Verification of Infinite State Systems by Compositional Model Checking

K. L. McMillan

Cadence Berkeley Labs

Abstract. A method of compositional verification is presented that uses the combination of *temporal case splitting* and *data type reductions* to reduce types of infinite or unbounded range to small finite types, and arrays of infinite or unbounded size to small fixed-size arrays. This supports the verification by model checking of systems with unbounded resources and *uninterpreted functions*. The method is illustrated by application to an implementation of Tomasulo's algorithm, for arbitrary or infinite word size, register file size, number of reservation stations and number of execution units.

1 Introduction

Compositional model checking reduces the verification of a large system to a number of smaller verification problems that can be handled by model checking. This is necessary because model checkers are limited with respect to state space size. Compositional methods are implemented in, for example, the SMV system [13] and the Mocha system [1]. The typical proof strategy using these systems is to specify *refinement relations* between an abstract model and certain variables or signals in an implementation. This allows components of the implementation to be verified in context of the abstract model. This basic approach is limited in two respects. First, it does not reduce data types with large ranges, such as addresses or data words. For example, it is ineffective for systems with 32-bit or 64-bit memory address spaces. Second, the approach can verify only a fixed 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).

Here, we present a method based on *temporal case splitting* and a form of *data type reduction*, that makes it possible to handle types of arbitrary or infinite range, and arrays of arbitrary or infinite size. Temporal case splitting breaks the correctness specification for a given data item into cases, based on the path the data item has taken through the system. For each case, we need only consider a small, fixed subset of the elements of the large data structures. The number of cases, while potentially very large, can be reduced to a small number by existing techniques based on symmetry [13]. Finally, for any given case, a data type reduction can 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 representing the remaining values in the original type. Thus, we reduce the large or infinite types and structures to small finite types and structures for the purpose of model checking.

Together, these methods also allow specification and verification using uninterpreted functions. For example, we can use model checking to verify the correctness of an instruction set processor independent of its arithmetic functions. This separates the verification of data and control flow from the verification the the arithmetic units.

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

Related work Data type reduction, as applied here, can be viewed as a special case of *abstract interpretation* [5]. It is also related to reductions based on *data independence* [18] in that a large data type is reduced to a small finite one, using a few representative values and an extra value to represent everything else. The technique used here is more general, however, in that it does not require control to be independent of data. For example, it allows control to depend on comparisons of data values. The technique may therefore be applied to reduce addresses, tags and pointers, which are commonly compared to determine control behavior (an example of this appears 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 handle systems with unbounded memories, FIFO buffers, *etc.*.

Lazic and Roscoe [11] also describe a technique for reducing unbounded arrays to finite ones 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. More importantly, the method of [11] does not allow equality comparison of values stored in arrays, nor the storage of array indices in arrays. Thus, for example, it cannot handle 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. Note that comparing values stored in an unbounded array, or even including one bit of status information in the 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 used by Long [12] in his work on generating abstractions using BDD's. However, that work applied only to concrete finite types. Here, types of arbitrary or infinite size are reduced to finite types. Also, Long's work did not treat the reduction of arrays. What makes it possible to do this here 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, both algorithmically and methodologically. The reason for using uninterpreted functions is the same – to abstract away from the actual functions computed on data. However, existing techniques using uninterpreted functions, such as [4, 10, 7, 14, 17, 2] are based essentially on symbolic simulation. The present method allows the combination of uninterpreted functions with model checking. This distinction has significant practical consequences for the user. That is, the existing methods are all based on proving commutative diagrams. 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, the user must in general provide an *inductive invariant*. The commutative diagram is proved only for those states satisfying the invariant. By contrast, in the present technique, there is no need to provide an inductive invariant, since the model checker determines the strongest invariant

by reachability analysis. This not only saves the user a considerable effort, but also improves the re-usability of proofs, as we will observe later.

In general, when uninterpreted functions with equality are added to temporal logic, the resulting logic is undecidable. The present method is not a decision procedure for such a logic, but rather a user-guided reduction to the propositional case that is necessarily incomplete. Note, an earlier semi-decision procedure for such a logic [8], is sound only in a very restricted case; for most problems of practical interest, the procedure is not sound, and can only be used to find counterexamples. Of the various non-temporal techniques using uninterpreted functions, the present method is most similar to [17], since it is also based on *finite instantiation*. However, the methods are not similar algorithmically.

Thus, the methods presented here are novel in three aspects: first the particular techniques of data type reduction and of handling uninterpreted functions are novel. 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.

Outline of the article Section 2 is a brief overview of earlier work on compositional methods and symmetry, on which the current work is based. Section 3 then describes temporal case splitting and its implementation in the SMV system. Section 4 covers data type reduction and its implementation in SMV. Finally, in section 5, these techniques are illustrated by applying them to an implementation of Tomasulo's algorithm. This is the same example used in [13], however in this case the new techniques substantially simply the proof, decrease the run-time of the prover, and allow verification for unbounded or infinite resources.

2 Compositional verification and symmetry

The SMV system uses compositional model checking to support refinement verification – proving that an abstract model, acting as the system specification, is implemented by some more detailed system model. Correctness is usually defined by *refinement relations* that specify signaling behavior at suitable points in the implementation in terms of events occurring in the abstract model (see fig. 1). Typically, the abstract model, the implementation and the refinement relations are all expressed in the same HDL-like language, as sets of equations that may involve time delay. Formally, however, we can view them as simply linear temporal logic properties.

The refinement relations decompose the system structurally into smaller parts for separate verification. This relies on a method of *circular compositional proof* whereby we may assume that one temporal property P holds true while verifying property Q, and *vice versa*. In the figure, for example, 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 [9]. To exploit the symmetry of a given type, we must guarantee that values of that type are only used in certain symmetric ways. For example, they may be compared for equality, or used as indices of arrays. SMV enforces these conditions by static type checking. For further details of these methods, the reader is referred to [13].

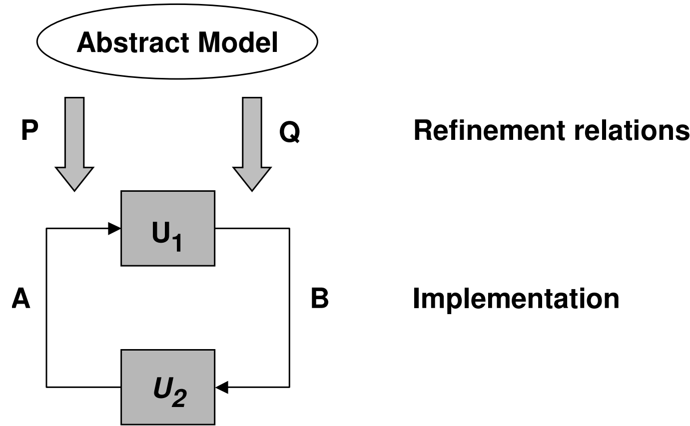

Fig. 1. Compositional refinement verification

3 Temporal case splitting

Hardware designs typically contain large arrays, such as memories, FIFO buffers, content-addressable memories (CAM's) and so forth. Their state space is often intractably large, and thus we usually cannot apply model checking to them directly. However, we can often confine the verification problem to one or two elements of the array 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 an array. Thus, we consider individually each path that a data item might take through a given system, and reduce the number of state variables accordingly.

Temporal case splitting breaks the proof of temporal property $G\phi$ (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 $v = i$. Then, since at all times v must have some value, we can infer that ϕ must be true at all times. This inference is based on the following fairly trivial fact about temporal logic:

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

Typically, v is an auxiliary variable recording the location in some array that was used to store the data item currently appearing at a unit output (see [13] for the use of auxiliary variables in compositional proofs). To prove a give case $v = i$, it is commonly only necessary to refer to element i of the array. The other elements of the array can be abstracted from the model by replacing them with an "unknown" value $\bot$, much as in ternary symbolic simulation [3]. Thus, in effect, we decompose a large array into its elements 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`.

4 Data type reductions

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

For example, let t be an arbitrary type. Regardless of the actual range of t, we can reduce the range to a set containing a 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` has index type t, yields the value of signal `a[i]` if $x = i$ and otherwise $\perp$. As a result, in the reduced verification problem, only one element, `a[i]`, of array `a` is referenced.

Using such an abstract interpretation, any formula that is true in the reduced 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 this 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, given the particular case being verified. For example, if the case 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.

Formalizing data type reductions Data type reductions, as used here, are a particular instance of abstract interpretation [5]. 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* $n_c \geq 0$. Examples of constructors in SMV's logic are function symbols, constants, variables and quantifiers. 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 triple $M = (\mathcal{R}, \mathcal{N}, \mathcal{F})$, 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. This is defined inductively. That is, 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 $(\mathcal{R}_r, \mathcal{N}_r, \mathcal{F}_r)$. For every type $t \in T$, let $\mathcal{R}_r(t)$ be 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 $\mathcal{Q}_r$, the set 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 $\perp_r$ 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 $x \in \mathcal{N}^n$, $y \in \mathcal{N}_r^n$ 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 $M \stackrel{h_r}{\to} M_r$. 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* $\phi^{M'} = \text{TRUE}'$ *implies* $\phi^M = \text{TRUE}$*.*

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 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 types in M are unknown, which also allows the possibility that they are infinite.

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

Combining the above methods – compositional reasoning, symmetry reduction, temporal case splitting and data type reduction – we can reduce the verification of a complex hardware system with unbounded resources to a collection of finite state verification problems, each with a small number of state bits. When the number of state bits in each subproblem is sufficiently small, verification can proceed automatically by model checking. As an example, we apply the above techniques to the verification of an implementation of Tomasulo's algorithm for out-of-order instruction execution.

Tomasulo's algorithm Tomasulo's algorithm [15] 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. Each pending instruction is held in a "reservation station" until the values of its operands become available. It is then issued to an execution unit. The flow of instructions in our implementation is pictured in figure 2. Each instruction, as it arrives, fetches its two operands from a special register file. Each register in this file holds either

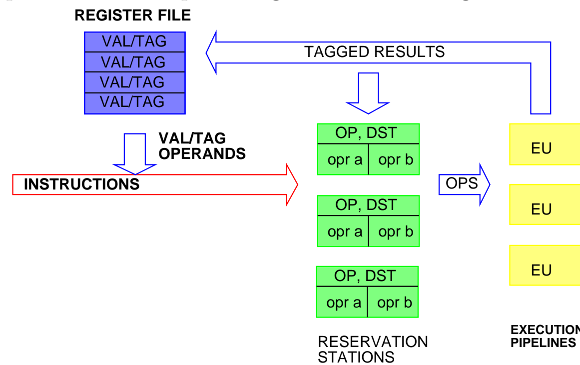

Fig. 2. Flow of instructions in Tomasulo's algorithm

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 tagged results returning from the execution unit. When a 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 unit. When the result returns from the execution unit, the RS is cleared, and the result value, if needed, is stored in the destination register.

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.

Structural decomposition The implementation is modeled in the SMV language at the RTL level. This machine is in turn specified with respect to an abstract model. This is a simple implementation of the instruction set that executes instructions sequentially. The input and output signals of the abstract model and the implementation are the same, so there is no need to write refinement relations between them.[1]

We begin the proof by using refinement relations to break the verification problem into tractable parts. In [13], the circular compositional rule was used decompose the arrays (*e.g.*, the register file, and reservation station array). Here, a substantially simpler proof is obtained using temporal case splitting and data type reductions. Essentially, we break the verification problem into two lemmas. The first lemma specifies the operand values stored in the reservation stations, while the second specifies the values returning on the result bus from the execution units, both in terms of the abstract model. We apply circular compositional reasoning, using operand correctness to prove result correctness, and result correctness to prove operand correctness.

To specify 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 (exactly as in [13]). 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 to be 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;
}
```

That is, if the machine does not stall, and we have 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. The variable `st_choice` indicates the reservation station to be used. Storing these values will allow us to verify that the actual operands and results we eventually obtain are correct.

The refinement relations themselves are also written as assignments, though they are treated as temporal properties to be proved. For example, here is the operand correctness specification (for operand `opra`):

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

[1] Here, we prove only safety. For the liveness proof, see "Circular compositional reasoning about liveness", in this volume.

For present purposes, the declaration "`layer lemma1:`" simply attaches the name `lemma1` to the property. `TAG` is the type of RS indices. Thus, the property must hold for all reservation stations k. If station `st[`k`]` is is `valid` (contains an instruction) and its `opra` operand is a value (not a tag), then the value must be equal to the correct operand value stored in the auxiliary array `aux`. Note that semantically, the assignment operator here simply stands for equality. Pragmatically, however, it also tells the system that this is a specification of signal `st[`k`].opra.val`, and that this specification depends on signal `aux[`k`].opra`. The SMV proof system uses this information when constructing a circular compositional proof.

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;
```

That is, 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.

Using temporal case splitting The refinement relations divide the implementation into two parts for the purpose of verification (operand forwarding logic and instruction execution logic). However, there remain large arrays in the model that prevent us from applying model checking at this point. These are the register file, the reservation station array and the execution unit array. Therefore, we break the verification problem into cases, as a function of the path a data item takes when moving from one refinement relation to another.

Consider, for example, a value returning on the result bus. It is the result produced by a reservation station i (the producer). It then (possibly) gets stored in a register j. Finally it is fetched as an operand for reservation station k (the consumer). This suggests a case split which reduces the verification problem to just two reservation stations and one register. For each operand arriving at consumer RS k, we split the specification into cases based on the producer i (this is indicated by the "tag" of the operand) and the register j (this is the source operand index of the instruction). To prove just one case, we need to use only reservation stations i and k, and register j. The other elements of these arrays are automatically abstracted away, replacing them with the "unknown" value $\perp$. The effect of this reduction is depicted in figure 3.

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

```
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 `lemma1` into an array of cases (i, j), where i is the producer reservation station and j is the source register. Note, we added an auxiliary variable `auk[k].srca` to record the 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 arrays down into their elements 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.

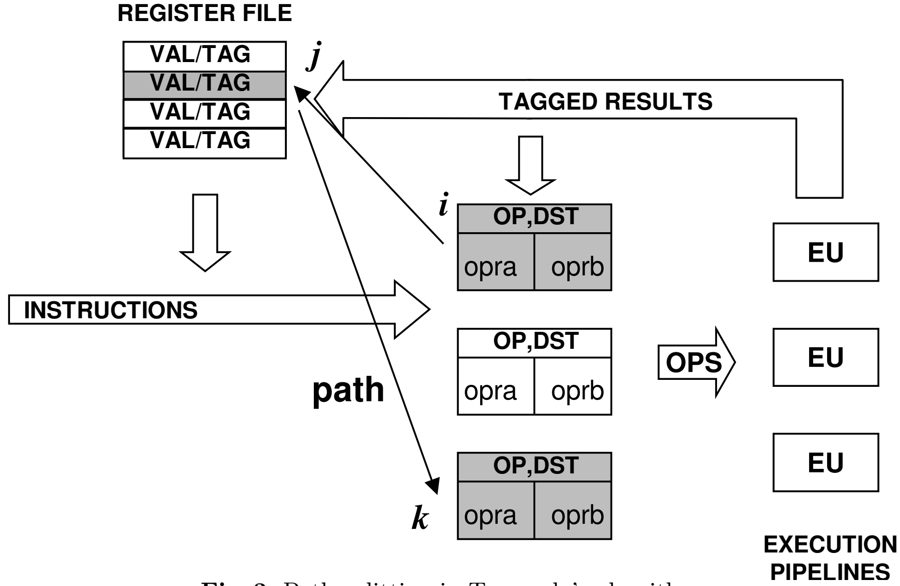

Fig. 3. Path splitting in Tomasulo's algorithm.

To verify operand correctness, 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. SMV verifies the symmetry and automatically chooses a set of representative cases. In fact, it chooses the cases $(i = 0, j = 0, k = 0)$ and $(i = 0, j = 0, k = 1)$. All other cases reduce to one of these by permuting the scalarset types. Thus, we have reduced $O(n^3)$ cases to just two.

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 remain, however, the large types, *i.e.* the data values and possibly the index types. To handle these, we use data type reduction to reduce these types to small sets consisting of a few relevant values and an abstract value representing 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 operand correctness, for a given case (i, j, k), SMV by default reduces the type `TAG` to just three values: i, k and an abstract value. Similarly, the type `REG` is reduced to just two values: j and

an abstract value. 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, due to symmetry, 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.

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 handle some other operations, such as multiplication. Moreover, it would be better to verify our implementation generically, regardless of the arithmetic operations in the instruction set. This way, we can isolate the problem of binary arithmetic verification. This is done by introducing an *uninterpreted function symbol* f for the ALU. 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. The uninterpreted function appraoch also has the advantage that the symmetry of data values is not broken. Thus, we can apply symmetry reductions to data values. As a result, we use only a few representative 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;
```

This declares a binary function `f` that takes two words `a` and `b` and returns a word `f[a][b]`. Since we want our arithmetic function to be invariant over time, we declare:

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

We replace ALU operations in both the abstract model and implementation with lookups in the the array `f`. We can exploit the symmetry of data words by declaring the type of data words to be a scalarset. That is:

```
scalarset WORD undefined;
```

In fact, since the actual range of the type is undeclared, in principle we are verifying the implementation for any size data word. We then use case splitting on data values to make the problem finite state. That is, we verify result correctness for the case when the operands have 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 needed to have a representative set is just $3! = 6$. Here is SMV declaration used 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 6 representatives. By default, it uses data type reduction to reduce the (possibly infinite) type of data words to the specific values a, b and c, and an abstract value. Thus, in the worst case, when a, b and c are all different, only two bits are needed to encode data words. We have thus reduced an infinite state verification problem to a finite number of finite state problems.

Applying model checking Applying the above proof decomposition, the SMV system produces a set of finite state proof subgoals for the model checker. When all are checked, we have verified 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 still applies only to a fixed number of execution units. For one execution unit, there are 11 model checking subgoals, with a maximum of 25 state variables. The overall processing time (including generation of proof goals and model checking) is just under 4 CPU seconds on a SPARC Ultra II server. Increasing the number of execution units to 8, the processing time increases to roughly one minute. We will discuss shortly how to generalize the proof to an arbitrary number of execution units.

Perhaps a more important metric for a technique such as this is the 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 [13]. The time required to write and debug the design was far greater than that required to effect the proof.[2]

Proving noninterference We can also verify the design for an arbitrary number of execution units. To do this, as one might expect, we split the result lemma into cases based on the execution unit used to produce the result, eliminating all the other units. This, however, requires introducing a "noninterference lemma". This states that no other execution unit spuriously produces the result in question. Such interference would confuse the control logic in the RS and lead to incorrect behavior. The noninterference lemma is stated in temporal logic as follows:

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

In effect, 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 this execution unit. We can prove this lemma by splitting cases on `pout.tag` and `complete_eu`:

```
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 the lemma for an arbitrary number of execution units. However, here an interesting phenomenon occurs: 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 $t-1$. In effect, we must prove that unit j is not the *first* unit to violate the lemma. This is done using the circular compositional method, with the following declaration:

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

[2] Details of this example 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`.

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 noninterference lemmas, where the first unit to violate the lemma may cause others to fail in the future. We can now use `lemma3` to prove result correctness for any reservation station i and execution unit j, for any number of execution units. The resulting model checking subgoals require only a few seconds to discharge.

Adding a reorder buffer Now, suppose that we modify the design to use a "reorder buffer". That is, instead of writing results to the register file when they are produced, we store them in a buffer and write them to the register file in program order. This might be 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 [4, 10, 7, 14, 17, 2], which are based on symbolic simulation. Because we are using model checking, it is not necessary to write inductive invariants. 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 an inductive invariant, this 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

Within a compositional framework, a combination of case splitting, symmetry, and data type reductions can reduce verification problems involving arrays 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 enabled by a new method of data type reduction and a method of treating uninterpreted functions in model checking. These techniques are 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 [6] and packet buffers for communication systems [16]. Note that the model checking, symmetry reduction, temporal case splitting, and data type reduction are tightly interwoven in this methodology. All are used, for example to support uninterpreted functions. Their integration into a mechanical proof assistant means that the proof does not rely in any way on reasoning "on paper".

One possible form of data type reduction is described here. There are, however, many other possibilities. For example, an inductive data type has been added, which allows incrementation (*i.e.*, a successor function). This can be used, for example, to

show by induction that a FIFO buffer delivers an infinite sequence of packets in the correct order.

The methodology used here is an attempt to combine in a practical way the strengths of model checking and theorem proving. The refinement relation approach, combined with various reductions and model checking, makes it possible to avoid writing assertions about all 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 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 basic 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

1. R. Alur, T. A. Henzinger, F. Mang, S. Qadeer, S. K. Rajamani, and S. Tasiran. Mocha: Modularity in model checking. In A. J. Hu and M. Y. Vardi, editors, *CAV '98*, number 1427 in LNCS, pages 521–25. Springer-Verlag.
2. S. Berezin, A. Biere, E. Clarke, and Y. Zhu. Combining symbolic model checking with uninterpreted functions for out-of-order processor verification. In *FMCAD '98*, number 1522 in LNCS, pages 351–68. Springer, 1998.
3. R. E. Bryant and C.-J. Seger. Formal verification of digital circuits using symbolic ternary system models. In R. Kurshan and E. M. Clarke, editors, *Workshop on Computer-Aided Verification*, New Brunswick, New Jersey, June 1990.
4. J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *Computer-Aided Verification (CAV '94)*. Springer-Verlag, 1994.
5. 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.
6. A. Eiriksson. Formal design of 1M-gate ASICs. In *FMCAD '98*, number 1522 in LNCS, pages 49–63. Springer, 1998.
7. R. Hojati and R. K. Brayton. Automatic datapath abstraction of hardware systems. In *CAV '95*, number 939 in LNCS, pages 98–113. Springer-Verlag, 1995.
8. R. Hojati, A. Isles, D. Kirkpatrick, and R. K. Brayton. Verification using uninterpreted functions and finite instantiations. In *FMCAD '96*, volume 1166 of *LNCS*, pages 218–32. Springer, 1996.
9. C. Ip and D. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1-2):41–75, Aug. 1996.
10. R. B. Jones, D. L. Dill, and J. R. Burch. Efficient validity checking for processor verification. In *ICCAD '95*, 1995.
11. 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.
12. D. E. Long. Model checking, abstraction, and compositional verification. Tecnical report CMU-CS-93-178, CMU School of Comp. Sci., July 1993. Ph.D. Thesis.
13. K. L. McMillan. Verification of an implementation of tomasulo's algorithm by compositional model checking. In *CAV '98*, number 1427 in LNCS, pages 100–21. Springer-Verlag, 1998.
14. J. U. Skakkabaek, R. B. Jones, and D. L. Dill. Formal verification of out-of-order execution using incremental flushing. In *CAV '98*, number 1427 in LNCS, pages 98–109. Springer-Verlag, 1998.

15. R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM J. of Research and Development*, 11(1):25–33, Jan. 1967.
16. 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.
17. M. Velev and R. E. Bryant. Bit-level abstraction in the verification of pipelined microprocessors by correspondence checking. In *FMCAD '98*, number 1522 in LNCS, pages 18–35. Springer, 1998.
18. P. Wolper. Epressing interesting properties of programs in propositional temporal logic. In *13th ACM POPL*, pages 184–193, 1986.