal logic, TL.
- The semantics of TL are given with respect to sequences of states by a satisfaction relation — for each sequence (e.g. for each run of the system) the satisfaction relation says to what degree the sequence satisfies the formula.
- Verification conditions are expressed as $\models \langle g \Longrightarrow h \rangle$. The TL formula g , called the *antecedent*, can be thought of representing the input to the circuit, and the formula h , the *consequent*, can be thought of as the output of the circuit. $\models \langle g \Longrightarrow h \rangle$ holds if every trajectory (every run of the model) that satisfies the formula g also satisfies the formula h . These verification conditions are called *assertions*.
- STE is an efficient algorithm for verifying assertions.

For the purpose of this paper all that is important to note is that the method of model representation supports abstraction in a very natural way. Through the choice of antecedent, the verifier can very naturally affect the amount and level of information used by the STE algorithm. For each assertion, therefore, an abstraction of the circuit is implicitly created. Not only does this lead to a computationally effective approach, but since the construction of the abstraction is implicit, it does not place a burden on the human verifier.

STE supports a good model of time, which allows timing properties of circuits to be checked. We can check timing at different levels — for example, we can check that output is produced in the correct clock cycle, as well as checking timing within each cycle.

A full description of the syntax and semantics of TL is not necessary. The only syntax a reader should understand is formulas of the form $\text{Global} [(f_0, t_0), \dots, (f_n, t_n)] g$, which is a formula that asserts that the property g holds between times f_i and t_i (inclusive) for $i = 0, \dots, n$.

A full discussion of this is beyond the scope of this paper. The rest of this section briefly sketches some of the technical details, and more detail can be found in [7, 8, 17].

Technical details

Formally, the model is $(\langle \mathcal{S}, \sqsubseteq \rangle, \mathbf{Y})$ where $\mathcal{S}$ is a complete lattice under the partial order $\sqsubseteq$, and $\mathbf{Y} : \mathcal{S} \rightarrow \mathcal{S}$ is a monotonic function. In analysing the behaviour of such a model, we will be interested in sequences of states, in particular those sequences of states that are compatible with the next state function. Such sequences are called trajectories: if $\sigma = s_0 s_1 s_2 \dots$ is a sequence, it is a trajectory if for all i , $\mathbf{Y}(s_i) \sqsubseteq s_{i+1}$.

The partial order of the lattice is an information ordering: the higher up in the lattice the more that is known about the state of the system. We can think of a state as being an abstraction of states above it in the state space, so if s is less than s_1 and s_2 it contains the common features of s_1 and s_2 and suppresses their differences.

The computational advantage of this approach is that given the right logical framework, proving that a property holds of a state, s , implies that the property also holds of all states above s in the information order.

The truth of a TL formula is given by a satisfaction relation which relates TL formulas to sequences of states: $\text{Sat}(\sigma, g)$ indicates whether the sequence σ satisfies g . For both conceptual and technical reasons, a two-valued logic system is inadequate. Instead a four-valued system $\mathcal{Q} = \{\perp, \mathbf{f}, \mathbf{t}, \top\}$ (proposed by Belnap and elaborated by others [1, 4, 5]) is used for truth values. $\perp$ represents an unknown truth value, and $\top$ an inconsistent truth value. So, $\text{Sat}(\sigma, g) \in \mathcal{Q}$.

The partial order on states can be extended to sequences of states. One of the fundamental results of STE is that given a TL formula there is a set (often a very small set) of minimal sequences that characterise the formula. An assertion, $\models \langle g \Longrightarrow h \rangle$, is verified by computing the characterising sets for g and h , and then comparing these two sets.

2.2 Compositional Theory

STE has the property that the cost of verification is dependent more on the formula being checked than the size of the circuit. For this reason, there can often be a substantial computational benefit obtained by breaking the formula down into ‘smaller’ formulas, checking the smaller formulas, and then combining the smaller results. The compositional theory allows us to do that.

The compositional theory is a set of inference rules for inferring verification assertions. The idea is that the verification problem will be broken down into a number of smaller sub-problems. The individual problems will be verified with STE; these results will then be combined using the inference rules. Note that the circuit is not partitioned: while it is important to be able to identify some structure in the circuit for the compositional approach to be applicable, it is not necessary to decompose the circuit. The basic compositional rules are shown in Table 1; the side conditions are conditions that can be checked mechanically. A full description of the compositional theory is beyond the scope of the paper (see [7]). The example below illustrates a few of the important compositional rules.

Name	Rule	Side condition
Identity	$\frac{}{\models \langle g \Rightarrow g \rangle}$	
Time-shift	$\frac{\models \langle g \Rightarrow h \rangle}{\models \langle \text{Next}^t g \Rightarrow \text{Next}^t h \rangle}$	$t > 0$
Conjunction	$\frac{\models \langle g_1 \Rightarrow h_1 \rangle \quad \models \langle g_2 \Rightarrow h_2 \rangle}{\models \langle g_1 \wedge g_2 \Rightarrow h_1 \wedge h_2 \rangle}$	
Disjunction	$\frac{\models \langle g_1 \Rightarrow h_1 \rangle \quad \models \langle g_2 \Rightarrow h_2 \rangle}{\models \langle g_1 \vee g_2 \Rightarrow h_1 \vee h_2 \rangle}$	
Consequence	$\frac{\models \langle g \Rightarrow h \rangle}{\models \langle g_1 \Rightarrow h_1 \rangle}$	$\Delta^t(g) \sqsubseteq_{\mathcal{P}} \Delta^t(g_1), \Delta^t(h_1) \sqsubseteq_{\mathcal{P}} \Delta^t(h)$
Transitivity	$\frac{\models \langle g_1 \Rightarrow h_1 \rangle \quad \models \langle g_2 \Rightarrow h_2 \rangle}{\models \langle g_1 \Rightarrow h_2 \rangle}$	$\Delta^t(g_2) \sqsubseteq_{\mathcal{P}} \Delta^t(g_1) \sqcup \Delta^t(h_1)$
Specialisation	$\frac{\models \langle g \Rightarrow h \rangle}{\models \langle \Xi(g) \Rightarrow \Xi(h) \rangle}$	Ξ a specialisation.
Until	$\frac{\models \langle g_1 \Rightarrow h_1 \rangle \quad \models \langle g_2 \Rightarrow h_2 \rangle}{\models \langle g_1 \text{ Until } g_2 \Rightarrow h_1 \text{ Until } h_2 \rangle}$	

Table 1: Compositional Inference Rules

Example

In this section, we present some of the compositional rules in an example taken from the larger verification presented later. We have two cells (one labelled 0:0 and one 1:1), each of which contains a multiplier, an adder and some clock-driven latches. Figure 1 gives a pictorial description of one cell. For the sake of simplicity here, the example ignores clocking, which is integral to the proof done later.

These cells are part of Benchmark 22 in which sixteen of these cells are combined in a systolic array. The C_Out output of Cell 1:1 is connected to the C_In input of Cell 0:0, as shown in Figure 2 (i.e., C_Out₁₁ is an alias for C_In₀₀). The verification of the entire circuit starts by verifying each of these cells individually, and then continues by considering how the cells interact.

The correctness of the two cells can be summed up by the following two assertions. (The constants 0,

Figure 1: Cell from Benchmark 22

Figure 2: 2Syst circuit: cell connections

99, 200, 299 in the assertion are determined by the clock. The clock cycle is assumed to be 200ns, with the clock being low for the first 100ns and high for the next 100ns. Results 1 and 2 show that given certain inputs, the cell computes the right output.)

$$\begin{aligned} & \models \langle \text{Global}[(0, 99)] ([A_In_{00}] = a \wedge [B_In_{00}] = b \wedge [C_Out_{11}] = c) \\ & \Rightarrow \text{Global}[(200, 299)] ([C_Out_{00}] = c + ab) \rangle \end{aligned} \quad (1)$$

$$\begin{aligned} & \models \langle \text{Global}[(0, 99)] ([A_In_{11}] = a_1 \wedge [B_In_{11}] = b_1 \wedge [C_Out_{22}] = 0) \\ & \Rightarrow \text{Global}[(200, 299)] ([C_Out_{00}] = a_1 b_1) \rangle \end{aligned} \quad (2)$$

This lets us show how these two cells interact with each other. What we want to see is how in a given clock cycle, Cell 0:0 uses the output of Cell 1:1 in the previous cycle. The first step is to apply the Time-Shift Rule to Result 1 to get Result 3 thereby shifting the result forward by one clock cycle (200 ns).

$$\begin{aligned} & \models \langle \text{Global}[(200, 299)] ([A_In_{00}] = a \wedge [B_In_{00}] = b \wedge [C_Out_{11}] = c) \\ & \Rightarrow \text{Global}[(400, 499)] ([C_Out_{00}] = c + ab) \rangle \end{aligned} \quad (3)$$

In the second clock cycle, the value on C_Out_{11} is $a_1 b_1$, and the input on A_In_{00} and B_In_{00} are, for example, a_2 and b_2 . Now Result 3 is true for all possible inputs, so in particular it will be true for particular inputs at that time. So we can *specialise* Result 3 to get Result 4.

$$\begin{aligned} & \models \langle \text{Global}[(200, 299)] ([A_In_{00}] = a_2 \wedge [B_In_{00}] = b_2 \wedge [C_Out_{11}] = a_1 b_1) \\ & \Rightarrow \text{Global}[(400, 499)] ([C_Out_{00}] = a_1 b_1 + a_2 b_2) \rangle \end{aligned} \quad (4)$$

Now we take Result 2 and strengthen its antecedent using a rule of consequence to take into account the input of Cell 0:0 in the second cycle. We get:

$$\begin{aligned} & \models \langle \text{Global}[(0, 99)] ([A_In_{11}] = a_1 \wedge [B_In_{11}] = b_1 \wedge [C_Out_{22}] = 0) \wedge \\ & \quad \text{Global}[(200, 299)] ([A_In_{00}] = a_2 \wedge [B_In_{00}] = b_2) \\ & \Rightarrow \text{Global}[(200, 299)] ([C_Out_{00}] = a_1 b_1) \rangle \end{aligned} \quad (5)$$

The reason we do this is to combine Result 5 with Result 4. We can see that the antecedent of 4 is implied by the antecedent and consequent of 5. (Note that we could have proved Result 5 directly using STE, but this would have been computationally more intensive since we would use extra information.) In the same way that we can use transitivity in logic (e.g. $((a \Rightarrow b) \wedge (c \Rightarrow d) \wedge ((a \vee b) \Rightarrow c)) \Rightarrow (a \Rightarrow d)$), we can use the Transitivity Rule to combine 4 and 5 to get:

$$\begin{aligned} & \models \langle \text{Global}[(0, 99)] ([A_In_{11}] = a_1 \wedge [B_In_{11}] = b_1 \wedge [C_Out_{22}] = 0) \wedge \\ & \quad \text{Global}[(200, 299)] ([A_In_{00}] = a_2 \wedge [B_In_{00}] = b_2) \\ & \Rightarrow \quad \text{Global}[(400, 499)] ([C_Out_{00}] = a_1 b_1 + a_2 b_2) \rangle \end{aligned} \tag{6}$$

2.3 Tool

The prototype tools that we built to test the methodology described here used the Voss system [16] as its core. The three key features of Voss are:

- A symbolic simulator which provides good models of gate-level and switch-level circuits;
- An efficient implementation of ordered binary decision diagrams (BDDs);
- A lazy functional language called FL (a dialect of ML), which acts as the interface to the verification system.

Voss can perform trajectory evaluation on assertions with antecedents and consequents of a simple structure. We have extended Voss so that it can verify a richer set of consequents — in particular negation and disjunction are fully supported. The compositional theory has also been implemented as a simple specialised theorem prover, which provides a set of inference rules to the user. One of the inference rules performs STE, the others implement the compositional rules. The theorem prover also provides some automated assistance to the user to simplify the proof process, and the proof rules can be packaged in different ways. For example, a common pattern in a proof is to combine two results by specialising the one and then using transitivity. We can combine these two steps into one step, as well as use a heuristic to find the necessary specialisation automatically. The extended Voss system including these features is called the VossProver.

FL is very important as a flexible interface to the verification system, providing a good framework for us as tool builders to extend Voss, and a good way for a verifier to interact with the tool. It is good for verifiers because it provides a simple yet powerful means of using the verification system, while at the same time promoting the safety of results since each step of the proof is mechanically checked.

The proof of correctness is written as an FL program. The verifier can work completely interactively with Voss using FL as a command language, or run a batch file, or some mixture of the two. Typically, the verifier starts working interactively with Voss, building up an FL program as a proof script that is later run as a batch file.

One of the key factors making efficient verification possible is appropriate use of data structures. BDDs are critical in this, as STE relies on their efficient implementation. However, BDDs do have their limitations. We use a symbolic representation of data such as integers. When performing STE, data is (automatically) translated into appropriate BDD structures. However, the symbolic representation allows us to use other ways of reasoning about and manipulating data.

3 Example Verification: Benchmark 17, an integer multiplier

The major case study presented in this paper is a systolic matrix multiplier which contains, among other components, sixteen integer multipliers. Because the verification of a multiplier is in itself interesting and is an important part of the overall verification, the verification of the multiplier is outlined here. The purpose of this section is just to sketch how the verification takes place to give background for Section 4 which describes our methodology more fully.

The multiplication of two bit-vectors a and b is computed using the following formula:

$$a \times b = 2^0 ab[0] + 2^1 ab[1] + 2^2 ab[2] + \dots + 2^{n-1} ab[n-1]$$

where $b[j]$ is the j -th bit of b . Since BDD-representations of multiplication take exponential space, we cannot verify this using STE alone.

The multiplier circuit contains a number of stages, each of which basically computes a term in the above series and adds this term to the previous terms in the series. The $(j+1)$ -th stage takes as input the vector $a, b[j]$ (the j -th bit of b), and the answer produced by the first $j-1$ stages (i.e. the sum of the first $j-1$ terms in the series). It then computes $ab[j]$ using a number of adders, and then adds in the sum previously computed; this new result is then passed to the next stage as input. (This description is somewhat simplistic)

Ignoring timing for the moment, we use STE to prove that the first stage computes $ab[0]$. If we were to verify subsequent stages using the actual input of the previous stages, we would soon run into problems with BDD sizes, so we do things in two steps.

To verify the second stage, we show that if the second stage is given circuit inputs $a, b[1]$, and the number c from the first stage, then the second stage computes $c + 2^1 ab[1]$. Since for an arbitrary integer variable c , $c + 2^1 ab[1]$ has a compact BDD representation, this can be accomplished efficiently using STE alone. Once proved, the result is stored symbolically rather than using BDDs. This result shows what happens when an arbitrary input c is taken from the previous stage. The second step of the proof is to specialise the result – substitute $ab[0]$ for c to show that if the second stage has input $ab[0]$, then it has output $ab[0] + 2^1 ab[1]$. The VossProver can rewrite this as $ab[1-0]$, where $b[j-0]$ represents the $j+1$ low order bits of b .

Verification of stage 3 is similar.

- We show that if it has input c from stage 2, and external inputs a and $b[2]$, that it computes as output $c + 2^2 ab[2]$. This is efficiently proved using STE alone, and then the results are stored symbolically.
- We specialise the result to show that when given its actual input, stage 2 produces $ab[1-0] + 2^2 ab[2]$.
- This is rewritten by the VossProver, using its knowledge of integers, as $ab[2-0]$.

This process is repeated for each stage of the multiplier. The actual proof must take into account timing and is more complicated because the implementation uses redundant representation of data at most stages in the circuit in order to reduce the circuit costs.

The important parts to note about the proof are:

- STE is used at the lowest level of the circuit. This removes tedium from the verifier. In the verification of a particular stage, the choice of antecedent implicitly determines the level of abstraction of the circuit. BDDs are used for STE.
- A good model of time is provided.
- Results are stored symbolically – this means we can use the power of BDDs where useful, and use other reasoning methods in other cases. In the proof, the VossProver uses knowledge of integer arithmetic a number of times after specialisation has been used.

- Compositional rules are used to combine results proved using STE.
- Each step in the proof is sound, and mechanically checked. The VossProver can help in the proof by finding candidate specialisations automatically.
- A complete proof of one 32-bit multiplier takes about 170s on a DEC Alpha 3000.

Using this example, we can now summarise our methodology. STE is used to verify properties wherever possible to reduce the load on the human verifier. For each verification condition checked, an abstraction is automatically created, which makes the method tractable even for large circuits. And since the abstraction is implicitly created by the STE algorithm, no undue load is put on the human verifier. The human verifier then uses the specialised theorem prover that implements the compositional theory to combine ‘smaller’ verification conditions into ‘larger’ ones. Full details of verification of multipliers using this methodology can be found in [7, 9, 10].

4 Case Study : Systolic Array

This section describes how our methodology is used to verify a large circuit. We describe the specification and implementation of the circuit, and then describe the verification. Finally, we make some comments on the lessons of the verification.

4.1 Specification

A filter circuit, based on a design of Mead and Conway, is Benchmark 22 of the IFIP WG10.5 suite [13]. The filter is a matrix multiplication circuit for band matrices (matrices of band width w must have zeros in certain positions and a maximum of w non-zero integer entries in any row or column). This circuit is called 2Syst (a full description can be found using the URL <ftp://goethe.ira.uka.de/pub/benchmarks/2Syst/>). The suite documentation gives a specification of the circuit for multiplying 4×4 matrices of band width, $w = 4$. Let A , B and C be the 4×4 matrices given below:

$$A = \begin{bmatrix} a_{11} & a_{12} & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 \\ a_{31} & a_{32} & a_{33} & a_{34} \\ 0 & a_{42} & a_{43} & a_{44} \end{bmatrix}, \quad B = \begin{bmatrix} b_{11} & b_{12} & b_{13} & 0 \\ b_{21} & b_{22} & b_{23} & b_{24} \\ 0 & b_{32} & b_{33} & b_{34} \\ 0 & 0 & b_{43} & b_{44} \end{bmatrix}, \quad C = A \times B = \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{bmatrix}$$

The 2Syst circuit has four inputs for coefficients of the A matrix (labelled a_0 to a_3), four inputs for the coefficients of B (labelled $b_0, \dots, b_3$), seven outputs for the coefficients of C (labelled c_0 to c_6), and a clock input.

The timing of when and where the inputs must be applied and the outputs become available is critical. The essential of the circuit specifications is a table which shows when and where the inputs must be applied, and when and where the outputs may be found.

4.2 Implementation

Matrix multiplication ($C = A \times B$) is computed using the following formula:

$$\begin{aligned} c_{ij}^{(1)} &= 0 \\ c_{ij}^{(k+1)} &= c_{ij}^{(k)} + a_{ik} b_{kj} \\ c_{ij} &= c_{ij}^{(r+1)} \end{aligned} \tag{7}$$

The algorithm works in phases: $c_{ij}^{(k)}$ is the value of the i, j coefficient of the matrix in the k -th phase. The basic operation of Equation 7 is performing an addition and a multiplication; this is modelled in the implementation, where the basic cell has an integer multiplier and adder to perform this. This cell was shown in Figure 1. To summarise, the cell has three inputs: C_In is an m bit number, containing a partial sum; and A_In and B_In are n bit data which are either zero or coefficients of the A and B matrices. A_Out, B_Out are two n -bit output values and C_Out is an m -bit output. If in one clock cycle A_In, B_In and C_In have the values a , b and c respectively, then at the start of the next cycle: A_Out = a , B_Out = b , C_Out = $ab + c$.

Thus, the cell has two purposes: it acts as a one clock-cycle delay buffer for coefficients of the matrices (which are passed on to neighbouring cells), and performs the basic operation of an addition and multiplication. The multiplier is Benchmark 17 in the benchmark suite described previously, and the adder is a conventional $2n$ -bit adder. Each register has an input, an output, and a clock and select pin. By connecting the select and clock pins to the same global clock, the registers become positive-edge triggered: when the clock rises the value at the register's input is latched, output, and maintained until the clock rises again.

These cells are connected in a systolic array: in each clock cycle, cells perform an addition and a multiplication and then pass their results to their neighbours for use in the next cycle. The cells are arranged as presented in Figure 2. The timings in the specification are designed so that cells get the right inputs at the right time. To help the description, each cell in the systolic array has been labelled by $i : j$.

The circuit is implemented in Voss's EXE format as a detailed gate level description, using a unit delay model. The implementation is based on the VHDL program given in the benchmark suite documentation.

4.3 Verification

The verification of this circuit for moderate bit widths is well beyond the capability of any automatic model checking method. For example, most successful model checking techniques use BDDs. Since BDDs cannot represent multiplication efficiently, the verification of even one of the cells using model checking techniques alone is out of the question.

To use the methodology proposed here, the key step is to partition the problem into smaller problems. It is necessary to use knowledge of the structure of the circuit to do this (although it isn't necessary to partition the circuit itself). The idea is to break the problem down into the largest sub-problems that can be solved using STE alone, and then to combine the mini-solutions using the compositional rules.

Here, the structure of the proof is

- Verify each cell
 - Verify each multiplier
 - Verify the adder and latches of each cell
 - Show the cell as a whole works
- Show that with the given input, all the cells connected together produce the desired output.

What makes this approach effective is that we use theorem proving and STE at the right level using appropriate data structures. The theorem prover uses symbolic representation and domain knowledge in the application of the compositional rules to verify higher level properties of the circuit. STE can use the power of BDDs when performing low level verification. Moreover, each time trajectory evaluation is performed, only as much information as necessary is used. Through the choice of antecedent, the user implicitly and easily performs abstraction. The use of a tool with a flexible interface integrating both theorem proving and STE makes this all practical.

Here, the verification task can be divided into two parts, the verification of the individual components, and using the verification of the components to show that the whole array is correct.

4.3.1 Verifying the cells

The verification of a cell must show the multiplier, adder and registers all work correctly. Each cell must be verified individually. The verification of each cell itself is a challenging exercise since the cells perform multiplication, which means that we have to break the verification of each cell down into smaller problems.

This section describes the verification of Cell $u:v$, and assumes for the sake of this exposition that the clock cycle is 200ns, and the bit-width is 4. When the cells are connected together, port C_In_{uv} is connected to $C_Out_{(u+1)(v+1)}$, port A_Out_{uv} is connected to $A_In_{u(v+1)}$, and B_Out_{uv} is connected to $B_In_{(u+1)v}$.

It turns out that it is useful to divide this proof into three parts:

- Given value a on A_In_{uv} , b on B_In_{uv} , and c on $C_Out_{(u+1)(v+1)}$, one clock cycle later $ab + c$ appears on C_Out_{uv} ;
- Given value a on A_In_{uv} , one cycle later a appears on $A_In_{u(v+1)}$; and
- Given value b on B_In_{uv} , one cycle later b appears on $B_In_{(u+1)v}$.

Of course, it is possible to combine all three into a one, stronger result. However, having three weaker results makes the proof more flexible. The costliest part of the proof is to show the multiplier works correctly, the proof of which was sketched earlier — the verification must be done for each multiplier in each cell. When this is complete, we know that for each u and v that:

$$\begin{aligned} & \models \text{By various rules} \\ & \langle \text{Global} [(0, 100)] \ ([A_In_{uv}] = a \ \wedge \ [B_In_{uv}] = b) \\ & \implies \text{Global} [(22, 100)] \ ([O_{uv}] = ab) \rangle \end{aligned} \quad (8)$$

The antecedent says that the inputs of the cell are held stable for 100ns (the first half of the clock cycle), and that valid output appears 22ns from the start of the cycle (this is determined by the circuit and is obviously dependent on the bit width).

In the cell, the clock has an important effect; to include information of when clocking happens, the rule of consequence is often used to strengthen the antecedent of a result. For convenience, let

$$\begin{aligned} \text{Clock}_k & \stackrel{\text{def}}{=} \text{Global} [(200k, 200k + 99), (200(k + 1), 200(k + 1) + 99)] \ ([\text{clock}] = \mathbf{f}) \ \wedge \\ & \text{Global} [(200k + 100, 200k + 199)] \ ([\text{clock}] = \mathbf{t}) \end{aligned}$$

which is the information about clocking which is needed in the verifying the circuit in the k -th cycle. Recall that each cycle is 200ns long. This TL formula says that at the start of the k -th cycle (i.e. from time $200k$), the clock is low for 100ns, then high for 100ns, and then low for 100ns (at the start of the $k + 1$ -th cycle).

Using this idea, Result 8 is transformed strengthening the antecedent, as well as taking into account the input on C_In . Although this is not useful for its own sake, it is useful in combining with other results.

$$\begin{aligned} & \models \text{By Rule of consequence} \\ & \langle \text{Global} [(0, 100)] ([A_In_{uv}] = a \wedge [B_In_{uv}] = b \wedge [C_Out_{(u+1)(v+1)}] = c \wedge Clock_0) \\ & \implies \text{Global} [(22, 100)] ([0_{uv}] = ab) \rangle \end{aligned} \quad (9)$$

In the next step we show that the adder works correctly and that the output of the adder is latched for the appropriate time. This can be done with one trajectory evaluation. Note that the time interval in the consequent could be made bigger, but the one given suffices.

$$\begin{aligned} & \models \text{By STE} \\ & \langle \text{Global} [(22, 100)] ([0_{uv}] = d) \wedge [C_Out_{(u+1)(v+1)}] = c \wedge Clock_0) \\ & \implies \text{Global} [(200, 300)] ([C_Out_{uv}] = c + d) \rangle \end{aligned} \quad (10)$$

Results 9 and 10 are now combined by specialising the latter result (substituting ab for d), and using transitivity. Note that this is just a special case of transitivity.

$$\begin{aligned} & \models \text{By Transitivity} \\ & \langle \text{Global} [(0, 100)] ([A_In_{uv}] = a \wedge [B_In_{uv}] = b \wedge [C_Out_{(u+1)(v+1)}] = c \wedge Clock_0) \\ & \implies \text{Global} [(200, 300)] ([C_Out_{uv}] = c + ab) \rangle \end{aligned} \quad (11)$$

Result 11 is the core result proved about the cell. The next two results show that the cell also acts as one cycle delay buffers for values of the A and B matrices. Both of these results can easily be done using STE alone.

$$\begin{aligned} & \models \text{By STE} \\ & \langle \text{Global} [(0, 100)] ([A_In_{uv}] = a \wedge Clock_0) \\ & \implies \text{Global} [(200, 300)] ([A_In_{u(v+1)}] = a) \rangle \end{aligned} \quad (12)$$

$$\begin{aligned} & \models \text{By STE} \\ & \langle \text{Global} [(0, 100)] ([B_In_{uv}] = b \wedge Clock_0) \\ & \implies \text{Global} [(200, 300)] ([B_In_{(u+1)v}] = b) \rangle \end{aligned} \quad (13)$$

4.3.2 Overall verification

Once each of the cells has been individually verified, the proofs about the individual cells must be combined to prove that the systolic array as a whole works correctly. The proof is modelled on how the systolic array computes its results; in its development the proof traces the behaviour of the circuit as it uses its inputs, computes results, and outputs the answers.

Consider the operation of one cell, At the beginning of clock cycle k , none, some, or all of the following will be known about Cell $u:v$'s input neighbours, where the I_j are some antecedent TL formulas that describe

the inputs to the circuit, and the θ_x are some integer expressions:

$$\models \langle I_1 \Rightarrow \text{Global}[(200k, 200k + 100)] \text{ [A_In}_{uv}] = \theta_a \rangle \quad (14)$$

$$\models \langle I_2 \Rightarrow \text{Global}[(200k, 200k + 100)] \text{ [B_In}_{uv}] = \theta_b \rangle \quad (15)$$

$$\models \langle I_1 \Rightarrow \text{Global}[(200k, 200k + 100)] \text{ [C_Out}_{(u+1)(v+1)}] = \theta_c \rangle \quad (16)$$

If all three results are known, then we use conjunction on Results 14–16, and introduce new clocking information. For convenience, let $I_4 = I_1 \wedge I_2 \wedge I_3 \wedge \text{Clock}_k$. This is the conjunction of I_1 , I_2 and I_3 and contains necessary clocking information for the k -th cycle. Then we have:

$$\begin{aligned} & \models \text{By Conjunction and Rule of Consequence} \\ & \langle I_4 \\ & \Rightarrow \text{Global}[(200k, 200k + 100)] \\ & \quad \text{[A_In}_{uv}] = \theta_a \wedge \text{[B_In}_{uv}] = \theta_b \wedge \text{[C_Out}_{uv}] = \theta_c \rangle \end{aligned} \quad (17)$$

Then Result 11 is time-shifted forward by $k - 1$ clock cycles to get:

$$\begin{aligned} & \models \text{By time-shifting} \\ & \langle \text{Global}[(200k, 200k + 100)] \\ & \quad (\text{[A_In}_{uv}] = a \wedge \text{[B_In}_{uv}] = b \wedge \text{[C_Out}_{(u+1)(v+1)}] = c \wedge \text{Clock}_k) \\ & \Rightarrow \text{Global}[(200(k + 1), 200(k + 1) + 100)] \text{ ([C_Out}_{uv}] = c + ab) \rangle \end{aligned} \quad (18)$$

Using specialisation and transitivity on Results 17 and 18 leads to:

$$\begin{aligned} & \models \text{By transitivity} \\ & \langle I_4 \\ & \Rightarrow \text{Global}[(200(k + 1), 200(k + 1) + 100)] \text{ ([C_Out}_{uv}] = \theta_c + \theta_a \theta_b) \rangle \end{aligned} \quad (19)$$

This is a proof of what Cell $u:v$ computes in the k -th cycle. In proving what happens in the $(k + 1)$ -th cycle, Result 19 is used in the proof of the behaviour of Cell $(u + 1):(v + 1)$.

Similarly, if Result 14 is known, then precondition strengthening is used to introduce new clocking information to get:

$$\begin{aligned} & \models \text{By Rule of consequence} \\ & \langle I_1 \wedge \text{Clock}_k \\ & \Rightarrow \text{Global}[(200k, 200k + 100)] \text{ [A_In}_{uv}] = \theta_a \rangle \end{aligned} \quad (20)$$

Then Result 12 is time-shifted by k clock cycles to get:

$$\begin{aligned} & \models \text{By STE} \\ & \langle \text{Global}[(200k, 200k + 100)] \text{ ([A_In}_{uv}] = a) \wedge \text{Clock}_k \\ & \Rightarrow \text{Global}[(200(k + 1), 200(k + 1) + 100)] \text{ ([A_In}_{u(v+1)}] = a) \rangle \end{aligned} \quad (21)$$

Transitivity/specialisation between Results 20 and 21 then yields:

$$\begin{aligned} & \models \text{By transitivity} \\ & \langle I_1 \wedge \text{Clock}_k \\ & \Rightarrow \text{Global}[(200(k + 1), 200(k + 1) + 100)] \text{ ([A_In}_{u(v+1)}] = \theta_a) \rangle \end{aligned} \quad (22)$$

This shows what Cell $u:v$ passes to one of its neighbours at the end of the k -th cycle, and this result will be used to prove properties of Cell $u:(v+1)$ in the $(k+1)$ -th cycle. A similar result shows that in the k -th Cell $u:v$ also passes on the value input on its B_In port,

$$\begin{aligned} & \models \text{By various rules} \\ & \langle I_2 \wedge \text{Clock}_k \\ & \implies \text{Global } [(200(k+1), 200(k+1) + 100] \text{ } ([\text{B_In}_{(u+1)v}] = \theta_b) \rangle \end{aligned} \quad (23)$$

4.4 Analysis and comments

The FL proof script which performs the proof uses the approach outlined above. First, the behaviour of each cell is individually verified. Then, the proof proceeds by proving properties of the circuit in each clock cycle.

A two dimensional array of proofs is kept: at the start of the k -th cycle, the array's (u, v) entry contains proofs of what the output of Cell $u:v$ input neighbours are at the end of the $(k-1)$ -th cycle. The proof then uses this information to infer as much as possible about the output of Cell $u:v$ at the end of the k -th cycle, and this information is then used to update the array of proofs so that Cell $u:v$'s output neighbours can use this information in the $(k+1)$ -th cycle.

The FL proof script uses STE and the inference rules to prove what the output of the circuit is at different stages. The first time the verification was attempted, a significant error was discovered in the benchmark (the specification gave wrong cycles for many outputs), which has subsequently been corrected as a result of this work. The proof script, including the proof of the correctness of all the multipliers and declarations, is approximately 650 lines long. The program itself is straightforward, although the use of a two dimensional array does not show off a functional, interpreted language at its best. The complete verification of a 4×4 systolic array of 32 bit multipliers takes just under three hours on a DEC Alpha 3000 with 512M of memory (11 hours on a Sun 10/51 with 64M of memory).

The verification is of a detailed gate-level description of a particular circuit. Unlike other approaches (e.g. [14]) we must verify each multiplier of a given bit-width individually. Nor do we exploit the regularity of design in order to reduce computational cost (i.e. we verify each multiplier in the systolic array). Clearly, there are computational disadvantages to this as we cannot just verify one parameterised design. On the other hand, this approach means that we can deal with timing easily; this is important since the timing characteristics of a circuit can be very sensitive to the bit-width. Moreover, from a proof development point of view, this is not a problem since proofs can easily be reused (the proof script for a 4 bit multiplier and a 64 bit multiplier differs in one line only). The incorporation of inductive methods is certainly possible.

As this verification was done at the same time as circuit implementation and system development, it is difficult to estimate how much human time it took — our very rough estimate is that it took a week of work. In order to use our system effectively, it is important to be a competent FL programmer, and this will be the most important part of the learning curve for an engineer. The theorem prover itself is a fairly simple with only a limited number of proof rules. It is also possible to use Voss in a graduated way, starting off using it as a symbolic simulator, then moving to verification using symbolic trajectory evaluation, and then starting to use the compositional theory in increasingly sophisticated ways.

5 Conclusion

The verification presented here shows a number of things. First, it provides further evidence that STE is a suitable basis for verification of very large state spaces, since abstraction is naturally supported by the lattice-structure of state space. This means that STE can be applied directly to models with large state spaces.

This paper has shown that using compositional model checking and abstraction, implemented in an integrated tool providing both STE and theorem proving, is a practical verification technique for large circuits. In the case study shown, a circuit containing 115 000 gates was verified, with reasonable computational and human cost. Importantly, both functionality and timing properties are proved — an approach that ignored timing issues would not have detected the serious problem in the original Benchmark documentation. We have applied this methodology to other case studies to confirm the method's general applicability.

Contact Addresses and Acknowledgements: Scott Hazelhurst is now at the Department of Computer Science, University of the Witwatersrand, Johannesburg, 2050 Wits, South Africa; email scott@cs.wits.ac.za. Carl Seger is now at the Intel Development Labs, JFT-102, Hillsboro, Oregon 97124-5961, United States; email cseger@ichips.intel.com.

Much of this work was done while the authors were in the Department of Computer Science at the University of British Columbia, Vancouver, Canada. This research was supported by operating grant OGPO 109688 from the Natural Sciences and Engineering Research Council of Canada, a fellowship from the B.C. Advanced Systems Institute, and by equipment grants from Sun Microsystems Canada Ltd. and Digital Equipment of Canada Ltd, and the University of the Witwatersrand Research Committee. We thank Thomas Kropf for his help with the benchmark suite, and Mark Greenstreet for his comments, and the ISD Lab at UBC for its support.

References

- [1] N.D. Belnap. A useful four-valued logic. In J.M. Dunn and G. Epstein, editors, *Modern Uses of Multiple Valued Logic*. D. Reidel, Dordrecht, 1977.
- [2] R.E. Bryant, D.L. Beatty, and C.-J. H. Seger. Formal Hardware Verification by Symbolic Ternary Trajectory Evaluation. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pages 397–407. 1991.
- [3] E.M. Clarke, O. Grumberg, and D.E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5), September 1994.
- [4] M. Fitting. Bilattices and the Theory of Truth. *Journal of Philosophical Logic*, 18(3):225–256, August 1989.
- [5] M.L. Ginsberg. Bilattices and modal operators. *Journal of Logic and Computation*, 1(1):41–69, 1990.
- [6] O. Grumberg and D.E. Long. Model Checking and Modular Verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.
- [7] S. Hazelhurst. *Compositional Model Checking of Partially-Ordered State Spaces*. PhD thesis, University of British Columbia, Department of Computer Science, 1996.

- [8] S. Hazelhurst and C.-J. H. Seger. Model Checking Partially Ordered State Spaces. Technical Report 95-18, Department of Computer Science, University of British Columbia, July 1995. Available by anonymous ftp as <ftp://ftp.cs.ubc.ca/pub/local/techreports/1995/TR-95-18.ps.gz>.
- [9] S. Hazelhurst and C.-J. H. Seger. Verification of Benchmarks 17 and 22 of the IFIP WG10.5 Benchmark Circuit Suite. Technical Report 95-21, Department of Computer Science, University of British Columbia, October 1995. Available by anonymous ftp as <ftp://ftp.cs.ubc.ca/pub/local/techreports/1995/TR-95-21.ps.gz>.
- [10] S. Hazelhurst and C.-J.H. Seger. A Simple Theorem Prover Based on Symbolic Trajectory Evaluation and BDD's. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(4):413–422, April 1995.
- [11] H. Hungar. Combining Model Checking and Theorem Proving to Verify Parallel Processes. In C. Courcoubetis, editor, *Proceedings of the 5th International Conference on Computer-Aided Verification*, Lecture Notes in Computer Science 697, pages 154–165, Berlin, July 1993. Springer-Verlag.
- [12] 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.
- [13] T. Kropf. Benchmark-Circuits for Hardware-Verification. In R. Kumar and T. Kropf, editors, *TPCD'94: Proceedings of the Second International Conference on Theorem Provers in Circuit Design*, Lecture Notes in Computer Science 901, pages 1–12, Berlin, September 1994. Springer-Verlag.
- [14] L. Pierre. VHDL Description and Formal Verification of Systolic Multipliers. In D. Agnew, L. Claesen, and R. Camposano, editors, *Computer hardware description languages and their applications : proceedings of the 11th IFIP WG10.2 International Conference on Computer Hardware Description Languages and their Applications - CHDL'93*, number 32 in IFIP Transactions A, pages 225–242, Berlin, 1993. Springer-Verlag.
- [15] S. Rajan, N. Shankar, and M.K. Srivas. An Integration of Model Checking with Automated Proof Checking. In P. Wolper, editor, *CAV '95: Proceedings of the Seventh International Conference on Computer Aided Verification*, Lecture Notes in Computer Science 939, pages 84–97, Berlin, July 1995. Springer-Verlag.
- [16] C.-J.H. Seger. Voss — A Formal Hardware Verification System User's Guide. Technical Report 93-45, Department of Computer Science, University of British Columbia, November 1993. Available by anonymous ftp as <ftp://ftp.cs.ubc.ca/pub/local/techreports/1993/TR-93-45.ps.gz>.
- [17] C.-J.H. Seger and R.E. Bryant. Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories. *Formal Methods in Systems Design*, 6:147–189, March 1995.
- [18] J. Staunstrup and T. Kropf. IFIP WG10.2 Benchmark Circuits for Hardware Verification. URL: <http://goethe.ira.uka.de/benchmarks/>.