Verification of a Multiplier: 64 Bits and Beyond

R. P. Kurshan
AT&T Bell Labs
Murray Hill, NJ 07974
k@research.att.com

Leslie Lamport
Digital Equipment Corporation
130 Lytton Avenue
Palo Alto, CA 94301
lamport@src.dec.com

14 April 1993

To appear in *Proceedings of the Fifth International Workshop on Computer-Aided Verification*

Verification of a Multiplier: 64 Bits and Beyond

R. P. Kurshan[1] and Leslie Lamport[2]

[1] AT&T Bell Labs
Murray Hill, NJ 07974
k@research.att.com
[2] Digital Equipment Corporation
Palo Alto, CA 94301

Abstract. Verifying a 64-bit multiplier has a computational complexity that puts it beyond the grasp of current finite-state algorithms, including those based upon homomorphic reduction, the induction principle, and bdd fixed-point algorithms. Theorem proving, while not bound by the same computational constraints, may not be feasible for routinely coping with the complex, low-level details of a real multiplier. We show how to verify such a multiplier by applying COSPAN, a model-checking algorithm, to verify local properties of the complex low-level circuit, and using TLP, a theorem prover based on the Temporal Logic of Actions, to prove that these properties imply the correctness of the multiplier. Both verification steps are automated, and we plan to mechanize the translation between the languages of TLP and COSPAN.

1 Introduction

For finite-state systems, it is in principle possible to use model checking to verify properties of a system automatically, with little human intervention. However, computational complexity limits the applicability of such methods. Verifying a 64-bit multiplier is beyond the capability of existing model checkers, even with indirect methods such as homomorphic reduction [8, 9], structural induction [10], and fixed-point algorithms using binary decision diagrams [2].

Mechanical theorem proving provides an alternative to automatic model checking. However, it is hard work. Proving that a system satisfies even a fairly simple property can be painful. Although progress is being made, and there have been some impressive verifications using theorem provers [3], it is unclear how soon theorem proving will be feasible for the routine verification of complicated systems. In any case, complementing a theorem prover with a model checker that, when feasible, verifies proof obligations automatically will surely save work.

We show how to combine theorem proving and model checking to mechanically verify systems that are more difficult or infeasible to verify by either method alone. Our approach applies to systems with a relatively small number of different high-level components. Components may be replicated without limit, if they are interconnected in a fairly regular fashion, and may have an arbitrarily complex low-level structure. Model checking is used to verify the individual components, and theorem proving is used to show that the complete system satisfies its specification if each component does.

As an example, we verify a $k \cdot 2^m$-bit multiplier, constructed from k-bit multipliers by recursively applying a method for implementing a $2N$-bit multiplier with four N-bit multipliers. The k-bit multiplier could implement a complex algorithm such as a radix-4 modified version of Booth's algorithm [7]. We could choose k as large as 8. The 8-bit multiplier is small enough to be verified by model checking, but complicated enough to make its verification with theorem proving very difficult. For $k \cdot 2^m$ equal to 64, the complete multiplier is too complex to be verified entirely by model checking. Abstracting components (homomorphic reduction [8]) cannot help, because it cannot reduce the complexity below the size of the combined inputs, which exceeds the limit of tractability for a multiplier.

In this simple example, we have verified not just a 64-bit multiplier, but an $8 \cdot 2^m$-bit multiplier for all values of m. Real multipliers are not constructed by such a simple recursive procedure. However, we expect our approach of combining model checking of local properties with theorem proving to work for real multipliers.

We combine two existing tools: the TLP theorem prover [4], which verifies models written in the Temporal Logic of Actions (TLA) [12], and the automata-theoretic model checker COSPAN [5], which verifies models written in the language S/R [6]. We chose to combine TLA and S/R because they are simple and have similar semantic bases. Moreover, a tool exists for the automatic synthesis to hardware of an S/R specification. In principle, we could write the models in either language and translate to the other. We translate from TLA to S/R both because TLA is more concise, and because theorem proving requires understanding what one is proving, which is easier if the theorem to be proved was written by a human rather than generated by a translator.

Four models are used: $M(N)$, an abstract N-bit multiplier, which essentially asserts that its output is the product of its inputs; $E(N)$, the multiplier's environment; $DM(N)$, a circuit that combines four N-bit multipliers to implement a $2N$-bit multiplier; and $B(k)$, the implementation of the k-bit multiplier. Although one can write $B(k)$ in TLA and translate it to S/R, this is not necessary because all verification involving $B(k)$ is done with COSPAN, so its TLA version is not needed.

2 Decomposition in TLA

We begin with a brief description of TLA; a more complete exposition appears in [12]. We assume an infinite collection of variables and a suitably large collection of values. A *state* is an assignment of values to variables. A *state function* is an ordinary expression, such as $x + y + 1$, built from values and variables, and a *predicate* is a Boolean-valued state function, such as $x > y + 1$. An *action* is a Boolean-valued expression, such as $x' > y + 1$, containing primed and unprimed variables. A pair $\langle s, t \rangle$ of states satisfies action $\mathcal{A}$ iff (if and only if) $\mathcal{A}$ is true when unprimed and primed variables are replaced by their values in s and t. (We let $\langle \ldots \rangle$ denote a sequence or tuple.) An action represents allowed state transitions—for example, $x' = x + 1$ allows any transition in which the value of x is incremented by 1. We let v' denote the expression obtained by priming all the variables in the state function v—for example $(x + y + 1)'$ equals $x' + y' + 1$. An action $\mathcal{A}$ is *enabled* in a state s iff there is some state t such that $\langle s, t \rangle$ satisfies $\mathcal{A}$.

A *behavior* is an infinite sequence of states. A TLA formula is true or false for a behavior. We write $\models F$ to denote that F is true for all behaviors. A TLA formula S specifies a system whose correct executions are represented by the behaviors satisfying S. A system with TLA specification S therefore satisfies a property P iff $\models S \Rightarrow P$.

The basic class of TLA formulas we will use are of the form $I \wedge \Box[\mathcal{N}]_v \wedge L$, where I is an "initial state" predicate, $\mathcal{N}$ is an action, v is a state function, $[\mathcal{N}]_v$ denotes $\mathcal{N} \vee (v' = v)$, and L is a conjunction of formulas of the form $\mathrm{WF}_v(\mathcal{A})$, defined below. A behavior satisfies this formula iff the initial state satisfies I, every successive pair of states satisfies $[\mathcal{N}]_v$ (representing either an $\mathcal{N}$ transition or one that leaves v unchanged—a "stuttering" step), and the behavior satisfies L. A behavior satisfies $\mathrm{WF}_v(\mathcal{A})$ iff the action $\mathcal{A} \wedge (v' \neq v)$ is satisfied by infinitely many pairs of successive states, or is disabled in infinitely many states of the behavior.

Another class of TLA formulas we use have the form $\Box P$, where P is a predicate. Such a formula is true for a behavior iff P is true for every state of the behavior. We also use the TLA operator $\boldsymbol{\exists}$, where $\boldsymbol{\exists}\, x : F$ essentially denotes the formula F with x "hidden". The precise definition of $\boldsymbol{\exists}$ can be found in [12].

A finite sequence ρ of states is said to satisfy a TLA formula F iff ρ is the prefix of a behavior that satisfies F.[3] A formula F is said to be a *safety property* iff the following condition is satisfied: if every finite prefix of σ satisfies F, then σ satisfies F. The closure $\mathcal{C}(F)$ of a formula F is the strongest safety property (conjunction of all such properties) implied by F. If L is the conjunction of formulas of the form $\mathrm{WF}_v(\mathcal{A})$ where each $\mathcal{A}$ implies $\mathcal{N}$, then

$$\mathcal{C}(I \wedge \Box[\mathcal{N}]_v \wedge L) \;\equiv\; I \wedge \Box[\mathcal{N}]_v \tag{1}$$

In particular, $I \wedge \Box[\mathcal{N}]_v$ is a safety property. For any predicate P, the formula $\Box P$ is also a safety property.

For any safety property E, we define $E^{\oplus}$ to be the property that is true "one step longer than" E is. In other words, a finite sequence of states ρ of length n satisfies $E^{\oplus}$ iff either $n = 1$, or $n > 1$ and the prefix of ρ of length $n - 1$ satisfies E. An infinite behavior satisfies $E^{\oplus}$ iff it satisfies E or has the form $\langle s_0, \ldots, s_n, s_n, s_n, \ldots \rangle$, where $n = 0$ or $\langle s_0, \ldots, s_{n-1} \rangle$ satisfies E. In general, $E^{\oplus}$ is not expressible with the TLA operators $'$, $\Box$, and $\boldsymbol{\exists}$. For example, $\mathit{false}^{\oplus}$ is the conjunction of the formulas $\Box[\mathit{false}]_x$ for all variables x, and such an infinite

[3] Since TLA formulas are invariant under stuttering [11], the definition of safety that follows would be the same had we defined ρ to satisfy F iff the behavior obtained by repeating the last state of ρ satisfies F.

conjunction cannot be expressed in TLA. However, formulas of the form $E^{\oplus}$ occur only in hypotheses of the form $\models E^{\oplus} \wedge P \Rightarrow Q$, which can be verified by substituting for $E^{\oplus}$ a "conservative approximation" $\widetilde{E}$ satisfying $\models E^{\oplus} \Rightarrow \widetilde{E}$. A suitable approximation is obtained by applying

$$\models (I \;\wedge\; \Box[\mathcal{N}]_v)^{\oplus} \quad\Rightarrow\quad \exists\, s : \widetilde{I} \;\wedge\; \Box[\widetilde{\mathcal{N}}]_{\langle s,\, v\rangle} \tag{2}$$

where s is a variable that does not occur in I, $\mathcal{N}$, or v, and

$$\begin{aligned} \widetilde{I} \;&\triangleq\; (I \wedge (s = 0)) \vee (\neg I \wedge (s = 1)) \\ \widetilde{\mathcal{N}} \;&\triangleq\; (s = 0) \;\wedge\; (\,(\mathcal{N} \wedge (s' = s)) \vee (\neg\mathcal{N} \wedge (s' = 1))\,) \end{aligned}$$

(The symbol $\triangleq$ means *equals by definition*.)

The key to verifying a complex system is decomposing the proof. In TLA, a system is decomposed into components by writing its specification as the conjunction of its components' specifications. We represent a system composed of n components by a TLA formula $E \wedge B_1 \wedge \ldots \wedge B_n$, where E is the environment specification and B_i is the specification of the ith component. We consider only the case of E a safety property. Our problem is to prove that, in the presence of a properly functioning environment, the system satisfies some property F. This requires proving $\models E \wedge B_1 \wedge \ldots \wedge B_n \;\Rightarrow\; F$. We do this by writing a high-level specification M_i for each component i, and proving[4]

$$\models E \wedge M_1 \wedge \ldots \wedge M_n \;\Rightarrow\; F \tag{3}$$

$$\models E \wedge B_1 \wedge \ldots \wedge B_n \;\Rightarrow\; E \wedge M_1 \wedge \ldots \wedge M_n \tag{4}$$

We prove (3) by standard TLA reasoning—that is, by theorem proving. To prove (4), we use the following theorem, which is proved in [1].

Decomposition Theorem *If E is a safety property and, for $i = 1, \ldots, n$,*

1. *E_i is a safety property.*
2. *$\models E \wedge \mathcal{C}(M_1) \wedge \ldots \wedge \mathcal{C}(M_n) \;\Rightarrow\; E_i$*
3. *(a) $\models E_i \wedge B_i \;\Rightarrow\; M_i$*
   (b) $\models E_i^{\oplus} \wedge \mathcal{C}(B_i) \;\Rightarrow\; \mathcal{C}(M_i)$
4. *$\models \mathcal{C}(M_1 \wedge \ldots \wedge M_n) \;\equiv\; \mathcal{C}(M_1) \wedge \ldots \wedge \mathcal{C}(M_n)$*

then (a) $\models E \wedge B_1 \wedge \ldots \wedge B_n \;\Rightarrow\; M_1 \wedge \ldots \wedge M_n$, and
(b) $\models E^{\oplus} \wedge \mathcal{C}(B_1 \wedge \ldots \wedge B_n) \;\Rightarrow\; \mathcal{C}(M_1 \wedge \ldots \wedge M_n)$.

Conclusion (a) provides the desired result (4). Conclusions (a) and (b) have the same form as hypotheses 3(a) and 3(b); as we shall see, this permits recursive application of the theorem.

To apply the Decomposition Theorem, we choose E_i to be an abstract specification of the ith component's environment. The second hypothesis asserts that

[4] In general, one might want to replace E by a more abstract environment specification in (3) and in the conclusion of (4). This generalization is not needed in our example.

E_i is implemented by the composition of the safety properties of the high-level specifications M_i and their environment. The third hypothesis essentially asserts that the composition of the low-level specification and its environment implements the high-level specification. The fourth hypothesis is a technical requirement; conclusion (a) remains valid even if this hypothesis is not satisfied.

In a simple application of the theorem, the three hypotheses are proved as follows. The first hypothesis will follow immediately from (1) and the form of E. The second hypothesis will be proved by standard TLA reasoning—that is, by theorem proving, using TLP. The third hypothesis will be proved by model checking, using COSPAN. The fourth hypothesis is checked with (1). Because our multiplier is defined recursively, it will be verified by recursive application of the Decomposition Theorem. As explained in Section 5 below, model checking is needed only for the base-case application of the theorem.

Theorem proving is applied only to the specifications E, E_i and M_i. The specifications E and E_i are abstractions of the environment, describing only as much of the environment's behavior as is necessary to ensure correct operation of the system or the individual component. The specification M_i is a high-level abstraction. Hence, we are proving theorems only about relatively simple specifications. All reasoning about B_i, the detailed specification of the component, is done with model checking. Thus, we need not even write the TLA formula B_i. We translate the TLA formulas E_i, M_i, $\mathcal{C}(M_i)$, and the approximation to $E_i^{\oplus}$ into S/R, the language of the COSPAN model checker, and use COSPAN to verify the S/R formulas corresponding to the TLA formulas of hypothesis 3.

Note that the Decomposition Theorem would be false if $E_i^{\oplus}$ were replaced by E_i in hypothesis 3(b). For example, with this change, the hypotheses would be satisfied by letting all the E_i and M_i be false, and conclusion (a) would assert $\models \neg(E \wedge B_1 \wedge \ldots \wedge B_n)$ for arbitrary E and B_i. It is trivial to infer the refinement relation (4) from $\models B_i \Rightarrow M_i$ for each i. However, in systems such as the multiplier, which have a high degree of global coordination among components, $\models B_i \Rightarrow M_i$ is almost never true; in these systems, one can generally prove little about the behavior of a component B_i without knowing something about the behavior of its environment E_i.

3 The Specification of the Multiplier

We write the specification in a formal language based on TLA containing operators for defining data structures; language structures for making declarations, definitions, and assumptions; and a module mechanism for encapsulating names and performing renaming. This language, which can be translated into S/R, is a subset of a more general specification language under development called TLA$^+$ [13].

We assume some conventional mathematical notation for numbers and sets. We write $f[x]$ for the application of a function f to a value x in its domain, and we let $[S \rightarrow T]$ denote the set of all functions with domain S and range a subset of T. Functions are explicitly defined with expressions of the form $[x \in S \mapsto e(x)]$, which denotes the function f with domain S such that $f[x] = e(x)$ for all $x \in S$.

We define $\mathit{BitVector}(i)$ to be the set of all i-bit-wide bit vectors and $\mathit{Val}(i, v)$ to be the value of the i-bit vector v interpreted as a binary number. They are defined formally as follows.

$$\begin{aligned} \mathit{BitVector}(i) &\triangleq [\{0, \ldots, i-1\} \rightarrow \{0, 1\}] \\ \mathit{Val}(i, v) &\triangleq \sum_{j=0}^{i-1} v[j] \cdot 2^j \end{aligned}$$

We let $\mathit{Multiply}(i, v, w)$ denote the $2i$-bit vector obtained by multiplying the i-bit vectors v and w. Formally, $\mathit{Multiply}$ is defined by the following axiom, where Nat denotes the set of natural numbers. (A list bulleted by $\wedge$ denotes the conjunction of the items.)

$$\begin{array}{l} \forall\, i \in \mathit{Nat} : \forall\, v, w \in \mathit{Bitvector}(i) : \\ \quad \wedge\ \mathit{Multiply}(i, v, w) \in \mathit{BitVector}(2i) \\ \quad \wedge\ \mathit{Val}(2i, \mathit{Multiply}(i, v, w)) = \mathit{Val}(i, v) \cdot \mathit{Val}(i, w) \end{array}$$

We specify an asynchronous multiplier circuit with inputs a and x and output out that synchronizes with its environment using a two-phase handshaking protocol on the bits sig and ack. The environment can change the inputs when $\mathit{sig} = \mathit{ack}$; it complements sig when the inputs are ready. The multiplier can change the output when $\mathit{sig} \neq \mathit{ack}$; it complements ack when the output is ready. Initially, sig and ack equal 0 and the multiplier is ready to receive input. The TLA formulas M specifying the multiplier and E specifying its environment are defined in the Mult module of Figure 1. The **parameters** section declares all the variables and unspecified constants that may appear in the module. The declaration of N asserts the assumption that N is an element of Nat. The initial conditions on the outputs are in MInit, and the initial conditions on the inputs are in EInit. A simple logical calculation shows that the formula $E \wedge M$ that describes the system consisting of the multiplier and its environment equals

$$(\mathit{MInit} \wedge \mathit{EInit}) \ \wedge\ \Box[\mathit{MNext} \vee \mathit{ENext}]_v \ \wedge\ \mathrm{WF}_v(\mathit{Finish})$$

where v denotes the 5-tuple $\langle a, x, \mathit{sig}, \mathit{out}, \mathit{ack}\rangle$. Hence, the system has the expected TLA specification.

Figure 2 contains the module $\mathit{DblMult}$, which defines the specification DM of a $2N$-bit multiplier implemented with four "internal" N-bit multipliers and a combinational four-input adder, and defines the specification DE of its environment. The implementation is based on the observation that if $a = 2^N \cdot aH + aL$ and $x = 2^N \cdot xH + xL$, then

$$a \cdot x \;=\; 2^{2N} \cdot (aH \cdot xH) + 2^N \cdot ((aH \cdot xL) + (aL \cdot xH)) + (aL \cdot xL)$$

The four multiplications are performed by four copies of the N-bit multiplier named HH, LH, HL, and LL. The external inputs and outputs of the $2N$-bit multiplier are given the same names as in the N-bit multiplier. The variables

module *Mult*

parameters

a, x, out, sig, ack : **variables**

N : Nat **constant**

predicates

$$MInit \triangleq (out \in BitVector(2 * N)) \wedge (ack = 0)$$

$$EInit \triangleq (a \in BitVector(N)) \wedge (x \in BitVector(N)) \wedge (sig = 0)$$

actions

$$\begin{aligned} Think \triangleq\ & \wedge\ sig \neq ack \\ & \wedge\ ack' = ack \\ & \wedge\ out' \in BitVector(2 * N) \end{aligned}$$

$$\begin{aligned} Finish \triangleq\ & \wedge\ sig \neq ack \\ & \wedge\ ack' = 1 - ack \\ & \wedge\ out' = Multiply(N, a, x) \end{aligned}$$

$$MNext \triangleq Think \vee Finish$$

$$\begin{aligned} ENext \triangleq\ & \wedge\ sig = ack \\ & \wedge\ sig' \in \{0, 1\} \\ & \wedge\ (a' \in BitVector(N)) \wedge (x' \in BitVector(N)) \end{aligned}$$

temporal formulas

$$M \triangleq MInit \wedge \Box[MNext]_{\langle out,\, ack \rangle} \wedge \mathrm{WF}_{\langle out,\, ack \rangle}(Finish)$$

$$E \triangleq EInit \wedge \Box[ENext]_{\langle a,\, x,\, sig \rangle}$$

Fig. 1. TLA specifications of an N-bit multiplier and its environment. The multiplier's $2N$-bit output *out* is the product of its N-bit inputs a and x. The variables *sig* and *ack* are used for synchronization.

$outHH$, …, $ackLL$ represent "internal wires" that hold the outputs and *ack*'s of the four multipliers. The four multipliers' *sig* inputs are taken from the external *sig* input, and their a and x inputs are the appropriate N-bit subvectors aH, aL, xH, and xL of the external a and x inputs.

The first **include** statement effectively defines the LL multiplier. Formally, it includes a copy of all the definitions from the *Mult* module, with the indicated substitutions for the parameters, and with the name of the defined symbols changed by prefacing them with "*LL.*". Thus the statement adds the following definition, among others, to module *DblMult*.

$$LL.MInit \triangleq (outLL \in BitVector(2 * N)) \wedge (ackLL = 0)$$

The next three **include** statements similarly include the other three needed copies of the N-bit multiplier, and the fourth one includes the definition of $Dbl.E$, the specification of a $2N$-bit multiplier's environment.

The temporal formula *ExtAck* has the usual form for a specification of a simple component that takes as input the multipliers' *ack* outputs and the external *sig* input and generates the external *ack* output.

module *DblMult*

parameters

$a,\ x,\ out,\ sig,\ ack,\ outLL,\ outLH,\ outHL,\ outHH,$
$\quad ackLL,\ ackLH,\ ackHL,\ ackHH\ :\ \textbf{variables}$

$N\ :\ Nat$

state functions

$$
\begin{array}{lcl}
aH & \triangleq & [i \in \{0, \ldots, N-1\} \mapsto a[i+N]\,] \\
aL & \triangleq & [i \in \{0, \ldots, N-1\} \mapsto a[i]\,] \\
xH & \triangleq & [i \in \{0, \ldots, N-1\} \mapsto x[i+N]\,] \\
xL & \triangleq & [i \in \{0, \ldots, N-1\} \mapsto x[i]\,] \\
acks & \triangleq & \langle ack,\ ackLL,\ ackLH,\ ackHL,\ ackHH \rangle
\end{array}
$$

include *Mult* **as** *LL* **with** $a \leftarrow aL,\ x \leftarrow xL,\ out \leftarrow outLL,\ ack \leftarrow ackLL$
include *Mult* **as** *LH* **with** $a \leftarrow aL,\ x \leftarrow xH,\ out \leftarrow outLH,\ ack \leftarrow ackLH$
include *Mult* **as** *HL* **with** $a \leftarrow aH,\ x \leftarrow xL,\ out \leftarrow outHL,\ ack \leftarrow ackHL$
include *Mult* **as** *HH* **with** $a \leftarrow aH,\ x \leftarrow xH,\ out \leftarrow outHH,\ ack \leftarrow ackHH$
include *Mult* **as** *Dbl* **with** $N \leftarrow 2 * N$

actions

$$
\begin{array}{lcl}
AckNext & \triangleq & ack' = \textbf{if}\ (ackLL = sig)\ \wedge\ (ackLH = sig) \\
 & & \qquad\qquad \wedge\ (ackHL = sig)\ \wedge\ (ackHH = sig) \\
 & & \qquad \textbf{then}\ sig \\
 & & \qquad \textbf{else}\ \ 1 - sig
\end{array}
$$

temporal formulas

$$
\begin{array}{lcl}
ExtAck & \triangleq & (ack = 0)\ \wedge\ \Box[AckNext]_{acks}\ \wedge\ \mathrm{WF}_{acks}(AckNext) \\
Adder & \triangleq & \Box \wedge out \in BitVector(4 * N) \\
 & & \quad \wedge\ Val(4 * N,\ out) = (2^{2*N} * Val(2 * N,\ outHH) \\
 & & \qquad\qquad + 2^{N} * (\,Val(2 * N,\ outLH) \\
 & & \qquad\qquad\qquad + Val(2 * N,\ outHL)) \\
 & & \qquad\qquad + Val(2 * N,\ outLL) \qquad)\ \mathrm{mod}\ 2^{4*N} \\
DM & \triangleq & LL.M\ \wedge\ LH.M\ \wedge\ HL.M\ \wedge\ HH.M\ \wedge\ ExtAck\ \wedge\ Adder \\
DE & \triangleq & Dbl.E
\end{array}
$$

Fig. 2. Specification of the $2N$-bit multiplier and its environment. Four copies of the N-bit multiplier, operating on half of each input value, are composed, and their outputs combined to form the output.

The formula *Adder* specifies the adder. It has the form $\Box P$, where P describes the appropriate relation between the adder's output *out* and its four inputs, which are the outputs of the internal multipliers. The specification is so simple because we are assuming a combinational adder, whose output is always a function of its inputs. In a real multiplier, the addition would be performed by a separate sequential component. We have used a combinational circuit both for simplicity and to illustrate how to represent such circuits. (The combinational adder makes our implementation so simple that it could be verified without the

Decomposition Theorem. However, the theorem would be needed to verify a more realistic implementation in which a send/acknowledgement protocol is used to transmit the outputs of the component multipliers to a sequential adder.)

Finally, the specification DM of the $2N$-bit multiplier is defined to be the conjunction of the specification of its components, and the specification DE of its environment is defined to be $Dbl.E$.

4 Translation into S/R

COSPAN is used to verify hypothesis 3 of the Decomposition Theorem. This requires S/R versions of E, an approximation to $E^{\oplus}$, M, $\mathcal{C}(M)$, $B(k)$ and $\mathcal{C}(B(k))$, where $B(k)$ is a specification of the low-level k-bit multiplier. These S/R specifications could all be obtained by translating TLA formulas. However, the TLA versions of $B(k)$ and $\mathcal{C}(B(k))$ are not needed. The specification $B(k)$ of the low-level multiplier can be written directly in S/R. We describe below how the S/R version of $\mathcal{C}(B(k))$ is constructed from the S/R version of $B(k)$. The S/R versions of the remaining formulas can be obtained by translation.

Since circuits are synthesized from their S/R specifications, our translation must guarantee that whatever we prove about a TLA specification is valid for its S/R translation. The TLA theorems we prove have the form $\models F \Rightarrow G$ for formulas F and G. Let $\widehat{F}$ denote the S/R translation of the TLA formula F. Correctness of the translation means that the TLA theorem $\models F \Rightarrow G$ implies the S/R theorem $\mathcal{L}(\widehat{F}) \subseteq \mathcal{L}(\widehat{G})$, where $\mathcal{L}(S)$ denotes the language of the S/R specification S.

In S/R, declaring the type of a variable guarantees that the variable's value is always of that type. Hence, to infer the validity of $\mathcal{L}(\widehat{F}) \subseteq \mathcal{L}(\widehat{G})$, it suffices to prove $\models F \Rightarrow G$ assuming type correctness. This means that we need only prove $\models F \wedge \Box T \;\Rightarrow\; G$, where T is the predicate asserting that the values of all variables have the type declared in the S/R specification. All the TLA proof obligations can thus be weakened by adding the hypothesis $\Box T$. However, this hypothesis should not be needed, because a TLA specification should imply type correctness.

The S/R translation of M appears in Figure 3. A **stvar** declaration defines a state variable, which is initialized by an **init** declaration and assigned values by an **asgn** declaration. (The $\rightarrow$ indicates that the value is assumed in the next step.) A **locvar** declaration defines a local (internal) combinational variable, which also is assigned values by an **asgn**. (The $:=$ indicates that the value is assumed in the current step, so no initialization is needed.) Assigning a set of values denotes nondeterministic choice, and the expression $e_1 \mathbin{?} b \mid e_2$ equals e_1 if b is true, else e_2. The **cyset** declaration asserts a fairness constraint, removing all behaviors that eventually remain within the declared set. The "**proc** K : KILL(p)" statement eliminates all behaviors that satisfy p.

The S/R version of M is obtained from the TLA formula as follows. The initial predicate $MInit$ yields the **init** statements for ack and out. The formula $\Box[MNext]_{\langle out,\, ack\rangle}$ yields the **asgn** statements for ack and out. The nondeter-

```
proctype M(N : integer;  a, x : (0 .. 2^N-1) ;  sig : (0, 1))
  import a, x, sig
  locvar choose : (1, 2, 3)    /* encodes [MNext]⟨out, ack⟩ choices */
   asgn choose := {1, 2, 3}  /* makes nondeterministic choice     */
  stvar ack : (0, 1)            /* ack of output variables            */
  stvar out[2∗N] : (0, 1)       /* output variables                   */
  stvar finish : (0, 1)         /* variable to express fairness       */
/* initial conditions */
  init ack := 0
  init [i in 0 . . 2∗N−1]{ out[i] := {0, 1} }
  init finish := 0
/* next-state relation */
  asgn ack →   ack       ? (choose = 1) ∗ (sig ≠ ack)
             | 1 − ack ? (choose = 2) ∗ (sig ≠ ack)
             | ack
  asgn [i in 0 . . 2∗N−1]{ out[i] →   {0, 1} ? (choose = 1) ∗ (sig ≠ ack)
                                    | mul[i] ? (choose = 2) ∗ (sig ≠ ack)
                                    | out[i]                              }
  asgn finish → 1 ? (sig ≠ ack) ∗ (choose = 2) | 0
/* implementation of "call" of Multiply in next-state relation */
  locvar mul[2∗N] : (0, 1)
   asgn [i in 0 . . 2∗N−1]{ mul[i] := {0, 1} }
  proc K : KILL( ∼( +[i in 0 . . 2∗N−1](mul[i] ∗ 2^i) =
                  +[i in 0 . . N−1](a[i] ∗ 2^i) ∗ (+[i in 0 . . N−1](x[i] ∗ 2^i))))
/* fairness */
  cyset { finish := 0 }
end M()
```

Fig. 3. The S/R translation of the TLA formula M.

minism expressed in TLA by writing the next-state relation as a disjunction is expressed in S/R by introducing a variable *choose* with as many values as there are disjuncts. In the S/R translation of M, the correspondence between values of *choose* and disjuncts of $[MNext]_{\langle out,\, ack\rangle}$ is:

$$\begin{array}{lcl}(choose = 1) & \Rightarrow & Think \\ (choose = 2) & \Rightarrow & Finish \\ (choose = 3) & \Rightarrow & \langle ack,\, out\rangle' = \langle ack,\, out\rangle\end{array}$$

A TLA fairness condition is expressed by adding a variable to record that the action has occurred and a **cyset** statement to require its occurrence. The TLA formula $\mathrm{WF}_{\langle out,\, ack\rangle}(Finish)$ is represented by the variable *finish* and the **cyset** statement.

It follows from (1) that $\mathcal{C}(M)$ equals $MInit \wedge \Box[MNext]_{\langle out,\, ack\rangle}$, so its S/R translation is the same as that of M, except with the **cyset** declaration removed. This can be deduced directly from the following result: if, in the state-transition graph, there is an exit from the set of states defined by each cyset declaration,

```
proctype E(N : integer;  out : (0 .. 2^(2*N)-1) ;  ack : (0, 1) )
  import out, ack
  locvar choose : (1, 2)         /* encodes [ENext]⟨a, x, sig⟩ choices  */
   asgn choose := {1, 2}         /*  makes non-deterministic choice */
  stvar sig : (0, 1)             /* signal of input variables          */
  stvar a[2∗N] : (0, 1)          /* variables of 1st input             */
  stvar x[2∗N] : (0, 1)          /* variables of 2nd input             */
/* initial conditions */
  init sig := 0
  init [i in 0 . . 2∗N−1]{a[i] := {0, 1}}
  init [i in 0 . . 2∗N−1]{x[i] := {0, 1}}
/* next-state relation */
  asgn sig →   {0, 1} ? (choose = 1) ∗ (sig = ack) | {sig}
  asgn [i in 0 . . 2∗N−1]{ a[i] →   {0, 1} ? (choose = 1) ∗ (sig = ack) | {a[i]} }
  asgn [i in 0 . . 2∗N−1]{ x[i] →   {0, 1} ? (choose = 1) ∗ (sig = ack) | {x[i]} }
  end E()
```

Fig. 4. S/R translation of the TLA formula E.

then the closure of an S/R specification can be obtained by removing its **cyset** declarations. This result allows us to obtain the closure $\mathcal{C}(B(k))$ of a low-level k-bit multiplier specification $B(k)$ written directly in S/R.

The S/R translation of TLA formula E is similar to that of M; it appears in Figure 4. The translations of *ExtAck* and *Adder* are straightforward and are omitted. The S/R version of the approximation to $E^{\oplus}$ can be obtained by translating the TLA formula computed with (2). It can also be obtained directly from the S/R translation of E.

The TLA to S/R translator will have to include directives for instantiating parameters and subformulas. For example, the S/R specification of a $2k$-bit multiplier is obtained from the translation of the formula *DM* from the *DblMult* module by substituting k for N and substituting copies of the S/R specification $B(k)$ of a k-bit multiplier for $LL.M$, $LH.M$, $HL.M$, and $HH.M$.

The S/R specification of the complete multiplier, from which an implementation can be synthesized, is obtained by repeated instantiation in the S/R version of DM. The translation to S/R of DM is a simple composition of the translations of its component multipliers and of the specifications *ExtAck* and *Adder*. The S/R versions of the component multipliers are obtained by instantiation of parameters from the S/R translation of M.

5 The Correctness Proof

We want to prove that our recursively-defined, low-level implementation of a $k{\cdot}2^m$-bit multiplier satisfies its high-level specification. Let $M(i)$ and $E(i)$ denote the specifications obtained by substituting i for N in the formulas M and E of module *Mult*. Let $B(k)$ be the low-level implementation of the k-bit multiplier, and recursively define $B(k \cdot 2^{m+1})$ to be the specification obtained by substituting $k \cdot 2^m$ for N and $B(k \cdot 2^m)$ for M in the formula DM of module *DblMult*. We want to prove $\models E(k \cdot 2^m) \wedge B(k \cdot 2^m) \Rightarrow M(k \cdot 2^m)$, for all $m \geq 0$. The proof is by induction on m. Letting N equal $k \cdot 2^m$, we:

(i) Prove (a) $\models E(k) \wedge B(k) \Rightarrow M(k)$
(b) $\models E(k)^{\oplus} \wedge \mathcal{C}(B(k)) \Rightarrow \mathcal{C}(M(k))$

(ii) Assume (a) $\models E(N) \wedge B(N) \Rightarrow M(N)$
(b) $\models E(N)^{\oplus} \wedge \mathcal{C}(B(N)) \Rightarrow \mathcal{C}(M(N))$
and prove (a) $\models E(2N) \wedge B(2N) \Rightarrow M(2N)$
(b) $\models E(2N)^{\oplus} \wedge \mathcal{C}(B(2N)) \Rightarrow \mathcal{C}(M(2N))$

Step (i), the base case of the induction, can be done by translating the formulas to S/R and using COSPAN. However, since one does not reason about it in TLA, the specification $B(k)$ can be written directly in S/R. If $B(k)$ implements a complex multiplier, such as the radix-4 modified version of Booth's algorithm [7], the COSPAN verification would be straightforward when $k = 8$.

Step (ii), the induction step, is proved with the Decomposition Theorem. We first express the specification $B(2N)$ as a conjunction of the specification of its components. Let prefixing by LL, LH, HL, and HH denote substitutions for a, x, out, and ack as in the *DblMult* module. Making the parameter N explicit, our recursive procedure for combining multipliers implies that $B(2N)$ equals

$$LL.B(N) \wedge LH.B(N) \wedge HL.B(N) \wedge HH.B(N) \wedge ExtAck(N) \wedge Adder(N)$$

We can infer the conclusion of (ii) from

$$\models E(2N) \wedge LL.M(N) \wedge \ldots \wedge Adder(N) \;\Rightarrow\; M(2N) \qquad (5a)$$

$$\models E(2N)^{\oplus} \wedge \mathcal{C}(LL.M(N) \wedge \ldots \wedge Adder(N)) \;\Rightarrow\; \mathcal{C}(M(2N)) \qquad (5b)$$

$$\models E(2N) \wedge B(2N) \;\Rightarrow\; LL.M(N) \wedge \ldots \wedge Adder(N) \qquad (6a)$$

$$\models E(2N)^{\oplus} \wedge \mathcal{C}(B(2N)) \;\Rightarrow\; \mathcal{C}(LL.M(N) \wedge \ldots \wedge Adder(N)) \qquad (6b)$$

We prove (6a) and (6b) with the Decomposition Theorem. Omitting trivial implications, the hypotheses of the theorem to be verified are:

1. $E(2N)$, $LL.E(N)$, ... , and $HH.E(N)$ are safety properties.
2. $\models E(2N) \wedge \mathcal{C}(LL.M(N)) \wedge \ldots \mathcal{C}(Adder(N)) \;\Rightarrow$
$LL.E(N) \wedge LH.E(N) \wedge HL.E(N) \wedge HH.E(N)$
3. (a) $\models E(N) \wedge B(N) \Rightarrow M(N)$
(b) $\models E(N)^{\oplus} \wedge \mathcal{C}(B(N)) \Rightarrow \mathcal{C}(M(N))$
4. $\models \mathcal{C}(LL.M(N) \wedge \ldots \wedge Adder(N)) \;\equiv\; \mathcal{C}(LL.M(N)) \wedge \ldots \wedge \mathcal{C}(Adder(N))$

Hypothesis 1 follows directly from (1). Hypothesis 3 is just the induction assumption (the assumption in step (ii)). Hence, to complete the proof, we must prove (5a), (5b), and hypotheses 2 and 4. The formula *Adder* is a safety property, since it has the form $\Box P$, so it equals its closure. Reverting to the notation of the *DblMult* module, we can therefore rewrite (5a), (5b), and hypotheses 2 and 4 as

$$\models DE \wedge DM \;\Rightarrow\; Dbl.M$$
$$\models DE^{\oplus} \wedge \mathcal{C}(DM) \;\Rightarrow\; \mathcal{C}(Dbl.M)$$
$$\models DE \wedge \mathcal{C}(LL.M) \wedge \ldots \wedge \mathcal{C}(HH.M) \wedge \mathcal{C}(ExtAck) \wedge Adder \;\Rightarrow\; LL.E \wedge \ldots \wedge HH.E$$
$$\models \mathcal{C}(DM) \;\equiv\; \mathcal{C}(LL.M) \wedge \ldots \wedge \mathcal{C}(HH.M) \wedge \mathcal{C}(ExtAck) \wedge Adder$$

We use (2) to substitute for $DE^{\oplus}$, and the closures are computed using (1). The resulting TLA formulas can then be verified using TLP. Urban Engberg has begun the TLP verification; he has encountered no serious difficulties and we expect the proof to be completed soon.

To complete the proof of an actual multiplier circuit, we must verify the low-level implementations of the circuits specified by the formulas *ExtAck* and *Adder*. Because *Adder* is purely combinational and *ExtAck* depends on its inputs in such a simple way, the implementations of these circuits make no assumptions about their environment (their environment specifications are identically true), and their correctness can be verified directly by COSPAN.

6 Conclusion

We have shown how two tools, TLP and COSPAN, can be combined to form a direct path from high-level specification to synthesis. Together, the tools can be applied to problems that would be difficult or impossible to handle with either tool separately. We can combine these very different tools because the languages on which they are based, TLA and S/R, have similar semantics. TLA is very expressive, but verifying TLA specifications requires theorem proving. S/R is less expressive, but S/R specifications can be verified by automatic model checking. Theorem proving involves a great deal of human effort. Model checking is automatic, but its use is limited by its inherent computational complexity. We combine the two approaches to make theorem proving as simple as possible by proving as much as we can with model checking.

Our approach does involve checking some steps by hand. For example, TLP does not verify that we have correctly applied the Decomposition Theorem. These steps are few, very simple, and easy to check—even for a complex problem. Mechanically verifying these steps is possible, but it may provide too small a gain in reliability to be worth the effort.

The work described here is not complete. We still need to mechanize the TLA to S/R translation, and to try more demanding examples in which the low-level specification is not only complex, but is also replicated in a more complex fashion.

References

1. Martín Abadi and Leslie Lamport. Open systems. To appear in 1993 as a SRC Research Report.
2. Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions On Computers*, C-35(8):677–691, August 1986.
3. Shiu-Kai Chin. Verified functions for generating signed-binary arithmetic hardware. *IEEE Transactions on Computer-Aided Design*, 11(12):1529–1558, December 1992.
4. Urban Engberg, Peter Grønning, and Leslie Lamport. Mechanical verification of concurrent systems with TLA. In *Computer-Aided Verification*, Lecture Notes in Computer Science, Berlin, Heidelberg, New York, June 1992. Springer-Verlag. Proceedings of the Fourth International Conference, CAV'92.
5. Z. Har'El and R. P. Kurshan. Software for analytical development of communication protocols. *AT&T Technical Journal*, 69(1):44–59, 1990.
6. J. Katzenelson and R. P. Kurshan. S/R: A language for specifying protocols and other coordinating processes. In *Proceedings of the 5th Annual International Phoenix Conference on Computer Communications*, pages 286–292, Scottsdale, Arizona, 1986. IEEE Computer Society.
7. Israel Koren. *Computer Arithmetic Algorithms.* Prentice Hall, Englewood Cliffs, New Jersey, 1993.
8. R. P. Kurshan. Reducibility in analysis of coordination. In P. Varaiya and A.B. Kurzhanski, editors, *Discrete Event Systems: Models and Applications*, volume 103 of *Lecture Notes in Control and Information Sciences*, pages 19–39, Berlin, 1987. Springer-Verlag.
9. R. P. Kurshan. Analysis of discrete event coordination. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*, pages 414–453. Springer-Verlag, May/June 1989.
10. R. P. Kurshan and K. McMillan. A structural induction theorem for processes. In *Proceedings of the 8th annual ACM Symposium on Principles of Distributed Computing*, pages 239–247. ACM Press, 1989.
11. Leslie Lamport. What good is temporal logic? In R. E. A. Mason, editor, *Information Processing 83: Proceedings of the IFIP 9th World Congress*, pages 657–668, Paris, September 1983. IFIP, North-Holland.
12. Leslie Lamport. The temporal logic of actions. Research Report 79, Digital Equipment Corporation, Systems Research Center, December 1991.
13. Leslie Lamport. Hybrid systems in TLA$^+$. In Hans Rischel and Anders P. Ravn, editors, *Hybrid Systems*, Lecture Notes in Computer Science, Berlin, 1993. Springer-Verlag. Proceedings of a Workshop on Hybrid Systems, to appear.