Verification of Infinite State Systems by Compositional Model Checking

K. L. McMillan
mcmillan@cadence.com

Cadence Berkeley Labs
2001 Addison St.
Berkeley, CA 94704
USA
email: mcmillan@cadence.com

Abstract. Compositional model checking methods can be used to reduce the formal verification of a complex system to model checking problems of tractably small size. However, such techniques are difficult to apply to systems that have large data types, such as memory addresses, or large data arrays such as memories or FIFO buffers. They are also limited to the verification of systems with fixed finite resources. In this paper, a method of compositional verification is presented that uses the combination of *temporal case splitting* and *data type reductions* to reduce types of unbounded range to small finite types, and arrays of unbounded size to small fixed-size arrays. The method also supports the use of *uninterpreted functions* in a novel way, that allows model checking to be applied to systems with uninterpreted functions. These techniques are implemented in a proof assistant that also supports compositional reasoning and reductions via symmetry. Application of the method is illustrated by using this tool to verify an implementation of Tomasulo's algorithm. The use of temporal case splitting, data type reductions and and uninterpreted functions substantially simplifies the proof for fixed finite configurations, and also allows a generic verification for configurations of arbitrary word size, arbitrary register file size, arbitrary number of reservation stations and arbitrary number of execution units.

©1999 Cadence Berkeley Labs, Cadence Design Systems.

1 Introduction

Compositional model checking methods are used to reduce the verification of a large complex system to a number of smaller, localized verification subgoals that can be effectively discharged by model checking. Such methods are necessary because of the limitations of model checking with respect to state space size. Compositional techniques are implemented in, for example, the SMV system [McM98] and the Mocha system [AHM+98]. The typical proof strategy in these systems is to specify *refinement relations* between an abstract model and certain variables or signals in the implementation. This allows components of the implementation to be verified in context of the abstract model. This technique, while effective, is limited in two respects. First, it does nothing to reduce very large data types, such as addresses or data words. This makes basic compositional model checking methods inapplicable to systems with for example, 32-bit or 64-bit memory address spaces. Second, it can verify only a fixed instance or configuration of a design, with fixed resources. It cannot, for example, verify a parameterized design for all values of the parameter (such as the number of elements in an array).

In this paper, we present a method based on *temporal case splitting* and a form of *data type reduction*, that makes it possible to deal with types of arbitrary or infinite range, and arrays of arbitrary or infinite size. Temporal case splitting is used to break the correctness of a given data item into cases based on the path the data item has taken through the system. For each case, we therefore need to consider only a small fixed set of elements of the large structures in the design. The number of cases, while potentially very large, can be reduced to a small number by existing techniques based on considerations of symmetry [McM98]. Finally, for any given case, a data type reduction can be used to reduce the large or infinite types to small finite types. The reduced types contain only a few values relevant to the given case, and an abstract value to represent the remaining values in the original type. Thus, in effect, we reduce the large or infinite types and structures to small finite types and structures for the purpose of model checking.

Taken together, these methods also allow specification and verification using uninterpreted functions. This makes it possible, for example, to use model checking to verify the correctness of an instruction set processor independent of the arithmetic functions it provides, or, put another way, to separate the verification of data flow in the processor from the verification the the arithmetic units.

The techniques described in this paper have been implemented in a (non-interactive) proof assistant which generates model checking subgoals to be discharged by the SMV model checker.

1.1 Related work

The technique of data type reduction presented here is a special case of *abstract interpretation* [CC77]. In particular, 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 (an example of this will appear later in the paper). 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 in this paper, 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,VB98,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. Of the various non-temporal techniques using uninterpreted functions, the present method is most similar to [VB98], since it is also based on *finite instantiation*.

Thus, the methods presented here are novel in three aspects: first the methods of data type reduction and of handling uninterpreted functions are novel in and of themselves. Second, the combination of these techniques with existing compositional methods and techniques of exploiting symmetry is novel. Finally, the implementation of all these techniques into a mechanical proof assistant based on symbolic model checking is novel.

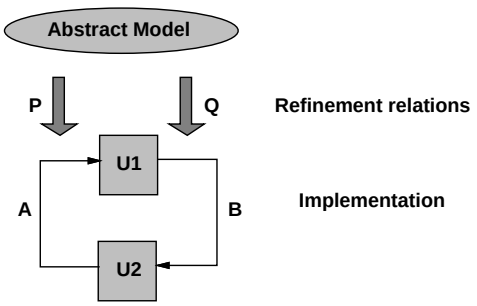

Fig. 1. Compositional refinement verification

1.2 Outline of the article

In section 2, we present a brief and informal overview of earlier work on compositional methods and symmetry, on which the current work is based. The next section describes the technique of temporal case splitting and its implementation in the SMV system. Section 4 then covers the data type reduction technique and its implementation in SMV. Finally in in section 5 the techniques are illustrated by applying them to an implementation of Tomasulo's algorithm. This is the same example used in [McM98], however in this case the new techniques substantially simply the proof, decrease the run-time of the prover, and allow the proof to be carried out for a design with unbounded resources.

2 Compositional verification and symmetry

The SMV system supports an approach to compositional model checking which is geared to proving that an abstract model, acting as the system specification, is implemented by some more detailed system model. The notion of correctness is usually defined in terms of *refinement maps* that relate signaling behavior at suitable points in the implementation with events occurring in the abstract model (see fig. 1). Typically, the abstract model, the implementation and the refinement maps are all expressed in the same HDL-like language, as sets of equations that may involve time delay. However, formally, we can view them as simply linear temporal logic properties.

The refinement maps decompose the system structurally into smaller parts for separate verification. To do this, we rely on a *circular compositional rule* which allows us to assume that one map (as a temporal property) holds true while verifying another map, and *vice versa*. Thus, we can assume that signal A is correct w.r.t. the abstract model when verifying signal B, and assume that signal B is correct w.r.t. the abstract model when verifying signal A. This makes it possible to compositionally very systems that have cyclic data flow, such as instruction set processors.

In addition, the SMV system can exploit symmetry in a design to reduce a large number of symmetric proof obligations to a small number of representative cases. This is based on the use of symmetric data types called *scalarsets*, borrowed from the Murphi language [ID96]. In order to exploit the symmetry of a given data type, we must guarantee that values of that type are only used in certain symmetric ways. In particular, they may be compared for equality, or used as indices of arrays, and finite arrays over scalarset types may be "summed" using any commutative/associative operator. SMV ensures that these conditions hold by applying appropriate type checking rules to programs. For further details of these methods, the reader is referred to [McM98].

3 Temporal case splitting

It is typical for hardware designs to contain large arrays, such as memories, FIFO buffers, content-addressable memories (CAM's) and so forth. These are problematic in that their state space is intractably large, thus we cannot apply model checking to them directly. However, we can often confine the verification problem to one or two components of the large structure by means of *temporal case splitting*. Using this approach, we verify the correctness of only those data items that have passed through a given fixed element of the array. This 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 each case accordingly.

Temporal case splitting breaks the proof of temporal property $G\phi$ (that is, ϕ at all times) 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. This inference is based on the following fairly trivial theorem about temporal logic:

Theorem 1. *If, for all i in the range of variable v, $\models G(v = i \Rightarrow \phi)$, then $\models G\phi$.*

Proof. Let $\sigma = s_0, s_1, \ldots$ be a sequence of states, and assume that for all i, $\sigma \models G(v = i \Rightarrow \phi)$. For any state σ_j, let v_j be the value of variable v in state σ_j. Since $\sigma \models G(v = v_j \Rightarrow \phi)$, it follows that ϕ holds in state σ_j. Hence $\sigma \models G\phi$.

Typically, v will be an auxiliary variable which stores the location in some large data structure that was used to store the data item currently appearing at a unit output (see [McM98] for the use of auxiliary variables in compositional proofs). To prove any give case $v = i$, it is typically only necessary to refer to element i of the array. The other elements of the array can be eliminated from the model by replacing them with an "unknown" value $\perp$, much as is done in ternary symbolic simulation [BS90]. Thus, we effectively decompose a large array into its individual components for the purposes of model checking. Several examples of this can be found in section 5.

In the SMV system, a specification `s` of the form Gp is split into cases using a declaration of the following form:

```
forall (i in TYPE)
  subcase c[i] of s for v = i;
```

Here, `s` is the name of the original specification, and `TYPE` is the type of variable `v`. This generates an array of specifications `c` where each specification `c[i]` is the formula $G((v = i) \Rightarrow p)$. If every element of `c` can be separately proved, then SMV infers the original specification `s`.

This technique can be extended to any set of cases q_i, such that $\models G \vee_i q_i$. We will call such a set of formulas a *complete* set of cases. In such a case, if $G(q_i \Rightarrow p)$ holds for all i, then we can infer Gp. There are several additional situations in which SMV can determine syntactically that a set of cases is complete. For example, let the formula $\phi @ \psi$ be an abbreviation for $\neg((\neg\psi)\ U\ (\psi \wedge \neg\phi))$. This formula is true at a given time if ϕ holds at the first future time when ψ holds (and is true if ψ never holds in the future). Clearly, the set of cases $q_i \equiv ((v = i)@\psi)$, for all i in the range if variable v, is complete. This is because either ψ never holds (in which case all the q_i are true) or it does eventually hold, at which time v must have some value i. SMV recognizes case splits of the form

```
forall (i in TYPE)
  subcase c[i] of s for (v = i) when psi;
```

to be complete. Here, `when` is used to represent the operator @. Splitting cases on the future value of a variable can be useful for liveness proofs.

4 Data type reductions

Although temporal case splitting can be used to reduce a large array down to a small number of elements, the model will typically still contain large data types, such as addresses, or data words. In this case, *data type reduction* can be used to reduce a large (perhaps unbounded or infinite) type to a small finite one, containing only one or two values that are relevant to the case being verified. The remaining values are represented by a single abstract value. Corresponding to this data abstraction, we have an abstract interpretation of operators in the logic. This abstraction is conservative, in that any property that is true in the reduced model is also true in the original model.

For example, let t be an arbitrary type. Regardless of the actual range of type t, we can reduce the range of t to a set containing some distinguished value i and an abstract value (which we will denote $t \setminus i$) representing the remaining values. Thus, in the reduced model, all variables of type t range over the set $\{i, t \setminus i\}$. Now, consider, for example, the equality operator. 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, the specific value i is equal to itself, and not equal to $t \setminus i$. However, two values not equal to i may themselves be equal or unequal. Thus, the result of comparing $t \setminus i$ and $t \setminus i$ for equality is an "unknown" value $\perp$.

Similarly, an array reference `a[x]`, where `a` is an array of index type t, will yield the value of signal `a[i]` if $x = i$ and otherwise $\perp$. Thus, an array reference `a[x]`, where `x` is an arbitrary expression can be replaced by the equivalent expression `(x = i) ? a[i] : bottom`, where `bottom` represents the unknown value $\perp$. As a result, in the abstracted verification problem, only one element of array `a` is referenced.

This interpretation of formulas in an abstract model yields a conservative abstraction. That is, any formula that is true in the abstract model will be true in the original. On the other hand, in some cases the truth value of a formula in the abstraction will be $\perp$, in which case we cannot infer anything about the truth of the formula in the original model. 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 SMV would reduce the type t to the two values i and j, and the abstract value $t \setminus \{i, j\}$. Some examples of this will appear in section 5.

4.1 Formalizing data type reductions

Data type reductions, as used here, are a particular instance of abstract interpretation [CC77]. This, of course, is an old subject, however the particular abstract interpretation used here is believed by the author to be novel. Formalizing the notion of data type reduction for the complete logic used by SMV would require at the very least introducing the entire logic, which is well beyond the scope of this paper. However, we can easily formalize data type reductions in a general framework, for an arbitrary logic, and show in some special cases how this relates to SMV's logic.

To begin with, suppose that we are given a set U of *values*, a set V of *variables*, a set T of *types* and a function $\mathcal{T} : V \to T$, assigning types to variables. Suppose also that we are given a language $\mathcal{L}$ of *formulas*. Formulas are built from a set of *constructors* C, where each constructor $c \in C$ has a *arity* $n_c \geq 0$. An *atomic formula* is $c()$, where $c \in C$ is of arity $n_c = 0$. A *formula*

is defined to be either an atomic formula, or $c(\psi_1, \ldots, \psi_n)$, where $c \in C$ is a constructor of arity $n_c = n \geq 1$, and $\psi_1, \ldots, \psi_n$ are formulas. Now, let a *structure* be a pair $M = (\mathcal{R}, \mathcal{N}, \mathcal{F})$, where $\mathcal{R}$ is a function $T \to \mathcal{P}(U)$, assigning a range of values to every type, $\mathcal{N}$ is a set of *denotations*, and $\mathcal{F}$ is an *interpretation*, assigning to each constructor $c \in C$, a function $\mathcal{F}(c) : \mathcal{N}^n \to \mathcal{N}$. The *denotation* of a formula f in structure M will be written f^M. The denotation of formulas is defined inductively. For formula $\phi = c(\psi_1, \ldots, \psi_n)$, where $c \in C$ is a constructor, $\phi^M = (\mathcal{F}(c))(\psi_1^M, \ldots, \psi_n^M)$.

We will assume that the set of denotations $\mathcal{N}$ admits a pre-order, $\leq$, and that every function $\mathcal{F}(c)$ where $c \in C$, is monotonic with respect to this pre-order. That is, for all $c \in C$, if $x_1 \leq y_1, \ldots, x_n \leq y_n$, where $x_i, y_i \in \mathcal{N}$, then $(\mathcal{F}(c))(x_1, \ldots, x_n) \leq (\mathcal{F}(c))(y_1, \ldots, y_n)$. Given two structures $M = (\mathcal{R}, \mathcal{N}, \mathcal{F})$ and $M' = (\mathcal{R}', \mathcal{N}', \mathcal{F}')$, we will say that a function $h : \mathcal{N} \to \mathcal{N}'$ is a *homomorphism* from M to M' when, for every $c \in C$,

$$h(\mathcal{F}(c)(x_1, \ldots, x_n) \geq (\mathcal{F}(c)(h(x_1), \ldots, h(x_n))$$

If h is a homomorphism from M to M', we will write $M \xrightarrow{h} M'$.

Theorem 2. *If $M \xrightarrow{h} M'$ then for all formulas $\phi \in \mathcal{L}$, $h(\phi^M) \geq \phi^{M'}$*

Proof. By induction over the structure of ϕ, using monotonicity.

Now, let us fix a structure $M = (\mathcal{R}, \mathcal{N}, \mathcal{F})$. Let a *data type reduction* be any function $r : T \to \mathcal{P}(U)$, assigning to each type $t \in T$ a set of values $r(t) \subseteq \mathcal{R}(T)$. That is, a data type reduction maps every type to a subset of its range in M. We wish to define, for every data type reduction r, a reduced structure M_r, and a map h_r, such that $M \xrightarrow{h_r} M_r$.

As an example, let us say that $\mathcal{N}$ is the set of functions $\mathcal{M} \to \mathcal{Q}$, where $\mathcal{M}$ is a set of *models* and $\mathcal{Q}$ is a set of *base denotations*. For now, let $\mathcal{Q}$ be simply the set U of values. A model is a function $\sigma : V \to \mathcal{Q}$, assigning to every variable $v \in V$ a value $\sigma(v)$ in $\mathcal{R}(\mathcal{T}(v))$, the range of its type in M. For any data type reduction $r : T \to \mathcal{P}(U)$, let M_r be a structure $(\mathcal{R}_r, \mathcal{N}_r, \mathcal{F}_r)$, where for every type $t \in T$, $\mathcal{R}_r(t)$ is the set $\{\{x\} \mid x \in r(t)\} \cup \{U \setminus r(t)\}$. That is, the range of type t in the reduced model is the set consisting of the singletons $\{x\}$, for x in $r(t)$ and the set containing all the remaining values. Let $\mathcal{N}_r$, the set of denotations of the reduced structure, be as above, except that set $\mathcal{Q}_r$ of base denotations in the reduced model is $\mathcal{P}(U)$. For any $x, y \in \mathcal{Q}_r$, we will say that $x \leq y$ when $x \supseteq y$, and we will equate $\bot_r$ with the set U of all values. The pre-order $\leq$ on $\mathcal{N}_r$ is the point-wise extension of this order to denotations. That is, for all $x, y \in \mathcal{N}$, $x \leq y$ iff, for all models $\sigma \in \mathcal{M}_r$, $x(\sigma) \leq y(\sigma)$. Finally, the map h_r is defined as follows:

$$h_r(x)(\sigma') = \{x(\sigma) \mid \sigma \in \sigma'\}$$

where for any $\sigma \in \mathcal{M}$ and $\sigma' \in \mathcal{M}_r$, we say $\sigma \in \sigma'$ iff for all variables $v \in V$, $\sigma(v) \in \sigma(v)$. Now, for every constructor c, we must define an abstract interpretation $\mathcal{F}_r(c)$, corresponding to the original interpretation $\mathcal{F}(c)$, such that $M \xrightarrow{h_r} M_r$.

Definition 1. *A constructor $c \in C$ is* safe *for r iff, for all $\sigma \in \mathcal{M}$ and $\sigma' \in \mathcal{M}_r$ for all $x, y \in \mathcal{N}^n$, if $\sigma \in \sigma'$ and $x_i\sigma \in y_i\sigma'$ for $i = 1 \ldots n$, then $(\mathcal{F}(c))(x_1, \ldots, x_n)\sigma \in (\mathcal{F}_r(c))(y_1, \ldots, y_n)\sigma'$.*

Theorem 3. *If every $c \in C$ is safe for r, then $M \xrightarrow{h_r} M_r$.*

Proof. Suppose that for all $\sigma \in \sigma'$, $(\mathcal{F}(c))(x_1, \ldots, x_n)\sigma \in (\mathcal{F}_r(c))(y_1, \ldots, y_n)\sigma'$. Then,

$$\begin{aligned} h_r(\mathcal{F}(c)(x_1, \ldots, x_n))(\sigma') &= \{\mathcal{F}(c)(x_1, \ldots, x_n)(\sigma) \mid \sigma \in \sigma'\} \\ &\subseteq \mathcal{F}_r(c)(y_1, \ldots, y_n)\sigma' \end{aligned}$$

Hence by definition $h_r(\mathcal{F}(c)(x_1, \ldots, x_n)) \geq (\mathcal{F}_r(c))(y_1, \ldots, y_n)$.

We will consider here a few atomic formulas and constructors in the SMV logic of particular interest. For example, every variable $v \in V$ is a atomic formula in the logic. In the original interpretation we have, somewhat tautologically,

$$\mathcal{F}(v)()(\sigma) = \sigma(v)$$

In the abstract interpretation, we have the same definition:

$$\mathcal{F}_r(v)()(\sigma') = \sigma'(v)$$

We can immediately see that v is safe for any data type reduction r. That is, if $\sigma \in \sigma'$, then by definition, for every variable v, $\sigma(v) \in \sigma'(v)$. Thus, $\mathcal{F}(v)()(\sigma) \in \mathcal{F}_r(v)()(\sigma')$.

Now let us consider the equality operator. The concrete interpretation of this operator is:

$$\mathcal{F}(=)(x_1, x_2)(\sigma) = \begin{cases} 1 & ; x_1\sigma = x_2\sigma \\ 0 & ; \text{else} \end{cases}$$

The abstract interpretation is:

$$\mathcal{F}_r(=)(y_1, y_2)(\sigma') = \begin{cases} \{1\} & ; y_1\sigma' = y_2\sigma',\ |y_1\sigma'| = 1 \\ \{0\} & ; y_1\sigma' \cap y_2\sigma' = \emptyset \\ \bot_r & ; \text{else} \end{cases}$$

That is, if the arguments are equal singletons, they are considered to be equal, if they are disjoint sets, they are considered to be unequal, and otherwise the result is $\bot_r$. This constructor is clearly safe for any r. That is, a simple case analysis will show that if $x_1\sigma \in y_1\sigma'$ and $x_2\sigma \in y_2\sigma'$, then $\mathcal{F}(=)(x_1, x_2)(\sigma) \in \mathcal{F}_r(=)(y_1, y_2)(\sigma')$.

Finally, we consider array references. An *array* a is an n-ary constructor. To each argument position $0 \leq i < n$, we association an *index type* $\mathcal{T}_i(a) \in T$. We also associate with a a collection of variables $\{a[x_1]\cdots[x_n] \mid x_i \in \mathcal{R}(\mathcal{T}_i(a))\} \subseteq V$. The interpretation of the array constructor a is:

$$\mathcal{F}(a)(x_1, \ldots, x_n)(\sigma) = \begin{cases} \sigma(a[x_1\sigma]\cdots[x_n\sigma]) & ; x_i\sigma \in \mathcal{R}(\mathcal{T}_i(a)) \\ \bot & ; \text{else} \end{cases}$$

That is, if all of the indices are in the range of their respective index types, then the result is the value of the indexed array element in σ, else it is $\bot$ (the "unknown" value in the concrete model). The abstract interpretation is

$$\mathcal{F}_r(a)(y_1, \ldots, y_n)(\sigma') = \begin{cases} \sigma'(a[m_1]\cdots[m_n]) & ; y_i\sigma' = \{m_i\},\ m_i \in r(\mathcal{T}_i(a)) \\ \bot_r & ; \text{else} \end{cases}$$

That is, if all of the indices are *singletons* in the *reduced* range of their respective index types, then the result is the value of the indexed array element in σ', else it is $\bot_r$. Thus, if a given array index has index type t, then the abstract interpretation depends only on array elements whose indices are in $r(t)$. This is what allows us to reduce unbounded arrays to a finite number of elements for model checking purposes. It is a simple exercise in case analysis to show that array references are safe for r.

Most of the remaining constructors in the SMV logic are trivially safe in that they return $\bot_r$ when any argument is not a singleton. Since all the constructors are safe, we have $M \overset{h_r}{\to} M_r$. Now, suppose that we associate with each structure $M = (\mathcal{R}, \mathcal{N}, \mathcal{F})$ a distinguished denotation $\textsc{True} \in \mathcal{N}$, the denotation of valid formulas. We will say that a homomorphism h from M to M' is *truth preserving* when $h(x) \geq \textsc{True}'$ implies $x = \textsc{True}$. This gives us the following trivial corollary of theorem 2:

Corollary 1. *If* $M \xrightarrow{h} M'$ *and* h *is truth preserving, then* $\phi^{M'} = \textsc{True}'$ *implies* $\phi^{M} = \textsc{True}$.

In the case of a data type reduction r, we will say that TRUE in structure M is the denotation that maps every model to 1, while $\textsc{True}_r$ in the reduced model M_r maps every model to the singleton $\{1\}$. In this case, h_r is easily shown to be truth preserving, hence every formula that is valid in M_r is valid in M. Note that, if r maps every type to a finite range, then the range of all variables is finite in M_r. Further, because of the abstract interpretation of array references, the number of variables that ϕ^{M_r} depends on is finite for any formula ϕ. Thus, we can apply model checking techniques to evaluate ϕ^{M_r} even if the ranges of variables in M are unknown.

The above treatment is simplified somewhat relative the the actual SMV logic. For example, the SMV logic is a linear temporal logic. To handle temporal operators, we must extend the type of base denotations $\mathcal{Q}$ from values in U to infinite sequences in U^{ω}. Further extensions of $\mathcal{Q}$ are required to handle nondeterministic choice and quantifiers. However, the basic theory presented above is not substantially changed by these extensions.

5 Verifying a version of Tomasulo's algorithm

Putting the above elements together – compositional reasoning, symmetry reduction, temporal case splitting and data type reduction – we have the tools to reduce the verification of a complex hardware system with unbounded resources to a collection finite state verification problems with a small number of state bits in each. When the number of state bits in each proof subgoal is sufficiently small, verification can proceed in a completely automated way without further manual decomposition. As an example of this, we now apply the above techniques to the verification of an implementation of Tomasulo's algorithm for out-of-order instruction execution.

5.1 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" to an execution unit. The flow of instructions in our implementation of Tomasulo's algorithm is pictured in figure 2. Each instruction, as it arrives, fetches its two operands

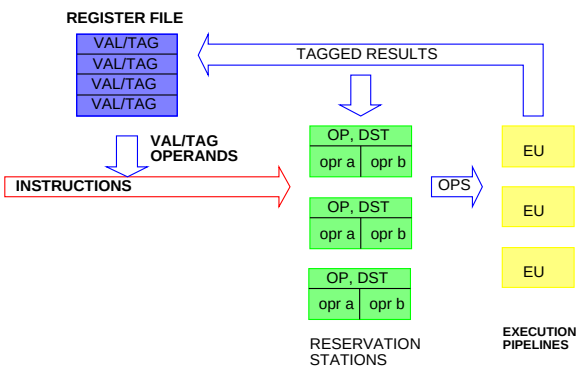

Fig. 2. Flow of instructions in Tomasulo's algorithm

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 (RS). The RS watches the results returning from the execution unit, and when a result's tag matches one of its operands, it records the value in place of the tag. When the RS 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 RS 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 RS to store the instruction, or because the value of the register to be read to an output is not yet available.

5.2 Structural decomposition

The machine is specified with respect to an abstract model. This is a simple implementation of the instruction set that executes instructions one at a time in sequence (this is commonly referred to as an "instruction set architecture" model). The input and output signals of the abstract model and the implementation are the same, so there is no need to write refinement maps relating them.

The first step in the proof is to use refinement maps to break the verification problem into tractable parts. In [McM98], the circular composition rule was used decompose the large data structures (*i.e.*, the register file, and reservation station array). Here, a substantially simpler proof is obtained by using temporal case splitting and data type reductions. Essentially, we break the verification problem into two lemmas. The first lemma states that operands fetched for instructions are correct, while the second states that the results produced from these operands are correct. Each lemma is specified by a refinement map. The first map specifies the operand values stored in the reservation stations, while the second specifies the values returning on the result bus from the execution units. We apply the circular compositional rule, using operand correctness to prove result correctness, and result correctness to prove operand correctness.

In order to be able to specify refinement maps 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 (this is exactly as in [McM98]). In this case, our auxiliary state variables record the correct values of the operands and the 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. The SMV code for this is the following:

```
if(~stallout & opin = ALU){
  next(aux[st_choice].opra) := opra;
  next(aux[st_choice].oprb) := oprb;
  next(aux[st_choice].res)  := res;
}
```

This states that, if the machine does not stall, and if the incoming instruction is an ALU operation, then we store in array `aux` 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.

The refinement maps themselves are normally also written as assignments, though they are treated as temporal properties to be proved. 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.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`.

5.3 Using temporal case splitting

The refinement maps divide the implementation into two parts for the purpose of verification (the operand forwarding logic and the instruction execution logic). However, 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 map to another.

Consider, for example, a value returning on the result bus. It is the result produced by a given reservation station i (which we will call the producer). It then (possibly) gets stored in a register j. Finally it is fetched as an operand for reservation station k (which we will call the consumer). This suggests a case split which we will reduce the size of the verification problem to just two reservation stations and one register. For each operand arriving at consumer reservation station k, we split the specification into 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 need to use only reservation stations i and k, and register j. The other elements of these arrays are abstracted away, by replacing them with the "unknown" value $\perp$. In fact, this is done automatically by data type reduction, as we will see later. The basic idea of the case split is depicted in figure 3. We might refer to this as "path splitting", since we are splitting cases based on the path taken by a given data item through the system.

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;
```

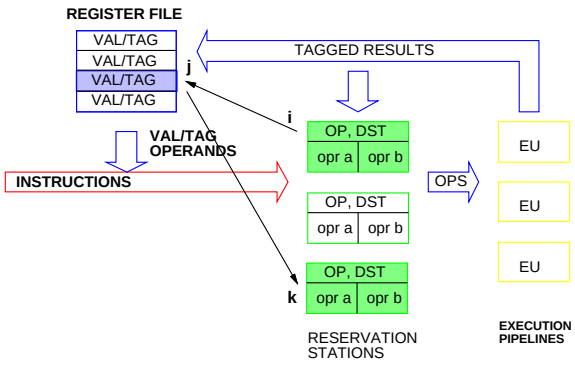

Fig. 3. Path splitting in Tomasulo's algorithm.

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 added an auxiliary variable `auk[k].srca` 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.

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. However, if all the registers are symmetric to one 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 simply declare the types of register indices and reservation station indices to be scalarsets. The SMV proof assistant 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.

5.4 Infinite state verification

Up to this point we have defined refinement maps, 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 of a few relevant values (to 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, reservation stations, and execution units. To do this, we simply declare the index types to be scalarsets with undefined range, as follows:

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

This declares both `REG` (the type of register indices) and `TAG` (the type of reservation station indices) to be symmetric, but does not declare ranges for these types. This is possible because the

verification process, using symmetry and data type reductions, is independent of the range of these types. For example, when verifying the operand correctness lemma, for a given case (i, j, k), SMV will by default reduce the type `TAG` to just three values: i, k and an abstract value to represent all the remaining values. Similarly, the type `REG` will be reduced to just two values: j and an abstract value. Note that this has the side effect of eliminating all the reservation stations other than i and k, and all the registers other that j, by substituting the value $\perp$.

Further, using symmetry reduction, we only need to verify a fixed set of cases for i, j and k, regardless of the actual range of the types. Thus, we can verify the system generically, for any range (finite or infinite) of these types.

5.5 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) for binary values of this width, they cannot typically handle many other operations, 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. This way, we can separate out the difficult problem of arithmetic verification. 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 can prove that our implementation is correct for all ALU functions. 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.

Interestingly, the techniques described above are sufficient to handle uninterpreted functions, without introducing any new logical constructs or decision procedures. To introduce an uninterpreted function in SMV, we simply observe that an *array* in SMV is precisely an uninterpreted function (or, if you prefer, the "lookup table" for an arbitrary function). Thus, to introduce an uninterpreted function symbol in SMV, we simply declare an array of the appropriate type. For example:

```
forall (a,b in WORD)
  f[a][b] : WORD;
```

declares a binary function `f` that takes two words `a` and `b` and yields a word `f[a][b]`. Note, however, that one difference between arrays in SMV and function symbols in a more typical logic is that SMV arrays can change over time (*i.e.*, they represent *flexible* function symbols). Since we want our arithmetic function to be constant in time, we can simply declare:

```
next(f) := f;
```

We model ALU operations in both the abstract model and implementation with lookups in the array `f`. With this abstraction, we can deal with any size data word, by declaring the type of data words to be a scalarset. That is:

```
scalarset WORD undefined;
```

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 make the problem finite state. 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;
```

SMV automatically applies symmetry reduction to reduce an infinite number of cases to just 6 representative ones. In addition, by default, it will use data type reduction to 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.

5.6 Applying model checking

By applying the above described proof decomposition, the SMV system produces a set of proof subgoals that can be discharged by model checking. If any of these are false, SMV will produce a sequential counterexample. In this way, we can verify our implementation of Tomasulo's algorithm 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. Note that we have not yet applied data type reduction to execution unit indices. Thus, our proof is still applicable to only a fixed number of execution units. For the case of one execution unit, there are 11 model checking subgoals, with a maximum of 25 state variables. The overall verification time (including the generation of proof goals and model checking) is just under 4 CPU seconds (on SPARC Ultra II server). If we increase the number of execution units to 8, the verification time increases to roughly one minute. While this is not unreasonable, we will discuss shortly how to generalize the proof to an arbitrary number of execution units.

Perhaps a more important measure for a technique such as this is the amount of user effort required. The time required for an experienced user of SMV (its author!) to write, debug and verify the proof was approximately an hour and ten minutes. Note that the design itself was already debugged and was previously formally verified using an earlier methodology [McM98]. The time required to write an debug the design was far greater than that required to execute the proof.[1]

5.7 Proving noninterference

It is also possible to use case splitting and data type reduction to verify the design for an arbitrary number of execution units. To do this, as one might expect, we split the result lemma into cases as a function of the execution unit used to produce a given result, and eliminate all but the execution unit in question. This, however, requires introducing a "noninterference lemma". This lemma states that no other execution unit spuriously produces the result in question. Such interference would confuse the control logic in the reservation station and lead to incorrect behavior. The noninterference lemma is stated in temporal logic as follows:

[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`.

```
lemma3 : assert G (pout.valid -> (complete_eu = aux[pout.tag].eu));
```

This says, in effect, that if the result bus `pout` is returning a value from an execution unit `complete_eu`, and if that value is tagged for reservation station i, then `complete_eu` must be the execution unit from which reservation station i is expecting a result. Note, an auxiliary variable `aux[`i`].eu` is used to store the index of the execution unit to which the instruction in reservation station i was issued. We can prove this lemma by splitting cases on `pout.tag` and `complete_eu`, as follows:

```
forall(i in TAG) forall(j in EU)
  subcase lemma3[i][j] of lemma3 for pout.tag = i & complete_eu = j;
```

In this case, the data types `TAG` and `EU` (the type of execution unit indices) are reduced to just the values i and j respectively, plus an abstract value. Thus, we can prove it for an arbitrary number of execution units using model checking. However, here an interesting phenomenon occurs: the the lemma for a given execution unit j holds at time t only if it is true for all the other units up to time $t-1$. That is, in effect, we must prove that unit j is not the *first* unit to violate the lemma. This is done using the circular compositional rule, with the following declaration:

```
forall(i in TAG) forall(j in EU)
  using (lemma3) prove lemma3[i][j];
```

The parentheses around `lemma3` tell SMV to assume the general lemma up to time $t-1$ when proving case i, j at time t. This is typical of such noninterference lemmas, where the first unit or process to violate the lemma may cause others to behave incorrectly in the future. We can now use `lemma3` to prove the result correctness lemma for any reservation station i and execution unit j. This allows us to use a data type reduction for type `EU`, and hence to verify the lemmas for an undefined range of execution unit indices (that is, for any number of units). The resulting model checking subgoals require only a few seconds to discharge.

5.8 Adding a reorder buffer

Now, suppose that we modify the design to use a "reorder 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 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,VB98,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 reorder 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.

6 Conclusions and future work

We have seen that, within a compositional framework, a combination of case splitting, symmetry, and data type reductions can be used to reduce verification problems involving structures of unbounded or infinite size to a tractable number of finite state subgoals, with few enough state variables to be verified by model checking. This is made possible by a new technique of data type reduction and a new way of treating uninterpreted functions in a model checking context. These techniques form part of an overall strategy for hardware verification, that can be applied to such diverse hardware applications as out-of-order instruction set processors, cache coherence systems [Eir98] and packet buffers for communication systems [Tru98]. Note that the techniques of model checking, symmetry reduction, temporal case splitting, and data type reduction are tightly interwoven in this methodology. All are used, for example to support verification using uninterpreted functions. Their integration into a mechanical proof assistant means that the proof does not rely in any way on reasoning "on paper".

In this work, one possible form of data type reduction is described. There are, however, many other possibilities. For example, the technique has 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 by induction that a FIFO buffer delivers an infinite sequence of packets in the correct order.

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 map approach, combined with the various available reductions and model checking, makes it possible to avoid writing 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. Such proofs, since they specify the behavior of fewer signals than do proofs involving inductive invariants, can be less sensitive to design changes, as we saw in the case of adding a reorder buffer. 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. By combining the strengths of these two methods, we may thus arrive at a scalable methodology for formal hardware verification.

References

[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.

[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.

[BS90] Randal E. Bryant and Carl-Johan Seger. Formal verification of digital circuits using symbolic ternary system models. In Robert Kurshan and Edmund M. Clarke, editors, *Workshop on Computer-Aided Verification*, New Brunswick, New Jersey, June 1990. Center for Discrete Mathematics and Theoretical Computer Science (DIMACS).

[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.

[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.

[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.

[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.

[McM98] K. L. McMillan. Verification of an implementation of tomasulo's algorithm by compositional model checking. In A. J. Hu and M. Y. Vardi, editors, *Conference on Computer-aided Verification (CAV '98)*, number 1427 in LNCS, pages 100–21. Springer-Verlag, 1998.

[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.

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

[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.

[VB98] 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.

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