Verification of an Implementation of Tomasulo's Algorithm by Compositional Model Checking

K. L. McMillan

Cadence Berkeley Labs
2001 Addison St., 3rd floor
Berkeley, CA 94704-1103
mcmillan@cadence.com

Abstract. An implementation of an out-of-order processing unit based on Tomasulo's algorithm is formally verified using compositional model checking techniques. This demonstrates that finite-state methods can be applied to such algorithms, without recourse to higher-order proof systems. The paper introduces a novel compositional system that supports cyclic environment reasoning and multiple environment abstractions per signal. A proof of Tomasulo's algorithm is outlined, based on refinement maps, and relying on the novel features of the compositional system. This proof is fully verified by the SMV verifier, using symmetry to reduce the number of assertions that must be verified.

1 Introduction

We present the formal design verification of an "out-of-order" processing unit based on Tomasulo's algorithm [Tom67]. This and related techniques such as "register renaming" are used in modern microprocessors [LR97] to keep multiple or deeply pipelined execution units busy by executing instructions in data-flow order, rather than sequential order. The complex variability of instruction flow in "out-of-order" processors presents a significant opportunity for undetected errors, compared to an "in-order" pipelined machine where the flow of instructions is fixed and orderly. Unfortunately, this variability also makes formal verification of such machines difficult. They are beyond the present capacity of methods based on integrated decision procedures [BD94], and are not amenable to symbolic trajectory analysis [JNB96].

This paper was inspired by Damm and Pnueli, who recently presented a pencil-and-paper proof of an implementation of Tomasulo's algorithm [DP97]. This proof is in two stages, first refining a sequential specification to an intermediate model based on partially ordered executions, and then refining this model to the implementation. The proof presented here has several advantages over this earlier work. First, it is conceptually simpler, since we refine the specification directly to the implementation, with no intermediate step, and no need to reason about second-order objects such as sets or partial orders. Second, the proof here is fully mechanically checked, using a verifier based on symbolic model checking. Although in principle, the proof of [DP97] can be carried out in a higher order prover such as PVS [ORSS94], this would require considerable elaboration. Here, the use of model checking to handle the details of the proof allows the proof to be presented here in the same form in which it is actually presented to the verifier. Third, the implementation here is at the bit level, meaning that it can be either synthesized automatically into gates and latches, or compared directly to a manual implementation by combinational equivalence checking methods [KSL95]. The disadvantage of the present proof is that it applies to only a given fixed configuration of the implementation. However, the proof is easily reverified for any desired configuration.

The compositional system used to construct the proof is similar in principle to the work of Abadi and Lamport [AL93], in that it allows the use of environment assumptions in a cyclic manner. As in [AL95], the proof relies on refinement maps. However, there are several distinctions. First, the system allows the use of synchronous processes with zero delay (*i.e.*, combinational logic), whereas [AL93] uses interleaving processes. Second, whereas in [AL95], the refinement maps are functions from implementation state to specification state, here maps in either direction may be used (though they are mostly in the opposite direction, from specification to implementation). They are expressed as processes within the system, and may be one-to-many. Third, all of the lemmas here are verified by model checking, rather than by manual proof or automated proof assistants. Hence, the degree of automation is greater (though far from complete).

Two techniques here are novel relative to previous work in compositional model checking. The first is the compositional rule, which is more general than those of [AH96, McM97] in that it allows the conjunction of multiple environment processes constraining the same signal, while still allowing cyclic environment reasoning. This ability is key to decomposing the proof of Tomasulo's algorithm. Second, the verification system exploits symmetry to reduce the number of proof obligations that need to be verified. This is key to the tractability of proof checking in the present example.

In this article, we begin with a brief overview of the compositional system and its implementation in the SMV model checking tool [McM93] (section 2). Then we introduce an implementation of Tomasulo's algorithm, and prove its correctness w.r.t. an executable specification within the compositional system (section 3). Finally, we show how SMV uses symmetries to reduce proof obligations to a tractable number (section 4), and conclude with some observations and areas for future work (section 5).

2 Proof framework

In this section, we briefly sketch the compositional system and its implementation as part of the SMV verifier.

A compositional system Let $\mathcal{S}$ be a finite set of *signals*, and $\mathcal{V}$ be a finite set of *values*. A *model* is a function $\pi : \mathcal{S} \rightarrow \mathbb{N} \rightarrow \mathcal{V}$ assigning an infinite sequence of values to each signal. A *process* is a predicate on models. It constrains the value of exactly one signal as a function of other signals, with either zero delay (a gate) or unit delay (a latch). A gate p is a predicate of the form: for all $t \geq 0$,

$$\sigma_p(t) \in f(\gamma_1(t) \ldots \gamma_k(t))$$

where $\sigma_p, \gamma_1 \ldots \gamma_k$ are signals, and f is a function $\mathcal{V}^k \rightarrow 2^{\mathcal{V}}$. That is, the set of possible values of σ_p at time t is a function of signals $\gamma_1 \ldots \gamma_k$ at time t. A latch

p is a predicate of the form

$$\sigma_p(t) \in \begin{cases} v_0 & ; t = 0 \\ f(\gamma_1(t-1)\ldots\gamma_k(t-1)) & ; t > 0 \end{cases}$$

That is, v_0 is the set of possible initial values of σ_p, and f gives the set of possible values of σ_p as a function of signals $\gamma_1 \ldots \gamma_k$ one time unit earlier. We assume the functions f never return the empty set.

Composition of processes in this system is simply conjunction. That is, the composition of two processes p_1 and p_2 is $p_1 \wedge p_2$. Now, suppose we are given two sets of processes: a specification P and an implementation Q, and we would like to prove $(\bigwedge Q) \Rightarrow (\bigwedge P)$. That is, the composition of the implementation processes implies the composition of the specification processes. Using a "compositional" approach, we would verify each component of P independently, using some small subset of Q. For example, we might prove that $q_1 \Rightarrow p_1, q_2 \Rightarrow p_2, \ldots$ and thus avoid the complexity of considering all of the processes at once. Often, however, this approach fails, as each process q_i functions correctly only given some constraints on its "environment". Absent these constraints, it will not satisfy its part of the specification, hence the compositional proof will fail.

As observed in [AL93], we must typically assume that process q_2 satisfies some specification p_2 to prove that q_1 satisfies p_1, and *vice versa.* This apparent circularity can be broken by induction over time. That is, let the notation $p \uparrow^\tau$ stand for $\forall t{\leq}\tau.\ p$, or "p holds up to time $t = \tau$". We can soundly reason as follows, by induction on τ:

$$\frac{\begin{array}{ccc} p_1 \uparrow^{\tau-1} & \Rightarrow & p_2 \uparrow^\tau \\ p_2 \uparrow^\tau & \Rightarrow & p_1 \uparrow^\tau \end{array}}{\forall t.\ (p_2 \wedge p_1)}$$

In the base case, when $\tau = 0$, note that $p_1 \uparrow^{\tau-1}$ is a tautology. Hence, we have $p_2 \uparrow^0$, and thus $p_1 \uparrow^0$, $p_2 \uparrow^1$, $p_1 \uparrow^1$ and so on. By reasoning inductively, we use each process's specification as the environment of the other, avoiding circularity. In general, given a well-founded order $\prec$ on P, when proving $p \uparrow^\tau$ we assume $p' \uparrow^\tau$ if $p' \prec p$ and $p' \uparrow^{\tau-1}$ otherwise. This rule of inference is stated formally in the following meta-theorem. We use $\mathcal{E}_p$ to stand for the environment of p and Z_p to stand for those processes which may be assumed with "zero delay" when proving p:

$$Z_p = Q \cup \{p' \in P :\ p' \prec p\}$$

Note that p itself may be in $\mathcal{E}_p$, but by definition, $p \notin Z_p$. Now, letting a set of processes stand for the conjunction of its components, and Z_p^c stand for the complement of Z_p, we have:

Theorem 1. *For all $p \in P$, let $\mathcal{E}_p \subseteq P \cup Q$. If, for each $p \in P$,*

$$(\mathcal{E}_p \cap Z_p) \uparrow^\tau \wedge (\mathcal{E}_p \cap Z_p^c) \uparrow^{\tau-1} \quad \Rightarrow \quad p$$

is valid, then $(\forall t.\ Q) \ \Rightarrow\ (\forall t.\ P)$ is valid.

Proof. Let $\sqsubset$ be the lexical order s.t. $(t,i) \sqsubset (t',i')$ iff $t < t'$ or $t = t'$ and $i \prec i'$. Then $p_i(t)$ holds by induction over $\sqsubset$.

Note that this rule allows us to assume that some environment process p_j holds for all times from 0 to τ when proving that p_i holds at $\tau + 1$ (or τ itself in the zero delay case). This allows us to take into account reachability from the initial state, using model checking techniques. Thus, the technique is quite different from proving mutually inductive invariants, as is typically done using theorem provers. Also, though the rule is similar to [AL93] in that it allows the cyclic use of environment assumptions, it differs in that it applies to synchronous processes that can have zero delay (as opposed to their interleaving, unit delay model). Other systems based on synchronous processes, such as [Kur94, GL94], do not allow cyclic assumptions. Further, the above rule allows environment assumptions to contain the conjunction of many processes constraining the same signal, whereas [McM97, AH96] do not. This ability is key to the proof presented here of Tomasulo's algorithm, in that it allows for case analysis.

Implementation in SMV In the SMV implementation of the above system, a "process" is an assignment of an expression to a signal (where the parameter t is implicit). Syntactically, a gate is written in this form:

```
signal := expression;
```

while a "latch" appears thus:

```
init(signal) := expression1;
next(signal) := expression2;
```

Here, `init(`σ`)` stands for $\sigma(0)$ and `next(`σ`)` stands for $\sigma(t+1)$. Assignments are grouped into named collections called "layers", as follows:

```
layer <name> : { <assignment1> <assignment2> ... }
```

Within a layer, a given signal may be assigned only once. Thus, an assignment (*i.e.* process) can be uniquely identified by a signal-layer pair, which is written `signal//layer`. The environment for proving a given specification component is determined by statements of the form

```
using signal1//layer1 prove signal2//layer2;
```

The well-founded order $\prec$ is determined automatically in most cases. SMV assumes that an assumption about signal σ_1 should be used to prove an assertion about σ_2 with zero delay only when there is some actual zero-delay dependency path from σ_1 to σ_2. In order to guarantee a well-founded order, however, there must be no zero-delay path from σ_2 to σ_1. This leaves an ambiguity in the case of assignments to the same signal, or to two signals on a zero-delay cycle. To resolve this, the user can enter declarations of the form

```
layer1 refines layer2;
```

indicating that `layer1` assignments should precede `layer2` assignments in the order. SMV then determines for each environment component whether it is in Z_p, and hence whether it is assumed up to τ or $\tau - 1$ when proving $p\uparrow^{\tau}$.

Verification by model checking The actual verification of each specification component p is done by symbolic model checking [BCM$^+$92, McM93]. This topic is mainly beyond the scope of this article. However, for the reader familiar with these methods, we outline one possible (but highly simplified) implementation, the understanding of which is not material to what follows. Briefly, the conjunction of the zero-delay environment assumptions $(\mathcal{E}_p \cap Z_p)$ is translated into a symbolically represented Kripke model (S_0, R_0, I_0). Here S_0 is a state invariant term derived from the gates, R_0 is a symbolic transition relation and I_0 is an initial condition (the latter two deriving from the latches). Similarly, the unit-delay environment assumptions $(\mathcal{E}_p \cap Z_p^c)$ are translated into a model (S_1, R_1, I_1), and p is translated into (S_p, R_p, I_p). Then our proof goal:

$$(\mathcal{E}_p \cap Z_p) \uparrow^{\tau} \wedge (\mathcal{E}_p \cap Z_p^c) \uparrow^{\tau-1} \quad \Rightarrow \quad p$$

is true iff the Mu-Calculus formula

$$(I_0 \wedge I_1 \wedge \mu H.\ (A \vee B \vee \mathrm{Img}^{-1}(R_0 \wedge R_1, H))) \vee (I_0 \wedge S_0 \wedge \neg(S_p \wedge I_p))$$

is empty, where $A = \mathrm{Img}^{-1}(R_0, S_0 \wedge \neg S_p)$, $B = \mathrm{Img}^{-1}(R_0 \wedge \neg R_p, S_0)$ and $\mathrm{Img}^{-1}(R, S)$ the the inverse image of S w.r.t. R. This formula can be evaluated using symbolic model checking methods, and a counterexample trace generated if the result is nonempty.

Auxiliary variables Often it is necessary to introduce auxiliary variables either as part of a specification, or part of the proof (as introduced by Owicki and Gries [OG76]). The definitions of these variables (or signals in our case) can be assumed, provided they are a "conservative extension". Formally, this means that if $\mathcal{S}_A$ is the set of auxiliary signals, Q is the implementation, and A is the set of auxiliary signal definitions, then $Q \Rightarrow \exists \mathcal{S}_A.(Q \wedge A)$. That is, for every implementation behavior, there exists a feasible valuation of the auxiliary signals. In this case, if we can prove that $Q \wedge A \Rightarrow P$, then we infer that $Q \Rightarrow \exists \mathcal{S}_A.P$. The following conditions are sufficient to ensure conservative extension:

1. Every signal in $\mathcal{S}_A$ has a unique assignment in A,
2. No assignment in Q refers to any signal in $\mathcal{S}_A$, and
3. There are no zero-delay cycles in A

In the SMV system, auxiliary variables are declared with the keyword `abstract`. The system guarantees that the above three conditions are met. As we will see, auxiliary variables allow us to use recorded history information in compositional proofs. We can then use these variables to express abstractions of the actual implementation variables. Used in specifications, they also allow us to express any regular-language property, which otherwise would not be possible.

3 Verifying Tomasulo's algorithm

In this section, we introduce Tomasulo's algorithm, and show how a proof of correctness can be constructed in the foregoing framework and mechanically checked using symbolic model checking. The proof is based on auxiliary variables and refinement maps. These maps can be viewed as an interpretation of the intended semantics of the various components of the implementation state.

Tomasulo's algorithm We consider here a pipelined arithmetic unit Q, executing a stream of operations on a register file. Tomasulo's algorithm allows execution of instructions in data-flow order, rather than sequential order. This can increase the throughput of the unit, by avoiding pipeline stalls. Each pending instruction is held in a "reservation station" until the values of its operands become available, then issued "out-of-order".

The flow of instructions is pictured in figure 1. Each instruction, as it arrives, fetches its operands from a special register file. Each register in this

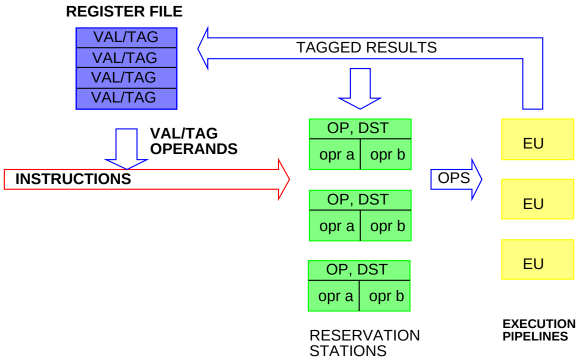

Fig. 1. Flow of instructions in Tomasulo's algorithm

```
stallout := {0,1};
if(~stallout)
  switch(opin){
    ADD_OP : {
      opr_a := r[srca];  opr_b := r[srcb];
      res := opr_a + opr_b;
      next(r[dst]) := res;
    }
    RD_OP : {
      dout := r[srca];
    }
    ...
```

Fig. 2. Specification code (partial).

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 pipelines, and when a result's tag matches one of its operands, it records the value in place of the tag. When the station has the values of all 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, since its value in a sequential execution would be overwritten.

The arithmetic unit also has instructions that read register values to an external output and write values from an external input, and has a "stall" output, indicating that an instruction cannot be executed because either there is no available RS, or the value of the register to be read is not yet available.

Verification approach The specification is a machine P that executes instructions in order as they arrive, and stalls nondeterministically. A fragment of the SMV code is shown in figure 2. Our goal is to prove that implementation Q implies specification P, with internal variables $\mathcal{S}_A$ projected. That is, $Q \Rightarrow \exists \mathcal{S}_A .\ P$.

We introduce auxiliary variables for each RS, and refinement maps relating the specification state, auxiliary variables and implementation state. Finally we verify these relations using the compositional rule. We thus decompose the proof into "lemmas" small enough to be verified by model checking.[1]

Auxiliary variables We begin this decomposition by introducing new state components. When an RS is loaded with an instruction, we record the actual values of the instruction operands and result, according to the specification machine. In SMV we have:

```
if(~stallout & IS_PIPE_OP){
   next(hist[st_choice].opr_a) := opr_a;
   next(hist[st_choice].opr_b) := opr_b;
   next(hist[st_choice].res) := res;
}
```

where `hist` is the array of auxiliary variables, and `st_choice` is the reservation station to which the arriving instruction is assigned. Because these are auxiliary variables, we cannot refer to them in the implementation. However, we can use them in refinement maps that specify components of the implementation state.

Refinement maps The first refinement map states that if a given implementation register holds a value (and not a tag), then that value must equal the "real" register value in the specification machine. In SMV, we write:

```
layer map1 :
  forall(i in REG)
    if(~ir[i].resvd) ir[i].val := r[i];
```

where `ir` is the implementation register file, and `resvd` is a bit indicating that the register holds a tag. The second maps states that, if a register holds a tag, the actual register value must equal the result of the indicated RS (an auxiliary variable):

```
forall(i in REG)
  layer  map2[i] :
    forall(j in TAG)
      if(ir[i].resvd & ir[i].tag = j){
        st[j].valid := 1;
        hist[j].res := r[i];
      }
```

Here, `st` is the RS array, `valid` is a bit indicating the station is full. The third map defines the "producer/consumer" relation between RS's. It states that if station j holds an operand value, then that value is the actual operand value. Otherwise, if station j is waiting for an operand from station k, then the result value of k is the actual operand value of j. This is the statement in SMV for the "a" operand (the "b" operand is similar):

```
forall(j in TAG)
  layer map3a[j] :
    if(st[j].valid){
      if(st[j].opr_a.valid) st[j].opr_a.val := hist[j].opr_a;
      else  hist[st[j].opr_a.tag].res :=  hist[j].opr_a;
    }
```

[1] The implementation and part of the proof text are omitted here for space reasons. See `http://www-cad.eecs.berkeley.edu/~kenmcmil` for their complete text.

Note here that `opr_a`, the "a" operand field of the RS, has three subfields: `valid` indicates that the value is available, `val` is the value, and otherwise, `tag` is the tag. The final map states that, if a value returns on the result bus with a tag j, then it must be the actual result of RS j:

```
forall (j in TAG)
  layer map4[j] :
    if(pout.tag = j & pout.valid) pout.val := hist[j].res;
```

Here, `pout` is the result bus, which has three fields: `valid`, indicating a result is present, `tag`, indicating a reservation station, and `val`, the result value.

Compositional proof Now we give the environment for proving each of the above maps. The compositional system allows us to use any map as an assumption when proving another, even in an apparently circular manner, while ensuring that the proof is inductively sound. The flow of this proof roughly follows the progress of an instruction through the machine, as follows:

1. `map2`: For each pair i, j if register i contains tag j, then the result of RS j must be the actual value of register i. This is verified using only the implementations of register i and RS j.
2. `map3` ("else" case): For each pair j, k, if RS j holds tag k as an operand, then the result value of k is the operand value of j. This is verified using `map2` and the implementations of RS's j and k.
3. `map3` ("if" case): The operand values obtained by RS j are always correct. Since operand values may come from the register file or the result bus, we need to assume both `map1` (for all registers) and `map4`. Since `map4` gives the result bus value in terms of the RS results, we also need `map3` for each RS result. We also use the implementation of RS j.
4. `map4`: Result bus values tagged j match RS j's result. To guarantee the operands used are correct we use `map3`. We also use the execution pipeline, RS j (including auxiliary vars) and the specification ALU.
5. `map1`: Values in implementation register i are correct. We use `map4` and `map3` (for each RS result) to prove this.

Note, for example, the circularity in steps 3–5. Also note that, with other refinement maps as the "environment", each of the maps is verified using the implementations of at most two registers or RS's, leaving the others as free variables. This addresses the state explosion problem and makes the verification by model checking tractable. The general principle is to break the proof down into lemmas relating pairs of array elements, rather than considering entire arrays.

As an example, consider the producer/consumer lemma (`map3`). We want to show that when RS j holds tag k as an operand, the result value of k must be the operand value of j. Suppose that an instruction is loaded into RS j, with register i as its source operand, and suppose that register i holds tag k. Now, `map2` states that the result of station k (`hist[k].res`) is in fact the correct value for register i (`r[i]`). It follows that at the next time the operand of j will be equal to the result of k (which is what we are trying to prove). The behavior of the reservation stations guarantees that this will continue to be true as long as station j holds tag k, since neither station can change until a result value for tag k returns on the result bus.

We don't need to spell out the above reasoning, however. To verify the lemma, we just apply model checking, assuming map2 holds between RS k and all registers i. In SMV, this is expressed as follows (for the "a" operand):

```
using hist[k].res//map2[i] prove hist[k].res//map3a[j];
```

If there are n registers, then in checking the lemma we are actually using a simultaneous conjunction of $n+1$ assignments to hist[k].res. One is the definition of this signal as an auxiliary variable and the others are maps that give its value as a function of each specification register. This allows us to prove the correctness of station k relative to each register i as a separate case, then to combine these cases to prove our lemma. The ability to use multiple assignments to the same signal in the environment is thus key to the tractability of the proof.

Verification by model checking As mentioned earlier, the SMV system translates each refinement map verification problem into a symbolic model checking problem. BDD-based methods are then used to either verify the map or produce a counterexample. A counterexample can indicate either that the map doesn't hold, or that insufficient environment assumptions were used.

Note that, using model checking, we can only verify a fixed finite configuration of the processor. For example, with 32-bit data, 16 registers, 16 RS's, and one four-stage execution pipeline with one operation (integer addition) the total model checking time (to verify all the refinement maps and the unit outputs) is 92 seconds, running on a 266MHz Pentium II processor, and a total of 52454 OBDD nodes used.[2]

Several factors make the verification tractable. First, by stating refinement maps and choosing environments appropriately, the number of state variables is reduced to a tractable level. One important factor in this process is "bit slicing". That is, for most of the data-related refinement maps, we treat each bit of the data path separately. This is done automatically by SMV, which prunes away those parts of the model which have no influence on the property being verified. The only exception is the verification of the execution pipeline, in which there are clearly dependencies between data bits.

4 Exploiting symmetry

The final important factor is the exploitation of symmetry. Notice that several of the refinement maps have an instance for each register/station or station/station pair (and also for each bit of the data path, if we use bit slicing). The number of lemmas to be proved is thus on the order of $R \times S \times B$ or $S \times S \times B$, where R is the number of registers, S is the number of stations and B is the number of bits. However, by exploiting symmetry we can reduce this to a small number of representative cases. Space allows only a brief synopsis of this technique here.

[2] In fact, the verification of the processor as described here fails, due to a design error. A register bypass is needed to handle the case when an instruction arrives exactly at the moment one of its operands is returning on the result bus. This requires the addition of a one-line refinement map, not described here. The verification time given is for the corrected version. Note also that if floating point arithmetic were used, more elaborate techniques would be required to verify the arithmetic at the bit level. Arithmetic verification, however, is beyond the scope of this work.

Scalarsets in SMV To express symmetries, the SMV language has been augmented with scalarset types, as used in the Murphi language [ID96]. A scalarset is a finite type, whose use is restricted such that a program's semantics is invariant under permutations of elements of the type. A scalarset type is introduced by a declaration such as:

```
scalarset TAG 0..15;
```

which creates a type called `TAG` with 16 values. Although these values are nominally in the range 0..15, no constants of a scalarset type may occur in the program. Values of scalarset type may appear only in certain symmetric constructions:

1. Two expressions of the same scalarset type may be compared for equality.
2. The index type of an array may be a scalarset. Subscripts applied to the array must be of the same type. For example, if `x` is of type `TAG` and `z` of type `array TAG of boolean`, then one could write `z[x]`, but not `z[0]`.
3. In a `forall` statement of the form:

   ```
   forall (i in type) { <statements> }
   ```

   the `type` may be a scalarset. The semantics of this statement is the conjunction of `statements` for all `i` in `type`.[3]
4. Any commutative/associative operator may be applied as a "reduction operator" over a scalarset type. For example we can take the conjunction of the elements of `z` as follows:

   ```
   &[ z[i] : i in TAG]
   ```

   Similarly, a "comprehension expression" can be formed over a scalarset type. For example, this expression {`i : i in TAG, z[i]`} denotes the set of values `i` in the type `TAG` such that `z[i]` is true.

Symmetry reduction technique The meaning of each of the above constructs is unchanged if we exchange the roles of any pair of elements of a scalarset type. As a result, the overall program semantics is invariant. So, if we have two assertions p_1 and p_2, such that one is obtained from the other by some permutation of scalarset values, then p_1 holds iff p_2 holds. Hence we need only verify p_1. SMV uses this fact in the following way: given a parameterized class of assertions to prove, SMV chooses a representative set of instances of the class, such that any instance can be reduced to one in the set by permuting scalarset values. For example, suppose that an arbiter is to acknowledge exactly one user of a resource, and that type "user" is a scalarset. The assertion `mutex[i][j]` states that users i and j are not acknowledged at the same time, if $i \neq j$. Here, two representative cases, `mutex[0][0]` and `mutex[0][1]`, suffice. All cases where $i = j$ reduce to the former, while all cases where $i \neq j$ reduce to the latter. In general, if we have k parameters of a given scalarset type, then no more than $k!$ instances are required, regardless of the type's size.

In the case of Tomasulo's algorithm, we have three scalarset types: `REG`, `TAG` and `BIT`, corresponding to registers, RS's, and bits of the data path, respectively. The "producer/consumer" lemma, for example, is a class of proof obligations of the form:

[3] As an aside for those familiar with Murphi, the use of a `forall` construct rather than an iterative `for` loop as in Murphi simplifies the rules for scalarsets, since there is no need to check for possible "side effects" between loop iterations.

```
hist[k].res[i]//map3a[j]
```

where j and k are tags, and i is a bit index, stating that when RS j points to RS k, bit i of k's result is bit i of j's operand. There are $R^2 \times B$ instances to prove, which reduce to two representatives: `hist[0].res[0]//map3a[0]` and `hist[1].res[0]//map3a[0]`. If there are 16 RS's and 32 bits, then a factor of 8192/2 in run time is saved.

Symmetry breaking In the case of Tomasulo's algorithm, the symmetry of the scalarset types is broken in several places. For example, the arithmetic unit breaks the symmetry of the data path bits, and the logic that selects an RS for the arriving instruction breaks the tag symmetry. Assignments that break the symmetry of a scalarset can be introduced with a declaration such as the following:

```
breaking (BIT)
    res := opr_a + opr_b;
```

which defines the adder function in the specification. The system ensures that a symmetry reduction over a given scalarset type is not performed when an assignment breaking that type's symmetry is used in the "environment". This allows us to localize the effect of symmetry breaking. For example, since the "producer/consumer" lemma does not depend on the adder function, we can still make use of the bit symmetry when proving it.

5 Conclusion

We have seen that it is possible using compositional model checking methods to formally verify an out-of-order processor. This was done by a direct refinement from specification to implementation, without need of an infinite-state intermediate abstraction or reasoning about partial orders. This refutes the claim in [DP97] that such an intermediate level is needed to give a concise statement of the refinement relation. In fact, the proof was possible even though the processes in the compositional system are not only finite-state, but are not even first-order expressive (much less regular-language expressive).

The proof itself is not automatic. Substantial human insight was required to decompose the proof into lemmas about small collections of state components. However, we note that the proof is at least textually short – substantialy shorter than the implementation – and that the refinement maps are a fairly natural representation of the function of the various machine components. Also note that other processor architectures (such as "in-order-completion" machines) are strictly more deterministic in their scheduling than Tomasulo's algorithm. Thus, it might be possible to reuse the present proof by refining Tomasulo's algorithm to various other architectures, and thereby save the effort of verifying them "from scratch".

There are several areas in which the present work could be extended or improved. First, the processor is unrealistic in that it has no provision for "exceptions" (caused, for example by, interrupts, arithmetic errors, or mispredicted branches). It would be useful to know, for example, if one could use a similar technique to verify a processor using "snapshots" or some other technique to roll back the processor state after an exception. Second, the verification of Tomasulo's algorithm should in principle be independent of the actual arithmetic functions used, since they have no effect on the scheduling of instructions.

If the present techniques were integrated with the uninterpreted function calculus techniques of [BD94], then the arithmetic unit might be modeled by an uninterpreted function symbol, allowing the problem of arithmetic verification to be separated. Third, note that this work and [DP97] only deal with the issue of safety and not of liveness (*i.e.*, a processor that always stalls would meet the specification). The compositional framework presented here cannot handle liveness properties (in fact, Abadi and Lamport [AL93] show that liveness assertions cannot be used as cyclic environment assumptions). The proof could, perhaps, be undertaken using assume/guarantee style temporal reasoning, which is also supported by SMV.

References

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

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

[AL95] M. Abadi and L. Lamport. Conjoining specifications. *ACM Trans. on Prog. Lang. and Syst.*, 17(3):507–534, May. 1995.

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–70, Jun. 1992.

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

[DP97] W. Damm and A. Pnueli. Verifying out-of-order executions. In D. Probst, editor, *CHARME '97*. Chapman & Hall, 1997. To appear.

[GL94] O. Grümberg and D. E. Long. Model checking and modular verification. *ACM Trans. Programming Languages and Systems*, 16(3):843–871, 1994.

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

[JNB96] A. Jain, K. Nelson, and R. E. Bryant. Verifying nondeterministic implementations of deterministic systems. In *Formal Methods in Computer-Aided Design (FMCAD '96)*, pages 109–25, 1996.

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

[Kur94] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes*. Princeton, 1994.

[LR97] D. Leibholz and R. Razdan. The alpha 21264: a 500 mhz out-of-order execution microprocessor. In *Digest of Papers, COMPCON Spring 97*, pages 28–36, 1997.

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

[McM97] K. L. McMillan. A compositional rule for hardware design refinement. In *Computer Aided Verification (CAV'97)*, pages 24–35, 1997.

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

[ORSS94] S. Owre, J. M. Rushby, N. Shankar, and M. K. Srivas. A tutorial on using PVS for hardware verification. In *Theorem Provers in Circuit Design (TPCD '94)*, pages 258–79. Springer, 1994.

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