A methodology for hardware verification using compositional model checking

K. L. McMillan
Cadence Berkeley Labs
2001 Addison St.
Berkeley, CA 94704
USA
mcmillan@cadence.com

April 15, 1999

Abstract

A methodology for system-level hardware verification based on compositional model checking is described. This methodology relies on a simple set of proof techniques, and a domain specific strategy for applying them. The goal of this strategy is to reduce the verification of a large system to finite state subgoals that are tractable in both size and number. These subgoals are then discharged by model checking. The proof strategy uses proof techniques for design refinement, temporal case splitting, data type reduction and the exploitation of symmetry. Uninterpreted functions can be used to abstract operations on data. A proof system supporting this approach generates verification subgoals to be discharged by the SMV symbolic model checker. Application of the methodology is illustrated using an implementation of Tomasulo's algorithm, a packet buffering device and a cache coherence protocol as examples.

1 Introduction

Formal methods for hardware verification divide into two broad categories. One set of methods is based on model checking and related techniques for finite state machine verification. The other is based on automated proof assistants. Model checking methods have the advantage that they can automatically verify a fairly broad class of properties – those expressible in temporal logic, or as the language of an automaton. They also can provide behavioral counterexamples in case a property is false. This can only be done, however, only in the case where the state space of the system being verified is relatively small. Realistically if the model being verified has 50 or 100 bits of state information, it is likely that it can be successfully verified (though not certain). On the other hand, with 500 to 1000 bits of state information, the chance of successful verification is slight. This is because model checking methods are based on exhaustively searching the model's state space, either explicitly or implicitly. Thus, one can use model checking methods to verify, for example, fairly complex finite state control circuits, or highly abstracted models of protocols, but not in general circuits that have large data paths or memories. While progress continues to be made in extending model checking to larger systems, real integrated systems are still several orders of magnitude too large to be verified in their entirety by model checking, and there is little prospect that this gap will close.

Theorem proving methods, on the other hand, impose no *a priori* limit on the size or complexity of the system that can be verified. One has in principle the possibility of breaking down proofs about very large systems into proofs about smaller components, and thus controlling the complexity of the verification process. However, effective automation for handling the detailed aspects of the proof has been lacking. As a result, when using a proof assistant, one is typically obliged to posit inductive invariants of the system, which can be quite detailed. Proving these invariants may involve interactively constructing very detailed proof scripts, in which the user guides the prover through an extensive case analysis. Inductive invariants and proofs scripts must also be maintained as the design changes.

It is natural to suppose that a synthesis of the two verification styles could be made, whereby the details of the proof are handled automatically by a model checker, while the reasoning at a more abstract level is handled by a proof assistant. In fact, some work has been done on integrating model checking algorithms into the framework of a theorem prover [Sha96]. However, integration of theorem proving and model checking tools is not sufficient by itself to provide a practical method of verification. One also requires a proof methodology that naturally yields proof subgoals verifiable by model checking. These subgoals must be reasonably small in number, and the state spaces involved must be small enough to allow a high confidence that model checking can be completed successfully.

The purpose of this article is to propose such a methodology, for the particular domain of hardware verification. The proof strategy is based on a small collection of techniques used to reduce large (possibly infinite state) verification problems to small finite state problems. These techniques have been implemented in a special-purpose proof system that produces finite state verification subgoals to be discharged by the SMV model checker [McM93]. The system to be verified, its specification, and the proof itself are all expressed in a somewhat extended hardware description language. Potentially, however, a similar system could be implemented as a "tactic" within a general purpose proof assistant. It should be noted that the process of proof decomposition is not automatic. It is aided by automatic tools, but requires human insight about the structure and function of the design being verified.

1.1 The scope of this article

This article is intended to provide an overview of a methodology, as opposed to a rigorous formal treatment. Thus, while proof techniques are introduced here in the abstract, we will not consider in detail the underlying model theory or prove that the techniques are sound in a given theory. These or similar techniques could be derived in many contexts. The intention here is to motivate the approach intuitively, and to illustrate by example a strategy by which the various techniques can be combined to reduce the verification of complex systems to small model checking problems.

We will begin, in the next section, with a discussion of the overall proof strategy in general terms. Following that, the method is illustrated by application to a simple implementation of Tomasulo's algorithm [Tom67], a technique of implementing out-of-order execution in instruction set processors. This will provide a more detailed view of how the methodology is supported by the SMV model checking system. There follows a less detailed discussion of two additional examples, a communications application and the cache coherence system of a multiprocessor. Details of these examples are omitted, as they are intended principally to show by example that the methodology generalizes to systems other than instruction processors. A discussion of related work is postponed to the end of the article, to allow a more meaningful comparison of techniques.

1.2 Preliminaries

Temporal logic is a logical language commonly used for expressing properties of concurrent systems. Here, we will use the variant called *linear temporal logic* or LTL. In its propositional version it allows the usual propositional operators (and, or and not) and in additional a collection of temporal operators used to express relations in time. For example, given a proposition p, the formula Fp is true at a given time if p is true at *some* time in the future, the formula Gp is true when p holds at *all* times in the future, and Xp is true when p holds at the next time instant (assuming time is discrete). The formula $p\ U\ q$ is true at a given time if q holds at some instant in the future, and p holds at all times at all times up to, but not necessarily including, that instant.

In LTL, a *model* is an infinite sequence $s_0, s_1, \ldots$ of *states*, representing consecutive time instants. Each formula is either true or false in a given state. We write $(M, s_i) \models \phi$ if the formula ϕ is true in state s_i of model M. Thus, for example:

$$\begin{array}{rcl} (M, s_i) \models X\phi & \text{iff} & (M, s_{i+1}) \models \phi \\ (M, s_i) \models \phi\ U\ \psi & \text{iff} & \text{for some } j \geq i, \\ & & \quad (M, s_j) \models \psi \text{ and for all } i \leq k < j,\ (M, s_k) \models \phi \end{array}$$

We say a model *satisfies* a formula, and write $M \models \phi$, when the formula is true in the initial state (that is, when $(M, s_0) \models \phi$). If a formula is true in all models, we say it is *valid* and write $\models \phi$. The *decision problem* is to determine whether a given formula is valid. Given a finite state graph, the *model checking problem* is to determine whether the formula is true for all paths in the state graph starting with an initial state. For propositional LTL, both these problems are PSPACE complete, and can be solved in time exponential in the formula size [VW86, LP85]. In the model checking case, the time used is linear in the size of the state graph, and exponential in the formula size. In practice, there is little operational distinction between validity checking (truth in all models) and model checking (truth in all paths in a state graph), since the state graph itself is usually characterized implicitly by a logical formula. The most important factor in the complexity of model checking finite state systems tends to be the number of reachable states, that is, feasible combinations of the state variables used to model the system. Since the number of reachable states can be exponential in the number of state variables, model checking is only effective in practice for models with a relatively small number of state variables.

2 Proof strategy

We now outline a proof strategy, based on a set of somewhat domain specific proof techniques, that is intended to reduce the verification of large, complex hardware systems to subproblems with few enough state variables to be verifiable by model checking. The overall approach is to verify that an abstract model, acting as the system specification, is implemented by some more detailed system model. Implementation is defined in terms of *refinement relations* that relate signaling behavior at suitable points in the implementation with events occurring in the abstract model. This situation is depicted in figure 1. Typically, the abstract model, the implementation and the refinement relationss are all expressed in the same HDL-like language, as sets of equations that may involve time delay. Formally, these are all viewed as "syntactic sugar" for properties in temporal logic.

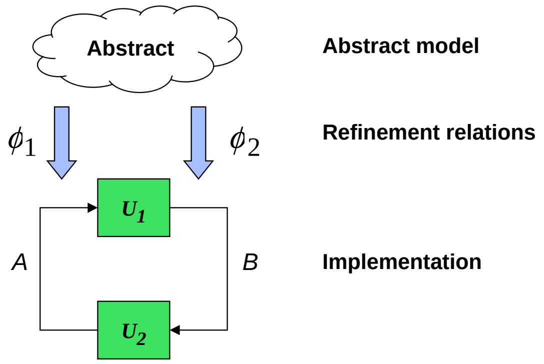

Figure 1: Compositional refinement verification

The refinement relations decompose the verification problem into smaller parts for separate, localized, verification. This is the most basic way in which large proofs are broken into smaller proofs. To do this, we rely on a technique of *circular compositional proof* which allows us to assume that one relation holds true while verifying another, and *vice versa*. In effect, the refinement relations allow us to use the abstract model as a context for verifying local components of the detailed model. Thus, we avoid the difficult problem of verifying components of the design in an unconstrained environment.

Having made a macro level decomposition of the problem by refinement relations, one is then commonly left with large data structures, such as memories or register files. These generally cannot be handled directly by a model checker because they have a large number of state bits. However, we can often reduce the model down to just one or two components of the larger structure by means of *temporal case splitting*. Using this technique, we can verify separately only those data items that pass through a given fixed element of a large array, and thus greatly reduce the number of state bits in the model checking problem. This technique is called *path splitting*. It allows us to consider individually every path that a data item might take through a given system, and to reduce the number of state variables in the model checking problem accordingly.

Use of refinement relations and path splitting can result in a very large number of individual cases to verify (each of which yields a model checking problem). We can often reduce these to a small, representative number of cases, however, by exploiting symmetry. The implementation of this technique in the SMV proof system is based on the use of symmetric data types called *scalarsets*, borrowed from the Murphi system [ID96]. Type checking ensures that a scalarset variable is used only in symmetric ways, guaranteeing that the truth of a given formula is invariant under permutations of the scalarset type. This allows a *symmetry reduction* to reduce a large (or even infinite) parameterized class of proof subgoals to just a few representative cases. Note that symmetry is *not* used here to reduce state space size, as it is in the Murphi system.

Finally, after the large arrays have been reduced to their individual components, one may still be left with large data types, such as addresses, or data words. Note that "large" here means that the type has a large number of possible values, not that it requires a large amount of storage space. In this case, a *data type reduction* can be used. This reduces a large (perhaps infinite) type to a small finite one, containing only one or two values that are of interest for the case being verified. The remaining values are represented by a single abstract value. Thus, data types represented by 32 or 64 bits can be reduced to just one or two bits for the purpose of model checking.

The above elements can be assembled into a general proof strategy for hardware designs. Using this approach, one can reduce the verification of a large and complex system down to finite state verification problems with small numbers of state bits. When the number of state bits in each proof subgoal is sufficiently small, verification can proceed in a completely automated way without further manual intervention. Thus, we do not rely on a powerful model checker to handle very large problems (though this can be useful), but rather on our ability to decompose large verification problems into small ones. Nonetheless, model checking is still a crucial component of this strategy. The application of model checking relieves the user of responsibility for the details of the proof, in particular of the the need to write inductive invariants. The result is a scalable, practical methodology for system level hardware verification.

We will now consider in a little more detail the various proof techniques described above, and how they are combined in a global proof strategy.

2.1 Refinement and circular compositional proofs

As mentioned above, in refinement verification, we specify a collection of temporal properties called refinement relations. These relate the behavior of a simple abstract model to a more complex and detailed implementation. While refinement relations can be arbitrary temporal properties, in practice they usually are downward translations, mapping events or data in the abstract model to corresponding signaling sequences at some chosen point in the implementation. Thus, for example, in figure 1, the temporal property ϕ_1 defines the legal behaviors of signal A, as function of the abstract model, while ϕ_2 defines the legal behaviors of signal B.

The refinement relations, once specified, can be used to to break the verification problem into smaller parts. For example, when we verify that signal B satisfies specification ϕ_2, we simply assume that signal A satisfies ϕ_1. This allows us to abstract away unit U_2. In effect, the asbtract model becomes the "environment" for verifying unit U_1. Similarly, when verifying that signal A satisfies property ϕ_1, we assume ϕ_2 and abstract away unit U_1. This approach solves a significant practical problem in system level verification. That is, we often cannot verify any useful properties of a given unit in a unconstrained environment. In this case, the abstract model provides the necessary environment constraits, by way of the refinement relations.

Note that, on the face of it, this reasoning is circular, since we have assumed ϕ_1 to prove ϕ_2, and assumed ϕ_2 to prove ϕ_1. However, there is a way of stating this argument which is in fact logically sound. To do this, we break the circularity by induction over time. When we prove that property ϕ_2 holds at time t, we assume that ϕ_1 holds at all times from 0 up to $t-1$. Similarly, to prove that ϕ_1 holds at time t, we assume that ϕ_2 holds at all times from 0 up to $t-1$. We can then infer by induction over time that both ϕ_1 and ϕ_2 hold for all time t. Schematically, we make the following inference:

$$\begin{array}{rcl} \phi_1 \text{ up to } t-1 & \Rightarrow & \phi_2 \text{ up to } t \\ \phi_2 \text{ up to } t-1 & \Rightarrow & \phi_1 \text{ up to } t \\ \hline \multicolumn{3}{c}{\text{for all } t,\ \phi_1 \text{ and } \phi_2} \end{array}$$

The two statements above the line, which we must prove, can be written succinctly in linear temporal logic, as follows:

$$\neg(\phi_1\ U\ \neg\phi_2)$$

$$\neg(\phi_2\ U\ \neg\phi_1)$$

Since we can state these propositions in linear temporal logic, we can verify them by model checking. In essence, these two proof subgoals state that neither ϕ_1 nor ϕ_2 is the first to be false. It follows trivially that both must be true at all times. That is, we can infer $G\phi_1$ and $G\phi_2$. In fact, this is true for any temporal properties ϕ_1 and ϕ_2, including "liveness" properties (properties that state some progress condition that must eventually occur). We prove the first subgoal using only unit U_1, and the second using only unit U_2. This results in two model checking problems with reduced state spaces, since the state variables of U_2 can be ignored when verifying U_1, and *vice versa.*

This circular compositional proof principle can be generalized to any number of refinement relations. In general, if we have some collection of properties $G\phi_i$ to prove, we can choose for each ϕ_i an arbitrary subset Δ_i of the properties to act as its "environment abstraction". If we can verify for every i that

$$\neg(\Delta_i\ U\ \neg\phi_i)$$

we can infer that $G\phi_i$ holds for all i. In effect, to prove each property ϕ_i at time t, we assume that the properties in Δ_i hold up to time $t-1$. We can, of course, also assume an arbitrary subset Γ_i of the properties defining the implementation. Thus, we verify that

$$\Gamma_i \Rightarrow \neg(\Delta_i\ U\ \neg\phi_i)$$

To use the circular compositional technique to verify a collection of refinement relations, the user need only specify an "environment" for proving each relation, in terms of Δ_i and Γ_i. In practice, as we will see later, only a small subset of the environment needs to be specified explicitly. The rest can be inferred using heuristics, including a "cone of influence" analysis.

2.1.1 Circular compositional reasoning with combinational paths

It may happen that assuming ϕ_1 up to time $t-1$ when proving ϕ_2 is not sufficient. This might be the case, for example, if the unit U_1 contained a combinational logic path from its input A to its output B. In this case, a change in A would be reflected at B with no intervening delay. Thus, it might be necessary to assume that ϕ_1 holds up to and *including* time t to prove correctness of ϕ_2 at time t.

This can be done by slightly generalizing the circular compositional proof technique. First, we put all of the properties ϕ_i into a well-founded order that reflects the combinational dependency relation between the signals. For example, if signal B depends on A via some zero delay path (but not the other way around), then we say $\phi_1 < \phi_2$. If $\phi_1 < \phi_2$, we can assume ϕ_1 up to time t when proving ϕ_2 at time t, otherwise we assume ϕ_1 only up to time $t-1$. That is, in general, we divide the "environment abstraction" for property ϕ_i into two subsets, Θ_i being those properties less than ϕ_i in the order (and thus assumed up to time t), and Δ_i being those not less than ϕ_i (and thus assumed up to time $t-1$). It is then sufficient to prove for every i that:

$$\Gamma_i \Rightarrow \neg(\Delta_i\ U\ (\Theta_i \wedge \neg\phi_i))$$

to infer $G\phi_i$ for all i. In essence, we show that no property ϕ_i is the first to fail, where "first" is determined primarily by time, but in case of a tie is determined by the given well-founded order on the properties. It follows that no property ever fails. Note once again that because the proof subgoals are expressed in temporal logic, they can be verified directly by model checking.

It should also be noted that the above proof goals are generated automatically by the proof system and not written by the user. The user supplies only the refinement relations. Further, the distinction between Θ_i and Δ_i is for the most part invisible to the user. The proof system makes this distinction automatically, based on a data-flow analysis of the system.

2.2 Auxiliary state

A useful strategy in defining refinement relations is to add auxiliary state information to the design. Usually, an auxiliary state variable is associated with a particular component of the design. It carries either the correct value of the data stored in that component, (according to the abstract model) or a pointer to that value in the abstract model. When introducing definitions of auxiliary variables, we must ensure that adding these definitions does not affect the truth value of propositions about existing implementation variables. Thus, if we can prove a property using some auxiliary definitions, we know that the property remains true when the auxiliary definitions are removed. This principle is known as "conservative extension".

Typically, conservative extension is ensured in a theorem proving system by requiring that each new definition introduces a new object never before referenced. However, it is useful to be able to write auxiliary definitions that are mutually dependent, provided conservative extension can be guaranteed (for example, this is useful for defining the internal state of abstract models).

In the SMV system, a signal definition is a nondeterministic assignment. This assignment gives the set of possible values for the signal at time t, as a function of other signals at time t, or time $t-1$. The former case is referred to as a zero delay assignment, while the latter case is referred to as a unit delay assignment. Formally, these are both just special cases of temporal properties. SMV guarantees conservative extension by enforcing the following rules:

- No implementation signal depends on an auxiliary signal.
- Each auxiliary signal is assigned only once.
- Every dependency cycle in the auxiliary signal assignments is broken by a unit delay.

Within the above rules, the user is free to introduce any number of auxiliary signals, arbitrarily defined, in no particular order. The rules ensure that for any implementation behavior, we can compute at least one behavior of the auxiliary signals that satisfies the auxiliary definitions. This allows us to make the following inference:

$$\frac{(\Gamma \wedge A) \Rightarrow \phi}{\Gamma \Rightarrow \exists v_A : \phi}$$

where A is the set of auxiliary definitions, and v_A is the set of auxiliary variables. That is, if we can prove a property ϕ using the implementation Γ and the auxiliary definitions A, then it follows that every implementation behavior satisfies ϕ for *some* valuation of the auxiliary signals. The auxiliary definitions provide the "witness" for this existential quantifier.

Note that this use of auxiliary variables is essentially the same as that introduced by Owicki and Gries [OG76]. The primary difference is that, in the Owicki and Gries method, the auxiliary variables are defined by statements added to a sequential program, whereas here they are introduced by signal assignments.

3 Temporal case splitting

Recall that our overall goal is to reduce a large verification problem down to small finite state problems. Having decomposed the verification problem into localized subproblems, using refinement relations and auxiliary state, the next problem that usually arises is the presence of large structures, such as register files, memories, FIFO buffers, *etc.* The number of state bits in these structures is often too large to handle them in their entirety with model checking. We solve this problem by splitting the verification subgoals into cases, where each case corresponds to just one element in a large structure (for example, just one memory address). Typically, this means that when verifying that correct data appear at the output of a given unit containing a register file, we will consider only the correctness of those data items that happen to have been stored in register i. When verifying the property for a particular value of i, we can abstract away the other elements of the register file, giving them an undefined value. The state space of the resulting model checking problem is thus reduced, since it involves only one register instead of many. This situation is depicted in figure 2.

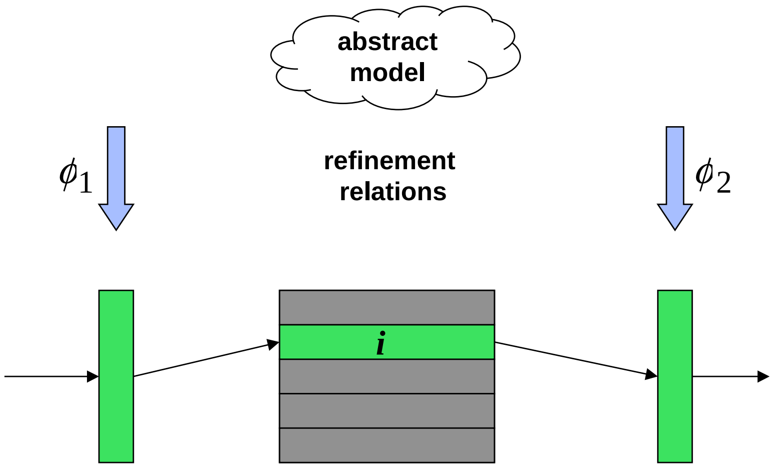

Figure 2: Path splitting.

To verify the correctness of only those data items that pass through a particular element in a large structure, we use *temporal case splitting*. This method is used to prove that a particular property ϕ holds at all times. It breaks the proof into cases based on the value of a given variable v. For each possible value i of v, we show that ϕ is true at just those times when $v = i$. Since at all times v must have some value, we can then infer that ϕ must be true at all times. Thus, if we can prove separately for each i that $G((v = i) \Rightarrow \phi)$, we can infer $G\phi$.

Typically, v is an auxiliary variable which keeps track of the location in some large data structure which was used to store a given item appearing at a unit output. In some cases, such a variable may already be present in the implementation (for example, the address field on a bus indicates which memory cell a given data item was obtained from). By case splitting, we obtain one proof subgoal for each possible value i of v. Each case requires that property ϕ hold at those times when $v = i$ (for example, those times when the current data item passed through memory location i). Thus, to prove each case, we commonly need to use the implementation definition of just one cell in the array. In effect, we have decomposed a large array into its individual components for the purposes of verification. This technique might be referred to as "path splitting", since we prove one case for each possible path that a given data item might take through a system.

4 Symmetry reductions

Clearly, the path splitting approach trades off a problem of state explosion for a problem of case explosion. For example, to prove a property of a memory of 2^{32} bytes, we would have 2^{32} separate cases to prove. Or, suppose that a data item entering a unit is stored first in element i of register file A, and then in element j of register file B, before being sent to an output. Here, we have two cases splits to perform, yielding $O(n^2)$ cases to prove, where n is the number of registers in each file. Verifying all of these cases would be wasteful, since it is likely that most of the cases are isomorphic. In other words, we should be able to prove just one case (say, $i = 0$, $j = 0$) and then infer all of the other cases "by symmetry".

In fact, this is possible, provided we make the symmetry in the system explicit. In the SMV system, this is done by means of symmetric data types called *scalarsets* [ID96]. Restrictions on the use of scalarset types ensure that the semantics of all assertions is invariant under permutations of values of a scalarset type. That is, we can swap the roles of any two constants of the type, without changing the truth value of a given formula. This property is enforced by means of type checking rules. In practice, this means that scalarset variables can be compared for equality against variables of the same type, but no other operations may be applied to them. In addition, they may be used as array indices, and arrays or functions over scalarset types may be "summed" using any commutative-associative operator. For example, we can form the logical conjunction of $p(i)$, for all i, where i is of scalarset type. Any other use of a scalarset variable breaks the symmetry of the type. In particular, individual constants of scalarset types may not be directly introduced without breaking the symmetry.

Now suppose that we have split a property into cases based on the value i of a scalarset variable v, and that we have proved the case $i = 0$. We can then infer that the case $i = 1$ holds as well, since we obtain the case $i = 0$ from $i = 1$ by simply transposing the values 0 and 1 of the scalarset type. We know that the truth value of formulas is invariant under such permutations. The same argument holds for all other values of the type. Thus, it is sufficient to prove just one case, say $i = 0$, to infer all the others by symmetry.

Carrying on, if we split cases based on two values i and j, of the same scalarset type, then it is sufficient to prove cases $i = 0, j = 0$ and $i = 0, j = 1$. All the cases where $i = j$ can be reduced to the former by permuting scalarset values, while all the cases where $i \neq j$ can be reduced to the latter. In general, if ϕ has some free parameter i of a scalarset type, and if ϕ contains the constants $0 \ldots k-1$ of the given type, it is sufficient to prove ϕ for the cases $i = 0 \ldots k$ to infer that ϕ holds for all i. The idea is that all cases $i > k$ are equivalent to $i = k$, by simply exchanging the constants i and k. As a result, if we have n parameters of a given scalarset type, then after applying the above reduction n times we will have $n!$ subgoals to prove (for the first parameter, $k = 0$, for the second $k = 1$ and so on). For example, if we have split cases on three addresses, ranging from 0 to $2^{32} - 1$, then we will have $3! = 6$ subgoals to prove instead of the $2^{32} \times 2^{32} \times 2^{32}$ subgoals we would have without symmetry reduction.

In order to use symmetry reduction in practice, we have only to specify the scalarset types of the variables. A representative set of cases for each property can then be generated automatically by the proof system.

5 Data type reductions

Up to this point, we have seen how to break the verification of large systems down to more localized subgoals by specifying refinement relations. The large structures are then broken down into their individual elements by path splitting, and the resulting large number of cases is reduced to a tractable number by symmetry considerations. At this point, there usually remains one additional impediment to obtaining tractably small state spaces for model checking. That is the presence of types with large (or infinite) ranges, such as addresses and data words.

Large data types are dealt with by reducing them to smaller abstract types. For example, if we are proving case $v = i$ of a property, where v is of a given type T, then we might reduce type T to just two values: the specific value i that we are interested in, and an abstract value $T \setminus i$ to represent all the other values in the type. A variable of the reduced type can thus be represented by a single bit.

Having abstracted the model in this way, we use a corresponding abstract interpretation of formulas. This interpretation guarantees that any property that is true in the reduced model is also true in the original model (that is, it is a "conservative abstraction"). For example, consider the equality comparison operation. In order to obtain a conservative abstraction, we use the following truth table for equality:

$=$	i	$T \setminus i$
i	1	0
$T \setminus i$	0	$\perp$

That is, clearly the specific value i is equal to itself, and not equal $T \setminus i$, since this represents all the values not equal to i. However, two values not equal to i may themselves be equal or unequal. The abstraction does not present enough information to make the distinction. Thus, to be conservative, the result of comparing $T \setminus i$ and $T \setminus i$ for equality is the unknown value $\perp$. With this interpretation, we can be sure that a property verified on the model with the reduced type is also true of the unreduced model. It is possible, however, for the truth value of a formula in the reduced model to be $\perp$, in which case we obtain no information about the formula.

In practice, an appropriate data type reduction for a given type can often be inferred automatically, based on the particular case being verified. For example, if the case we are verifying has two parameters, i and j, both of type T, then by default type T would be reduced to the set $\{i, j\}$, plus an abstract value.

5.1 Summary

In summary, we have seen a collection of proof techniques designed to be used in a general strategy for hardware verification. Their primary purpose is to reduce the verification of large systems to a small number of tractable model checking problems. The framework of the proof strategy is refinement verification. Here, we specify the behavior of a system with respect to an abstract model, by means of refinement relations, or translations from abstract to detailed behavior. Refinement relations can be introduced as needed to divide the global verification problem into localized subproblems. In each subproblem, a part of the implementation is verified in the environment of the abstract model, with the refinement relations acting as intermediary. A method of *circular compositional proof* makes this style of proof decomposition possible.

For pure control logic, this style of structural decomposition is often sufficient to reduce the verification to small finite state subgoals. However, for designs with data, additional reductions are required. First, the large structures, such as memories and FIFO buffers, are divided into their individual elements by means of *path splitting*. This allows us to consider each possible path that a data item might take through a system as a separate case, and thus allows us to abstract away all but a few elements of the large structures. The large number of resulting cases is then reduced to

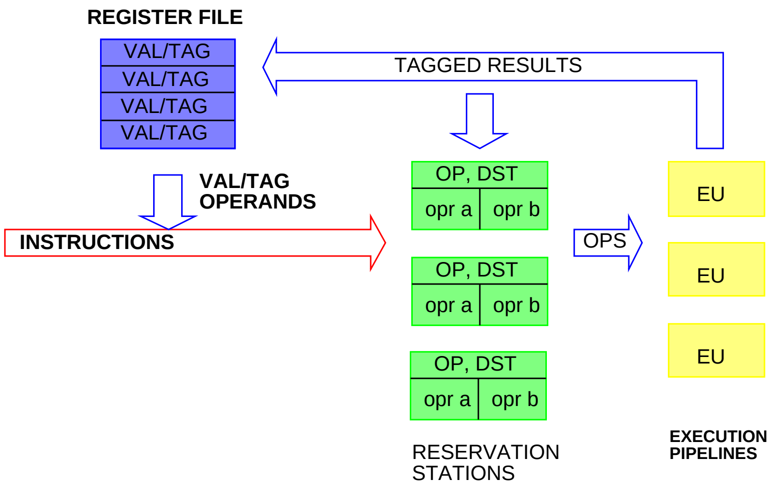

Figure 3: Flow of instructions in Tomasulo's algorithm

a few representative cases by means of *symmetry reduction*, which relies on the use of symmetric data types called *scalarsets*.

Finally, the types with large or infinite ranges (such as memory addresses, or packet identifiers) are reduced to small finite types by means of a *data type reduction*. A suitable abstract interpretation of operators guarantees that the reduced model is a conservative abstraction of the original. The result is a set of proof subgoals whose models are small enough to handle directly with model checking.

6 Verifying a version of Tomasulo's algorithm

In this section, we will see a concrete example of how the above strategy can be applied to reduce a large verification problem to tractable finite state proof subgoals.

6.1 Tomasulo's algorithm

Tomasulo's algorithm [Tom67] allows an instruction set processor to execute instructions in data-flow order, rather than sequential order. This can increase the throughput of the unit, by allowing instructions to be processed in parallel, or avoiding pipeline stalls due to hazards. Each pending instruction is held in a "reservation station" until the values of its operands become available, then issued "out-of-order".

The flow of instructions in our implementation of Tomasulo's algorithm is pictured in figure 3. Each instruction, as it arrives, fetches its two operands from a special register file. Each register in this file holds either an actual value, or a "tag" indicating the reservation station that *will* produce the register value when it completes. The instruction and its operands (either values or tags) are stored in a reservation station. The reservation station watches the results returning from the execution pipelines, and when a result's tag matches one of its operands, it records the value in place of the tag. When the reservation station has the values of both of its operands, it may issue its instruction to an execution pipeline. When the tagged result returns from the pipeline, the reservation station is cleared, and the result value, if needed, is stored in the destination register. However, if a subsequent instruction has modified the register tag, the result is discarded. This is because its value in a sequential execution would be overwritten.

In addition to ALU instructions, our implementation includes instructions that read register values to an external output and write values from an external input. There is also a “stall” output, indicating that an instruction cannot currently be received. A stall can happen either because there is no available reservation station to store the instruction, or because the value of the register to be read to an output is not yet available.

6.2 Structural decomposition

To specify our machine, we begin by writing an abstract model. The abstract model is a simple implementation of the instruction set that executes the instructions one at a time in sequence (this is commonly referred to as an “instruction set architecture” model). The abstract model is shown schematically in figure 4. In this case, the input and output signals of the abstract model and the implementation are the same, so there is no need to write refinement relations for them.

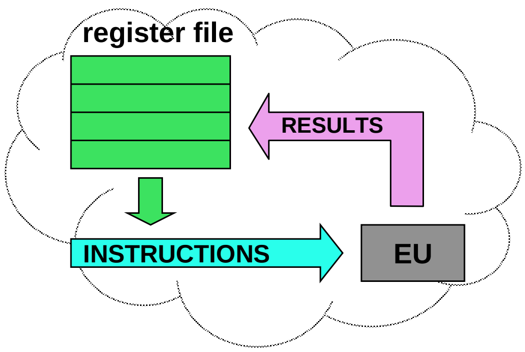

Figure 4: Instruction set architecture model.

Typically, the first decomposition one makes when verifying an instruction set processor is to break the problem into two lemmas. The first states that operands fetched for instructions are correct, while the second states that the results produced from these operands are correct. Each of these lemmas is a refinement relation. In this case, the first relation specifies the operand values stored in the reservation stations, while the second specifies the values returning on the result bus from the execution units. Naturally, we apply circular compositional proof, using operand correctness to prove result correctness and result correctness to prove operand correctness.

The use of these two refinement relations divides the verification of our instruction set processor fairly neatly in two parts. When proving that the fetched operands are correct, we use the result refinement relation to specify the result bus. This eliminates from the proof all of the logic involved in issuing instructions to the execution units, the execution units themselves, and the completion logic that routes execution results onto the result bus. This logic is removed from the model automatically by data flow analysis. That is, when we choose to define the result bus values according to the refinement relation, the dependency of this bus on the execution unit logic is eliminated. A so-called “cone of influence” reduction therefore removes this logic from the model, on the grounds that it cannot influence the model checking result. The resulting system abstraction is depicted in figure 5. Conversely, when the result correctness lemma is proved, we assume that the operands fetched by the reservation stations are correct, and hence we eliminate all of the operand fetch logic and the register file from the model.

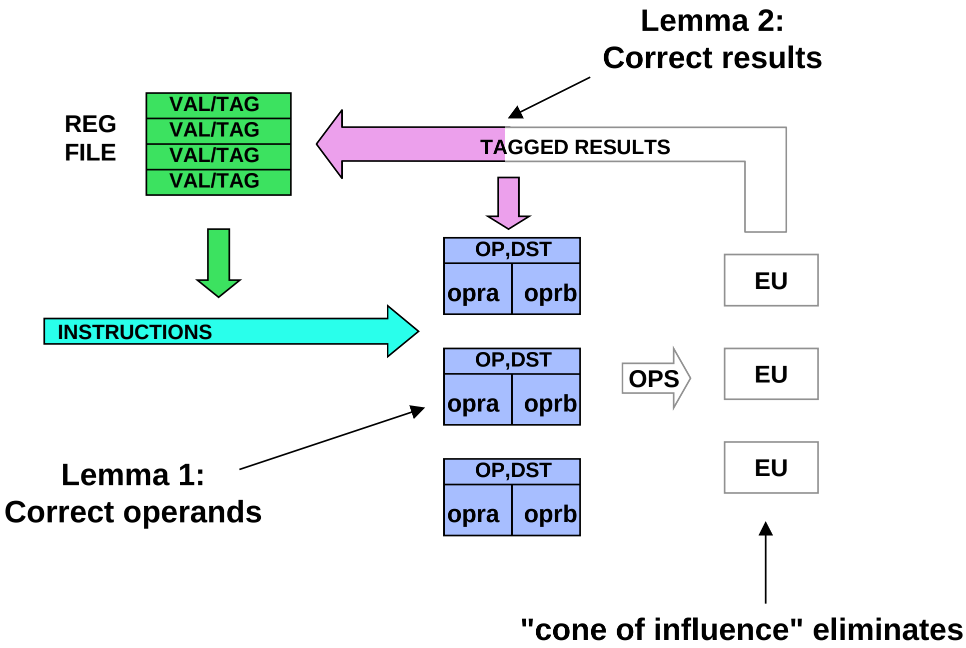

Figure 5: Decomposition by means of refinement relations.

6.2.1 Auxiliary state

In order to be able to specify refinement relations for the operand and result values, we need to know what the correct values for these data items actually are. We obtain this information by adding auxiliary state to the model. In this case, our auxiliary state variables record the correct values of the operands and result of each instruction, as computed by the abstract model. These values are recorded at the time an instruction enters the machine and is stored in a reservation station. This operation is depicted in figure 6, along with the SMV assignments that implement it. For the auxiliary state, we use an array `aux` with one element per reservation station. The signal `st` indicates the reservation station in which the incoming instruction is being stored. Thus, if the machine does not stall, and if the incoming instruction is an ALU operation (meaning it is destined for an execution unit), then we store in the auxiliary array the correct values of the two operands (`opra` and `oprb`) and the result `res` from the abstract model. Storing these values will allow us to verify that the actual operands and results we eventually obtain are correct. As mentioned earlier, assignments are just a special case of temporal properties. In this case, we have unit delay assignments, which specify the next value of a signal as a function of the current values of some other signals. Also note that, as `aux` is an auxiliary signal, the implementation is not allowed to refer to it.

In SMV, the refinement relations themselves are normally also written as assignments. They could be written as arbitrary temporal formulas, but the assignment form has the advantage that we can readily determine which signals each signal depends on when performing the "cone of influence" computation. These assignments are special, however, in that they are properties to be proved, and not part of the implementation. They are grouped into named collections called *layers*. Layers have a special function in the "design by refinement" mechanism of SMV [McM99b], but for present purposes can be thought of as simply collections of properties to be proved. In particular, it is possible to specify many refinement relations assigning the same signal, but only if they are

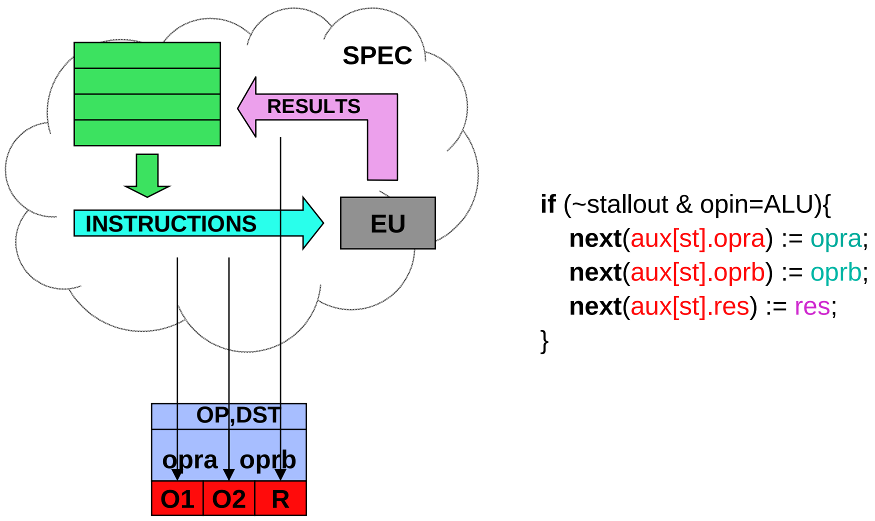

Figure 6: Auxiliary state derived from abstract model.

contained in different layers. Thus, we can uniquely identify a property to be proved by specifying the name of the signal assigned, and the name of the layer containing the assignment.

As an example, here is the SMV specification for the operand correctness layer (for the `opra` operand):

```
forall(k in TAG)
  layer lemma1 :
   if(st[k].valid & st[k].opra.valid)
     st[k].opra.val := aux[k].opra;
```

Here, `TAG` is the type of reservation station indices. Thus, for all reservation stations, if the station is `valid` (contains an instruction) and its `opra` operand is a value (not a tag), then the value must be the correct operand value that we stored in the auxiliary array `aux`. The result correctness lemma is just as simply stated:

```
forall (i in TAG)
  layer lemma2[i] :
   if(pout.tag = i & pout.valid)
     pout.val := aux[i].res;
```

It says that, for all reservation stations i, if the tag of the returning result on the bus `pout` is i, and if the result is valid, then its value must be the correct result value for reservation station i, as stored in the auxiliary array `aux`. Note that in this case we have defined an array of refinement relations, one for each reservation station. All specify the same signal, `pout.val`.

Also note that in these two refinement relations, the only implementation signals that we refer to are the reservation station operands, the reservation station valid bits, and the result bus. There is no reference in the proof, for example, to the register file, the issue logic, the execution units or the completion logic. As a result, this proof is likely to be more robust with respect to small design changes than a proof that references many signals. We will observe this later when introducing a "re-order buffer" to the design.

6.3 Path splitting

Although the refinement relations divide the implementation into two parts for the purpose of verification, there are still large structures in the model that prevent us from applying model checking at this point. These are the register file array, the reservation station array and the execution unit array. Therefore, we break the verification problem into cases, as a function of the particular path a data item takes when moving from one refinement relation to another.

Consider, for example, a value returning on the result bus. This result was produced by some reservation station i (which we will call the producer). It then (possibly) gets stored in a register j. Finally it is read as an operand for reservation station k (which we will call the consumer). This suggests a case split which will reduce the size of the verification problem for operand correctness to just two reservation stations and one register. For each operand arriving at consumer reservation station k, we split cases based on the producer reservation station i that it came from (this is indicated by the "tag" of the operand) and on the register j that it passed through (this is the source operand index for the given instruction). To prove this one case, we use just reservation stations i and k, and register j. The other elements of these arrays are abstracted away by assigning them undefined values. This situation is depicted in figure 7.

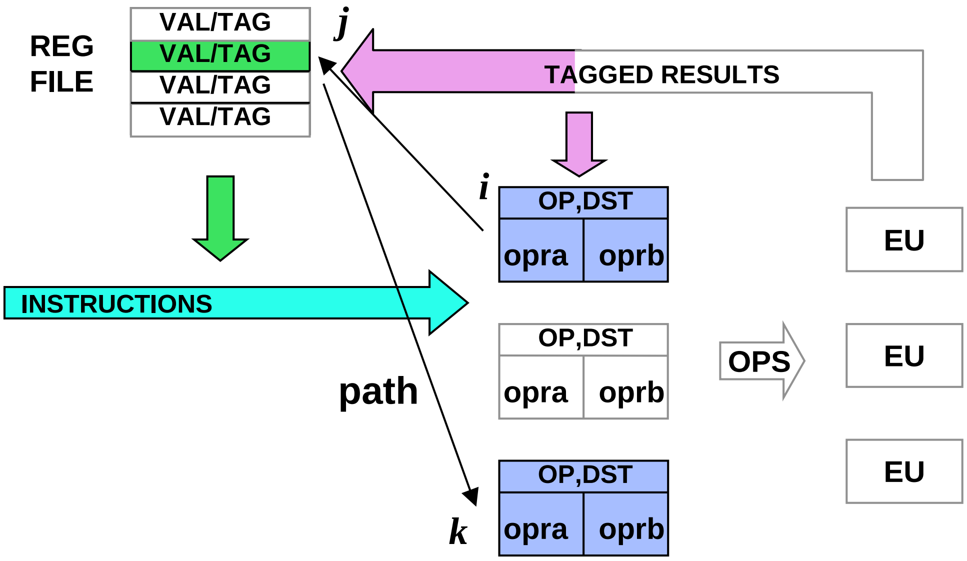

Figure 7: Path splitting in Tomasulo's algorithm.

To apply temporal case splitting in SMV, we use the following declaration (for the `opra` operand):

```
forall (i,k in TAG; j in REG)
   subcase lemma1[i][j]
   of st[k].opra.val//lemma1
   for st[k].opra.tag = i & aux[k].srca = j;
```

That is, for all consumer reservation stations k, we break the operand correctness lemma into an array of cases (i, j), where i is the producer reservation station and j is the source register. Note that we have to add an auxiliary variable to remember source operand register `srca`, since the implementation does not store this information. Verifying each case requires only one register and two reservation stations in the model. Thus, we have effectively broken the large data structures down into their components for verification purposes. For the result lemma a similar case splitting declaration can be specified; we split cases on the producing reservation station of the result on the bus, and the execution unit that computed it.

6.4 Exploiting symmetry

To verify the operand correctness lemma, we now have one case to prove for each triple (i, j, k) where i, k are reservation stations and j is an element of the register file. This could be a very large number indeed if, for example, there are 64 registers and 64 reservation stations. However, if all the registers are symmetric to on another, and all the reservation stations are similarly symmetric, then two representative cases will suffice: one where $i = k$ and one where $i \neq k$. To exploit the symmetry of the design in this way in SMV, we declare the types of register indices and reservation station indices to be scalarsets:

```
scalarset REG 0..63;
scalarset TAG 0..63;
```

When these declarations are made, the SMV proof system automatically chooses a set of representative cases under symmetry. In this instance it chooses the cases $(i = 0, j = 0, k = 0)$ and $(i = 0, j = 0, k = 1)$. All of the remaining cases reduce to one of these by permuting the scalarset types. Thus, we have reduced $O(n^3)$ cases to just two.

Note that declaring types to be scalarsets means that we can use values of these types as array indices, and we can compare them for equality, but we can't perform any other operations on them if we want to make use of symmetry reduction for these types. Fortunately, these are (with one exception) the only operations that are required on the index types, as no actual computation is performed on them. The exception is the allocation of empty reservation stations for incoming instructions. This will typically include some form of priority encoder, which will break the symmetry of reservation station indices. However, for purposes of verifying the operand correctness lemma, we do not need to use this logic in the environment – a completely nondeterministic allocation policy will suffice. Hence we do not break the symmetry when verifying the lemma. In general, when applying symmetry reduction to a particular type, SMV automatically excludes any assignments that break the symmetry of the type from the environment. This guarantees that the proof subgoal we are checking obeys the scalarset type rules, and hence that the symmetry reduction is sound.

6.5 Infinite-state verification

Up to this point we have defined refinement relations, used path splitting to decompose the large structures, and applied symmetry to reduce the number of cases to a tractable level. There remains, however, the issue of large types, in this case, the data values and possibly the index types. To handle these, we use a data type reduction to reduce the types to small sets consisting a few "interesting" values (for the particular case we are proving) and an abstract value to represent the rest. In fact, using data type reduction, we can verify our implementation for an arbitrary (or infinite!) number of registers and reservation stations. To do this, we simply declare the index types to be scalarsets with undefined range, as follows:

```
scalarset REG undefined;
scalarset TAG undefined;
```

When verifying the operand correctness lemma, for example, for a given case (i, j, k), SMV will automatically reduce the type TAG to just three values: i, k and an abstract value to represent the rest. Similarly, the type REG will be reduced to just two values: j and an abstract value. As a result, in the model checker, only two bits will be used to represent tags (enough to represent 3 values), while one bit will be used to represent register indices.

Note that data type reductions can also be specified manually, but in the present case the default reductions are adequate, so no action needs to be taken by the user.

6.6 Uninterpreted functions

Finally, we come to the question of data values. Suppose, for example that the data path is 64 bits wide. Although model checkers can handle some arithmetic operations (such as addition and subtraction) at this width, they cannot typically handle others (such as multiplication). Moreover, it would be better to verify our implementation of Tomasulo's algorithm generically, independent of the arithmetic operations implemented by the instruction set. We can do this by introducing an *uninterpreted function symbol* f for the ALU function. Assuming only that the abstract model and the implementation execution units compute the same function f, we should be able to prove that our implementation is correct regardless of the ALU function. Using an uninterpreted function symbol also has the advantage that the symmetry of data values is not broken. Thus, we can apply symmetry reductions to data values, and as a result, consider only a few representative cases of data values rather than all 2^{64} possible values.

To introduce an uninterpreted function in SMV, we simply declare an array to represent the lookup table of the function. Thus, if `a` and `b` are two operand values, then `f[a][b]` is the result. Since the actual contents of the array are unspecified (except that the contents do not change over time), all possible functions f are represented in this way. We model ALU operations in both the abstract model and implementation with lookups in the array `f`.

With this abstraction, we can easily deal with any size data word, by declaring the type of data words to be a scalarset. In fact, we can leave the actual range of the type undeclared, so that in principle we are verifying the implementation for any size data type. We then use case splitting on the data values to reduce the problem to a finite state problem. In particular, we verify the result correctness lemma for only the case when the operands are some particular values a and b, and where the result f[a][b] is some particular value c. Since we have three parameters a, b and c of the same type, the number of cases we require to have a representative set is just $3! = 6$. Here is the declaration we use in SMV to split the problem into cases:

```
forall(i in TAG; a,b,c in WORD)
   subcase lemma2[i][a][b][c]
   of pout.val//lemma2[i]
   for aux[i].opra = a & aux[i].oprb = b & f[a][b] = c;
```

Given this declaration, SMV will automatically apply symmetry reduction to reduce an infinite number of cases to just 6 representative ones. In addition, it will automatically reduce the (possibly infinite) type of data words to just the specific values a, b and c, and an abstract value. Thus, in the worst case, when a, b and c are all different values, the number of bits required to encode data words for the model checker is two. Thus, we have reduced an infinite state verification problem to a finite number of finite state problems.

6.7 Summary

The result of applying the above described proof decomposition is a set of proof subgoals that can be solved by model checking (and which, if false, will yield sequential counterexamples). Our implementation of Tomasulo's algorithm is verified for an arbitrary (finite or infinite) number of registers and reservation stations, for an arbitrary (finite or infinite) size data word, and for an arbitrary ALU function. It is also possible to use similar techniques to verify it for an arbitrary number of execution units (this requires one "noninterference lemma", which is not discussed here).

All told, there are 11 cases of the various lemmas to prove: two cases each for the `opra` and `oprb` operand correctness, 6 cases for the result correctness, and one case for the correctness of the data output (not discussed here). For the case of one execution unit, the largest model checking problem has 25 state variables. As a result, the overall verification time (including the generation of proof goals and model checking) is just under 4 CPU seconds (on SPARC Ultra II server). The verification time for 8 execution units is roughly one minute. The time required for an experienced user of SMV (the author) to write, debug and verify the proof was approximately an hour and ten minutes (the design itself was already debugged and was previously formally verified using an earlier methodology). [1]

In summary, the basic strategy we used to reduce the verification problem to tractable model checking problems was the following:

1. **Refinement relations and auxiliary state**. We broke the problem into two parts, by writing refinement relations that specify the correct values for the operands and results obtained in the implementation. To do this, the correct values are obtained from the abstract model, and stored in auxiliary state.

2. **Path splitting**. We broke the large data structures (the register file and reservation station array) down into just a few components by splitting cases on the path taken by a data item from one refinement relation to another.

3. **Symmetry**. The large number of cases produced by the above two steps are reduced to a small finite number by considerations of symmetry.

4. **Data type reductions**. After case splitting, we reduced the large (or infinite) types, such as data words, to small finite types, representing all the irrelevant values by a single abstract value. A special case of this is the uninterpreted function abstraction, in which we use a lookup table to represent and arbitrary function, then split cases such that we use only one element of the table for each case.

6.8 Adding a re-order buffer

Now, suppose that we modify the design to use a "re-order buffer". This means that instead of writing results to the register file when they are produced by an execution unit, we store them in a buffer, and write them back to the register file in program order. This is usually done so that the processor can be returned to a consistent state after an "exceptional" condition occurs, such as an arithmetic overflow. The simplest way to do this in the present implementation is to store the

[1] Details of this example (and extending the proof to an arbitrary number of execution units) can be found in a tutorial on SMV, included with the SMV software. At the time of this writing, the software and tutorial can be downloaded from the following URL:

`http://www-cad.eecs.berkeley.edu/~kenmcmil/smv`.

result in an extra field `res` of the reservation station, and then modify the allocation algorithm so that reservation stations are allocated and freed in round-robin order. The result of an instruction is written to the register file when its reservation station is freed.

Interestingly, after this change, the processor can be verified without modifying one line of the proof! This is because our three lemmas (for operands, results and noninterference) are not affected by the design change. This highlights an important difference between the present methodology and techniques such as [BD94, JDB95, HB95, SJD98, VB98a, BBCZ98], which are based on symbolic simulation. Because we are using model checking it is not necessary to write inductive invariants of the design. Instead, we rely on model checking to compute the strongest invariant of an abstracted model. Thus, our proof only specifies the values of three key signals: the source operands in the reservation stations, the value on the result bus and the tag on the result bus. Since the function of these signals was not changed in adding the re-order buffer, our proof is still valid. On the other hand, if we had to write inductive invariants, these would involve in some way all of the state holding variables. Thus, after changing the control logic and adding data fields, we would have to modify the invariants. Of course, in some cases, such as very simple pipelines, almost all states will be reachable, so the required invariant will be quite simple. However in the case of a system with more complex control (such as an out-of-order processor), the invariants are nontrivial, and must be modified to reflect design changes. While this is not an obstacle in theory, in practice a methodology that requires less proof maintenance is a significant advantage.

In the next section we will consider applications of the same basic strategy to other hardware examples, with somewhat different characteristics.

7 Other applications

7.1 A packet buffering application

The first example is a component of the InfoPad wireless terminal [TPDB98] developed at the University of California. In this hand-held device, a variety of multimedia data sources and sinks, including a pen input device, speech input and output, a video stream a graphics, are connected via an 8-bit bus. A component called the "transmit multiplexer", implemented as an ASIC, is used to consolidate byte-at-a-time transfers over the multiplexed bus into complete packets, sending the completed packets to an FPGA for encoding and eventual transmission by a wireless modem. Truman [Tru98] reports on the specification and verification of a model of this buffering device, using compositional model checking and a refinement based methodology. An earlier version of SMV was used for this purpose. Here we consider not the specific proof in that paper, but how the present methodology might be applied to the transmit buffer, and similar packet storage and forwarding devices. Nonetheless, the important ideas in the proof are due to Truman.

In this case, the abstract model is extremely simple. It consists simply of a stream of arbitrary packets to be sent by each each source device. The streams are represented by arrays, whose content is undefined, except that it is constant in time. No operation is performed on these arrays, since the system only transfers data and does not perform any actual computation. Given that packet streams are presented appropriately at the input of the transmit buffer, we wish to show that the same data appears in an appropriate manner at the output. The manner of presentation of data at the input and output is specified by refinement relations. This situation is depicted in figure 8.

In order to specify the refinement relations for input and output, we have to know at any given time which data item from the abstract packet streams is currently being transferred across the given interface. This information is obtained by attaching a tag to each data item. The tags are auxiliary variables, not part of the design, but they travel through the design along with the

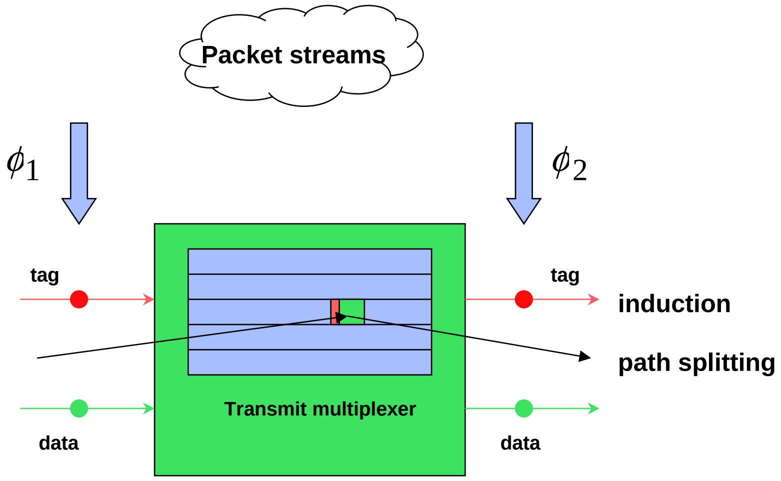

Figure 8: Path splitting in the InfoPad transmit multiplexer

corresponding data, exactly as if they were part of the real hardware. A tag indicates for a given data item exactly which byte of which packet of which stream the data item derives from. Thus, given the tag and the abstract model, we can always determine the correct value of the data item. Note that, while in the previous example, we used the auxiliary state to store actual data values from the abstract model, in this case, we store pointers into the abstract model instead.

Now we come to the proof decomposition. There are two large structures that we have to consider when decomposing the problem. The first is the set of abstract packet arrays. The second is the buffer memory within the transmit multiplexer. This large space of buffer memory is used to store packets from the various sources while they are being accumulated and waiting for transmission. As before, we can use path splitting to break these large structures into their respective components. When verifying that a data item at the output is correct according to its tag, we first of all consider only the case of one particular tag i. This corresponds to one particular data item in the abstract model. Hence, all of the other items can be abstracted away in the proof. Second, we consider only data items which have been stored in one particular cell of the buffer memory. As before, this means we may have to add some auxiliary state to remember the storage location of the item currently being output. With this reduction, we can consider only one cell of the buffer memory and one data item in the abstract packet stream. By declaring the index types for packet streams and the buffer array to be symmetric, we can reduce the number of cases to prove down to just one. Thus we have reduced a problem of arbitrary size to a problem of fixed finite size.

Having shown that all data items reaching the output of the transmit multiplexer are correct, we also have to show that the items appear in the correct order. Note that this property relates only to the tags and not to the data items themselves (this specification is also part of the refinement relations). Provided that the range of the tags is not too large, this proof can be done directly by model checking. This is because the number of states in the model, when data are excluded, is proportional simply to the number of tags. Using induction (not discussed here) we can also show correct ordering for infinite streams of packets.

7.2 A cache coherent multiprocessor

Our final example is a cache coherence protocol of a commercial multiprocessor, and its implementation in an ASIC. The design and formal verification of the protocol, and the portion of the ASIC that implements it, is reported by Eiriksson [Eir98]. This work was also done using an earlier version of SMV. Again, the purpose here is not to describe Eiriksson's proof, but to show how in outline the proof fits into the present methodology.

The system in question consists of a collection of processors sharing a common memory address space. Each processor stores a working set of memory locations in a local cache. Consistency between these locally cached copies of memory locations is maintained by a protocol, which sends messages over an unordered store-and-forward network. The protocol itself was verified by model checking, using an abstract model of the system. In this model, receiving a message and transmitting a response is modeled as a single atomic event. This abstract model, once verified, is used as the basis to specify and verify the implementation, by means of refinement relations. This approach is depicted in figure 9.

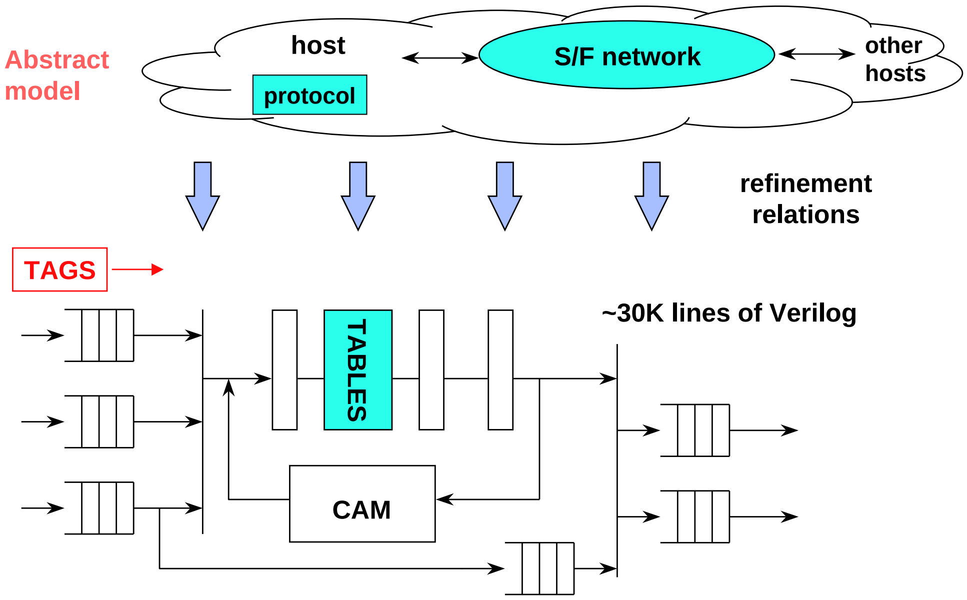

Figure 9: Cache protocol refinement verification.

The implementation itself (described at the RTL level in the Verilog language) is substantially more complex than the abstract model. The design contains, for example, a collection of FIFO buffers to handle incoming and outgoing messages, arbitration logic to handle the routing of messages onto buses, a pipeline to handle the processing of protocol state transitions, a content addressable memory (CAM) to store the state of pending transactions by address, an intermediate storage memory for holding data associated with pending transactions, and so on. All told the protocol is implemented in roughly 30,000 lines of synthesizable Verilog

The implementation logic is verified by writing refinement relations that relate values at certain points in the implementation with values in the abstract model. Once again, to specify the correct value of data items at any point in the design, we add auxiliary state to the model. The auxiliary variables in this case are tags that indicate which protocol message in the abstract model a given data item is associated with. These tags move through the system along with the corresponding data. This makes it possible to specify, at a given time and place, what the correct value of a given data item is. This in turn allows us to break a large design into pieces of tractable size for verification, and verify each piece in the context of the abstract model.

As in the previous example, there are also several large data structures that need to be decomposed by path splitting. The most important of these is the memory itself, which we divide into its elements by considering the correctness of only those transactions relating to one particular memory address. Similarly, to handle large FIFO buffers, we consider the correctness of only data items that pass through a particular cell of the circular buffer that implements the FIFO. To handle the CAM that holds the state of pending transactions, we consider only those transactions that fall into a particular cell of the CAM, and so on. As before, symmetry considerations then can be used to reduce the number of proof obligations to a tractable level. For example, the memory addresses are symmetric, as are CAM indices, and the message buffers in the abstract model that represent the store-and-forward network. Data type reductions can then be used to reduce the large types, such as addresses, to a small number of values, for any given case.

Thus the cache coherence system is verified by applying the same basic proof methodology as in the previous examples.

8 Related work

This work is chiefly concerned with combining a set of proof techniques into a methodology for system level formal verification in the hardware domain. Many of the individual techniques used in this methodology are related to techniques in the literature, and we consider some of these connections here.

We begin with the circular compositional method. Compositional methods in the temporal framework were originally described in terms of an "assumption/guarantee" paradigm [Pnu84]. Each component of a system was to be specified by those temporal properties which it assumes about its environment, and those temporal properties which it will guarantee, provided the assumptions hold. Thus, if a given process assumes P and guarantees Q, then if we compose it with a process that guarantees P (under no assumptions), we can infer Q. This is essentially no different from Hoare logic, except in that the formulas are temporal, and composition of programs is parallel rather than sequential. However, this temporal aspect introduces a serious difficulty. That is, when composing two processes A and B, it is often necessary to assume correctness of A to verify B and *vice versa*. This is not a problem in Hoare logic, since for a sequential program, we do not require that the guarantees made by A and B hold simultaneously. The inference rule for the "while" statement in effect allows us to construct circular proofs. In the parallel case, however, both specifications A and B must hold simultaneously, and thus a proof may not be circular. In practice, we may easily find that there is no place to begin a non-circular assume/guarantee proof, because no process satisfies any useful properties in the absence of environment assumptions.

This "environment" problem was recognized by Abadi and Lamport [AL93]. They showed the somewhat surprising fact that under certain restrictions on the processes and on the temporal properties to be proved, if one can prove P assuming Q and prove Q assuming P, then both P and Q must hold. To simplify somewhat, these restrictions were essentially as follows: 1) the processes must modify disjoint subsets of the system variables in an interleaved manner, and 2) each property must be a safety property and restrict only state variables of one process. Given these assumptions, one can show that the truth of P at time t cannot depend on Q holding at any time later than $t-1$, and *vice versa*. Thus, P and Q are proved in effect by mutual induction over time. This argument would not hold, however, if either P or Q contained operators that refer to the future.

The technique is thus restricted to safety properties (in temporal logic, the properties expressible in the form Gp, where p is a past-time formula). Alur and Henzinger [AH96] extended this approach to the case where the processes and properties are both represented by synchronous Mealy machines (a method embodied in the Mocha system [AHM$^+$98]). This relaxes the requirement of interleaving concurrency, but still allows only safety properties to be verified.

Nonetheless, mutual dependence of liveness properties does occur in real systems. Consider, for example, the problem of multiple execution units in an instruction set processor. At some times, the instruction in unit A may depend on the result of the instruction in unit B, and at other times the inverse relation may hold. Thus, in order to prove that unit A is live (always eventually produces a result), we must assume the B is live, and *vice versa*. An example of such a liveness proof can be found in [McM99a]. The technique of circular compositional proofs used here allows for this kind of circular reasoning, even about liveness properties. In essence, it makes explicit the induction over time implied in the above approach, by assuming property P *only* up to time $t-1$ when proving Q at time t, and *vice versa*. As we have seen, since this condition (Q up to $t-1$ implies P up to t) is expressible in temporal logic, the local proof subgoals generated by a circular compositional proof can be verified by model checking.

The technique of temporal case splitting can be viewed as simply a temporal version of the "excluded middle" principle. Thus it undoubtedly has many precedents in earlier systems. What is possibly new about this technique is that, rather than splitting a proof into cases based on the truth of a proposition, one splits into cases based on the set of *times* at which a variable has a particular value. More particularly, the notion of splitting up a proof into cases based on the path a data item takes through a system appears to be novel, at least insofar as its application to model checking is concerned.

The use of symmetric types to infer symmetry in a design has its roots in the Murphi verification system [ID96]. However, in Murphi, the symmetry is used to reduce the size of the state space, while here it is used to reduced the number of cases of a parameterized property that must be verified.

The technique of data type reduction presented here is a special case of *abstract interpretation* [CC77]. It is related to reductions based on *data independence* [Wol86] in that large data types are reduced to small finite ones by using a few specific values and an extra value to represent the remaining elements of the type. However, the technique does not require control to be independent of data. For example, it allows control to depend on comparisons of data items. This is significant, in that the technique can be applied to reduce addresses, tags and pointers, which are commonly compared to determine control behavior. Also, the technique reduces not only the data types in question, but also any arrays indexed by these types. This makes it possible to reduce systems with large or unbounded memory arrays, register files, FIFO buffers, *etc.*, to a small finite size.

Lazic and Roscoe [LR98] also describe a technique for reducing systems with unbounded arrays to finite systems for verification, under certain restrictions. Their technique is a complete procedure for verifying a particular property (determinism). It works by identifying a finite configuration of a system, whose determinism implies determinism of of any larger configurations. The technique presented here, on the other hand, is not restricted to a particular property or class of properties. More importantly, the method of [LR98] does not allow equality comparison of values stored in arrays, nor the storage of array indices in arrays. Thus, it cannot be applied to, for example, unbounded cache memories, content-addressable memories, or the example presented here, an out-of-order processor that stores tags (*i.e.*, array indices) in arrays, and compares them for equality. It should be noted, in fact, that comparing values stored in an unbounded array, or even including one bit of status information in elements of an unbounded array, is sufficient to make reachability analysis undecidable. Unfortunately, these conditions are ubiquitous in hardware design. Thus, while the technique presented here is incomplete, being based on a conservative abstraction, this incompleteness should be viewed as inevitable if we wish to verify hardware designs for unbounded resources.

Data type reduction has also been described previously by Long [Lon93] in his work on automatically generating abstractions. However, that work applied only to concrete finite types. Here, data type reductions are used in combination with symmetry and case splitting, to allow the reduction of types of arbitrary or infinite size to finite types. In addition, Long's work did not treat the reduction of arrays. What makes it possible to do this with the present technique is the combination of data type reductions with temporal case splitting and symmetry reductions, a combination which appears to be novel.

The use of uninterpreted functions here is also substantially different from previous applications. The basic reason for using uninterpreted functions is the same – to abstract away from the actual functions computed on data in order to reason separately about arithmetic and data flow. However, existing techniques using uninterpreted functions, such as [BD94, JDB95, HB95, SJD98, VB98a, BBCZ98] are based essentially on symbolic simulation. In these methodologies, one attempts to prove a commutative diagram. In the simplest case, one shows that, from any state, applying an abstraction function and then a step of the specification model is equivalent to applying a step of the implementation model and then the abstraction function. However, since not all states are reachable, in general the user must provide an *inductive invariant* on the state space. The commutative diagram is proved only for those states satisfying the invariant. By contrast, in the present technique, uninterpreted functions are used in the context of *temporal verification* (*i.e.*, model checking). In this case, there is no need to provide an inductive invariant, since the model checker is proving that a given temporal property holds at the initial state, rather than that a commutative diagram holds at every state. In effect, we are relying on the model checker's ability to compute the *strongest invariant* (*i.e.*, the set of reachable states) rather than supplying an invariant by hand. Note that in general, when uninterpreted functions with equality are added to temporal logic, the resulting logic is undecidable. The technique presented here is not a decision procedure for such a logic, but rather a technique of reduction to propositional temporal logic that is necessarily incomplete. Others have presented a semi-decision procedure for such a logic [HIKB96], however this technique is sound only in a very restricted case; for most problems of practical interest, the procedure is not sound, and can only be used as a heuristic to find counterexamples.

Finally, a number of authors report the use of general purpose proof assistants, without model checking, in processor verification (for example [Cyr96, VB98b, WAH94, SWAH98]).

To conclude, the methods presented here are novel in several aspects: first the particular methods of circular compositional proof, symmetry reduction, and data type reduction and the method of handling uninterpreted functions are novel in and of themselves. Second, the combination of these techniques into a methodology for hardware verification is novel. Finally, the implementation of all these techniques into a mechanical proof system based on symbolic model checking is novel.

9 Conclusion

We have seen that a proof strategy based on a few simple, somewhat domain specific proof techniques can be used to break the verification of a large and complex hardware system down into a tractable number of finite state subgoals, with few enough state variables to be verified by model checking.

This strategy appears to be reasonably generic, in that it can be applied, for example, to such diverse hardware applications as out-of-order instruction set processors, cache coherence systems and packet handlers for communication systems. It allows these designs to be verified at both an abstract level, and at the "RTL" level. From this level, designs may be either synthesized directly into logic gates, or compared to gate level designs using combination equivalence tools [KSL95]. The methodology has in practice been applied to a very large design in a commercial design environment [Eir98]. Nonetheless, there is no evidence to indicate that application of the methodology to additional designs will not reveal weaknesses which would require the incorporation of additional proof techniques. For example, the notion of data type reduction has already been extended to support an inductive data type, which allows incrementation (*i.e.*, a successor function) as well as equality comparison. This can be used, for example, to show that a FIFO buffer delivers an infinite sequence of packets in the correct order. Thus, the system described here should be viewed as a work in progress.

In essence, the methodology presented here is an attempt to combine in a practical way the strengths of model checking and theorem proving. The refinement relation approach, combined with the various available reductions and model checking, makes it possible to avoid writing detailed assertions about every state holding component of the design, and also to avoid interactively generated proof scripts. In this way, the manual effort of proofs is reduced. In particular, the model checker's ability to compute the strongest invariant of a model is pervasively exploited to avoid writing inductive invariants of the system by hand. The proofs thus obtained, involving less design detail, are arguably less vulnerable to small changes in the design. This is illustrated by the example of adding a re-order buffer to an implementation of Tomasulo's algorithm, while reusing without modification the original proof. On the other hand, the ability of theorem proving to break large proofs down into smaller ones is exploited to avoid model checking's strict limits on model size. Thus, by combining the strengths of these two methods, we may arrive at a scalable methodology for formal hardware verification.

References

[AH96] R. Alur and T. A. Henzinger. Reactive modules. In *11th annual IEEE symp. Logic in Computer Science (LICS '96)*, 1996.

[AHM+98] R. Alur, T. A. Henzinger, F.Y.C. Mang, S. Qadeer, S. K. Rajamani, and S. Tasiran. Mocha: Modularity in model checking. In A. J. Hu and M. Y. Vardi, editors, *Conference on Computer-aided Verification (CAV '98)*, number 1427 in LNCS, pages 521–25. Springer-Verlag, 1998.

[AL93] M. Abadi and L. Lamport. Composing specifications. *ACM Trans. on Prog. Lang. and Syst.*, 15(1):73–132, Jan. 1993.

[BBCZ98] S. Berezin, A. Biere, E. Clarke, and Y. Zhu. Combining symbolic model checking with uninterpreted functions for out-of-order processor verification. In G. Gopalakrishnan and P. Windley, editors, *Formal Methods in Computer-Aided Design (FMCAD '98)*, number 1522 in LNCS, pages 351–68. Springer, 1998.

[BD94] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *Computer-Aided Verification (CAV '94)*. Springer-Verlag, 1994.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th POPL*, pages 238–252. ACM Press, 1977.

[Cyr96] D. Cyrluk. Inverting the abstraction mapping: a methodology for hardware verification. In M. Srivas and A. Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD '96)*, volume 1166 of *LNCS*. Springer-Verlag, 1996.

[Eir98] A. Eiriksson. Formal design of 1M-gate ASICs. In G. Gopalakrishnan and P. Windley, editors, *Formal Methods in Computer-Aided Design (FMCAD '98)*, number 1522 in LNCS, pages 49–63. Springer, 1998.

[HB95] R. Hojati and R. K. Brayton. Automatic datapath abstraction of hardware systems. In P. Wolper, editor, *Conference on Computer-aided Verification (CAV '95)*, number 939 in LNCS, pages 98–113. Springer-Verlag, 1995.

[HIKB96] R. Hojati, A. Isles, D. Kirkpatrick, and R. K. Brayton. Verification using uninterpreted functions and finite instantiations. In M. Srivas and A. Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD '96)*, volume 1166 of *LNCS*, pages 218–32. Springer, 1996.

[ID96] C.N. Ip and D.L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1-2):41–75, Aug. 1996.

[JDB95] R. B. Jones, D. L. Dill, and J. R. Burch. Efficient validity checking for processor verification. In *IEEE/ACM Int. Conf. on Computer Aided Design (ICCAD '95)*, 1995.

[KSL95] A. Kuehlmann, A. Srinivasan, and D. P. LaPotin. Verity – a formal verification program for custom CMOS circuits. *IBM J. of Research and Development*, 39(1–2):149–65, Jan.–Mar. 1995.

[Lon93] D. E. Long. Model checking, abstraction, and compositional verification. Tecnical report CMU-CS-93-178, Carnegie Mellon School of Computer Science, July 1993. Ph.D. Thesis.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Twelfth Annual ACM Symp. on Principles on Programming Languages*, January 1985.

[LR98] R. S. Lazić and A. W. Roscoe. Verifying determinism of concurrent systems which use unbounded arrays. Technical Report PRG-TR-2-98, Oxford Univ. Computing Lab., 1998.

[McM93] K. L. McMillan. *Symbolic Model Checking.* Kluwer, 1993.

[McM99a] K. L. McMillan. Circular compositional reasoning about liveness. http://www-cad.eecs.berkeley.edu/˜kenmcmil/papers/1999-02.ps.gz, February 1999.

[McM99b] K. L. McMillan. Verification of infinite state systems by compositional model checking. http://www-cad.eecs.berkeley.edu/˜kenmcmil/papers/1999-01.ps.gz, February 1999.

[OG76] S. Owicki and D. Gries. Verifying properties of parallel programs. *Comm. ACM*, 19(5):279–85, May 1976.

[Pnu84] A. Pnueli. In transition from global to modular temporal reasoning about programs. In *Logics and Models of Concurrent Systems*, NATO ASI Series, pages 123–144. Springer-Verlag, 1984.

[Sha96] N. Shankar. PVS: Combining specification, proof checking and model checking. In M. Srivas and A. Camilleri, editors, *Formal Methods in Computer-Aided Design (FM-CAD '96)*, volume 1166 of *LNCS*, pages 257–64. Springer, 1996.

[SJD98] J. U. Skakkabaek, R. B. Jones, and D. L. Dill. Formal verification of out-of-order execution using incremental flushing. In A. J. Hu and M. Y. Vardi, editors, *Conference on Computer-aided Verification (CAV '98)*, number 1427 in LNCS, pages 98–109. Springer-Verlag, 1998.

[SWAH98] J. Sawada and Jr. W. A. Hunt. Processor verification with precise exceptions and speculative execution. In A. J. Hu and M. Y. Vardi, editors, *Conference on Computer-aided Verification (CAV '98)*, number 1427 in LNCS, pages 135–46. Springer-Verlag, 1998.

[Tom67] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM J. of Research and Development*, 11(1):25–33, Jan. 1967.

[TPDB98] T. Truman, T. Pering, R. Doering, and R. Broderson. The InfoPad multimedia terminal: A portable device for wireless information access. *IEEE Trans. on Computers*, 1998. to appear.

[Tru98] T. E. Truman. *A Methodology for the Design and Implementation of Communication Protocols for Embedded Wireless Systems.* PhD thesis, Dept. of EECS, University of CA, Berkeley, May 1998.

[VB98a] M. Velev and R. E. Bryant. Bit-level abstraction in the verification of pipelined microprocessors by correspondence checking. In G. Gopalakrishnan and P. Windley, editors, *Formal Methods in Computer-Aided Design (FMCAD '98)*, number 1522 in LNCS, pages 18–35. Springer, 1998.

[VB98b] M. Velev and R. E. Bryant. Verification of data-intensive circuits: An in-order-retirement case study. In G. Gopalakrishnan and P. Windley, editors, *Formal Methods in Computer-Aided Design (FMCAD '98)*, number 1522 in LNCS, pages 351–68. Springer, 1998.

[VW86] M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *First Annual Symp. on Logic in Comp. Science*, June 1986.

[WAH94] Jr. W. A. Hunt. *FM8501: A verified microprocessor.* Number 795 in LNCS. Springer-Verlag, 1994.

[Wol86] P. Wolper. Epressing interesting properties of programs in propositional temporal logic. In *13th ACM POPL*, pages 184–193, 1986.