Circular compositional reasoning about liveness

K. L. McMillan
mcmillan@cadence.com

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

Abstract. Compositional proofs about systems of many components often involve apparently circular arguments. That is, correctness of component A must be assumed when verifying component B, and *vice versa*. The apparent circularity of such arguments can be resolved by induction over time. However, previous methods for such circular compositional reasoning apply only to safety properties. This paper presents a method of circular compositional reasoning that applies to liveness properties as well. It is based on a new circular compositional rule implemented in the SMV proof assistant. The method is illustrated using Tomasulo's algorithm for out-of-order instruction execution. An implementation is proved live for arbitrary resources using compositional model checking.

1 Introduction

Compositional methods are used in conjunction with model checking to reduce the verification of large systems to a number of smaller, localized verification problems. This is necessary because model checkers are limited by the "state explosion problem". Compositional techniques have been implemented in, for example, the SMV model checking system [McM98] and the Mocha system [AHM$^+$98]. Such methods were originally described in terms of an assumption/guarantee paradigm [Pnu84]. Each component of a system was to be specified in terms of those properties which it assumes about its environment, and those properties which it would then guarantee, provided the assumptions hold. Thus, if a given process assumes P and guarantees Q, then when we compose it with a process that guarantees P (under no assumptions), we can infer Q. Much work on modular verification is based on this paradigm (*e.g.*, [GL91]). However, the limitation of this approach is that, when composing two processes A and B, it is often necessary to assume correctness of A to verify B and *vice versa*. That is, there is no place to begin the assumption/guarantee chain, because no process satisfies any useful properties in the absence of environment assumptions.

Abadi and Lamport [AL93] showed the somewhat surprising fact that under certain restrictions on the processes and the properties to be proved, if one can prove P assuming Q and prove Q assuming P, then both P and Q must hold. The restrictions were essentially as follows: 1) the processes must modify disjoint subsets of the system variables in an interleaved manner, and 2) each property must be a safety property and restrict only state variables of one process. Given these assumptions, one can show that the truth of P at time t cannot depend on Q holding at any time later than $t-1$, and *vice versa*. Thus, P and Q are proved by mutual induction over time. However, this argument would not hold if either P or Q contained operators that refer to the future. The technique is thus restricted to safety properties (in temporal logic, the properties expressible in the form Gp, where p is a past-time formula). Alur and Henzinger [AH96] extended this approach to the case where the processes and properties are both represented by synchronous Mealy machines. This relaxes the requirement of interleaving concurrency, but still allows only safety properties to be verified.

Nonetheless, mutual dependence of liveness properties does occur in real systems. Consider, for example, the problem of multiple execution units in an instruction set processor. At some times, the instruction in unit A may depend on the result of the instruction in unit B, and at other times the inverse relation may hold. Thus, in order to prove that unit A is live (always eventually produces a result), we must assume the B is live, and *vice versa*. In this paper, we introduce a compositional technique that allows for this kind of reasoning. In essence, it makes explicit the induction over time implied in the above approach, by assuming property P *only* up to time $t-1$ when proving Q at time t, and *vice versa*. This condition (Q up to $t-1$ implies P up to t) is expressible in temporal logic. Thus, the proof subgoals generated by the rule can be verified by model checking. The proof rule has been integrated with others in a proof assistant based on the SMV model checking system. This extends the technique of [McM98], which also applied only to safety properties. The integration of the circular proof rule with various reduction techniques, including *symmetry reduction*, *temporal case splitting* and *data type reduction* makes it possible to verify liveness of systems with unbounded arrays, as we illustrate, using Tomasulo's algorithm as an example.

The remainder of this paper is structured as follows. In section 2, some background is provided on the SMV system, and the various reduction techniques that it implements. In the following section, the circular compositional rule is introduced, and its soundness proved. It is generalized somewhat to allow reasoning about systems with zero-delay communication (*i.e.*, Mealy machines). The implementation of the rule in SMV is discussed, including optimizations that improve the veri-

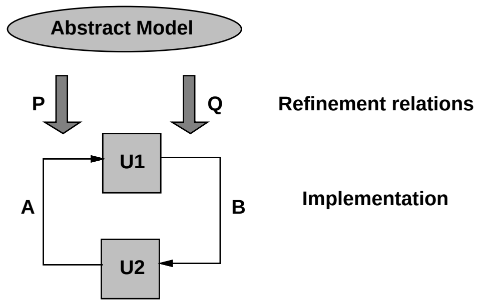

Fig. 1. Compositional refinement verification

fication complexity in common cases (including refinement verification). Finally, section 5 illustrates the application of the rule, using the example of Tomasulo's algorithm.

2 Background

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 verify systems that have cyclic data flow, such as instruction set processors. However, as implemented prior to this work [McM98], the circular rule applied only to properties expressed as *assignments*, which are a class of safety property.

2.1 Symmetry reduction

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].

2.2 Temporal case splitting

If a system contains a large, regular array structure, such as a memory, a register file, or a FIFO buffer, we can often confine the verification problem to one or two elements of the large array by means of a *temporal case splitting* rule. Using this rule, we can verify at one time 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 the model checking problem accordingly.

The temporal case splitting rule breaks the proof of $G\phi$ 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. That is, if $G(v = i \Rightarrow \phi)$ holds for all i in the range of variable v, then Gp must hold. Typically, v is an auxiliary variable which remembers the location in some large data structure which was used to store the item currently appearing at a unit output. To prove the subgoal for a given value of i, it is commonly only necessary necessary to refer to one element of the array. Thus, we effectively decomposed a large array into its individual components for the purposes of model checking.

2.3 Data type reduction

Finally, after the large data structures have been decomposed into their components, there may still be large data types, such as addresses, or data words. In this case, *data type reduction* can be used. This reduces a large (perhaps infinite) type to a small finite one, containing only one or two values that are 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 operations. This abstraction is conservative, in that any property that is valid in the reduced structure is also valid in the original structure. For example, when comparing two abstract values for equality, the result is the "unknown" value $\perp$. Any property proved in the reduced structure is true in the original. On the other hand, it is possible that the truth value of a property in the reduced structure will be $\perp$, in which case we obtain no information. By using such a reduction, in combination with temporal case splitting, symmetry reduction and model checking, it is possible to properties of systems with unbounded arrays [McM99].

3 Circular compositional rule

Now we will consider the problem of extending circular compositional reasoning to liveness properties. Using the framework of linear temporal logic (LTL), we will formalize an inference rule for circular compositional reasoning, and prove its soundness. We begin by defining our model and logic. Note that there is no notion of process or process composition in this system. As in, for example, TLA [Lam91], a process is simply a temporal property.

We use the standard linear temporal logic. That is, to begin with, a *model* σ is an infinite sequence of *states* $\sigma_0, \sigma_1, \ldots$. A *formula* is either an *atomic proposition*, or one of

$$\neg p,\ p \wedge q,\ p\ U\ q,\ Xq$$

where p and q are formulas. The exact definitions of *state* and *atomic proposition* are not relevant here, so we omit them. We use the notation $(\sigma, i) \models p$ to indicate that model σ satisfies formula p in the ith state. This relation must satisfy the following (standard) definition:

Definition 1. *For all models σ, and formulas ϕ, ψ:*

1. $(\sigma, i) \models \neg\phi$ iff $(\sigma, i) \not\models \phi$,
2. $(\sigma, i) \models \phi \wedge \psi$ iff $(\sigma, i) \models \phi$ *and* $(\sigma, i) \models \psi$,
3. $(\sigma, i) \models \phi\ U\ \psi$ iff *there exists* $j > i$ *such that* $(\sigma, j) \models \psi$, *and for all* $i \leq k < j$, $(\sigma, k) \models \psi$,

4. $(\sigma, i) \models X\phi$ iff $(\sigma, i+1) \models \phi$.

We will say that $\sigma \models \phi$ when $(\sigma, 0) \models \phi$ (that is, we take the initial interpretation of formulas) and $\models \phi$ (*i.e.*, ϕ is valid) when $\sigma \models \phi$ for all models σ. In addition, we will use the usual abbreviations, for example, Fp (eventually p) is equivalent to true $U \neg p$ and Gp (henceforth p) is equivalent to $\neg F \neg p$.

Now, suppose that we have a collection of formulas P, and we would like to prove Gp for all $p \in P$ (that is, all the properties p hold at all times). We first fix a well founded order $\prec$ on the formulas in P (that is, a transitive, irreflexive relation with no infinite backward chains). The intuitive meaning of this order is that, if $p \prec p'$ then we may assume p up to time t when proving p' at time t, otherwise we may assume p only up to time $t-1$. For any proposition $p \in P$, we will denote by Δ_p the set of propositions that are assumed up to time t when proving p, and by Θ_p the set that are assumed at time t. By necessity, every element of Θ_p must be less than p, according to $\prec$. However, any $p' \in P$ (including p itself) may be an element of Δ_p. This is what allows us to construct (apparently) circular arguments. The notion that "p' up to time $t-1$ implies p at time t" can be expressed in as a formula in the logic, using the "until" operator, as in the following theorem:

Theorem 1. *Given sets Γ, P of formulas, a well founded order $\prec$ on P, and, for all $p \in P$, sets $\Delta_p \subseteq \Theta_p \subseteq P$, such that $q \in \Delta_p$ implies $q \prec p$, if*

$$\models \Gamma \Rightarrow \neg(\Delta_p \ U \ (\Theta_p \wedge \neg p))$$

for all $p \in P$, then $\models Gp$ for all $p \in P$.

Proof. Let $\sqsubset$ be the lexical order on $\mathbb{N} \times P$ such that $(t,p) \sqsubset (t',p')$ *iff* $t < t'$ or $t = t'$ and $p \prec p'$. This order is well founded. Now, let σ be some model of Γ. Suppose towards a contradiction that for some pair (t,p), where $t \in \mathbb{N}$ and $p \in P$, that $(\sigma, t) \not\models p$. Since $\sqsubset$ is well founded, we may suppose without loss of generality that (t,p) is a minimal such pair w.r.t. $\sqsubset$ (in a well founded order, every nonempty set has at least one minimal element). It follows that, for all $t' < t$, $(\sigma, t') \models \Delta_p$ and for all $t' \leq t$, $(\sigma, t') \models \Theta_p$. Thus, $\sigma \not\models \neg(\Delta_p \ U \ (\Theta_p \wedge \neg p))$, a contradiction. □

Thus, to prove Gp for all p, it suffices to prove $\Gamma \Rightarrow \neg(\Delta_p \ U \ (\Theta_p \wedge \neg p))$ for all p. Note that the latter are linear temporal formulas whose validity we can verify by model checking methods. This might be viewed as problematic, in that known algorithms for LTL model checking are exponential in the formula size. However, we will observe shortly that, in certain common cases, the problem can be reduced to the model checking of a CTL formula of the same size. In particular, for the restricted class of properties covered in [McM98], we recover the same complexity.

3.1 Proof graphs

The SMV system applies the above theorem in the context of a *proof graph*. This framework supports both assumption/guarantee style reasoning, where we assume A at all time to prove B at all time, and circular compositional reasoning, where we assume A only up to time t or $t-1$.

A proof graph is a directed graph (V, E), where the vertices in V are the propositions to be proved, and an edge $(p, p') \in E$ indicates that proposition p is to be assumed when proving p'. A subset $E^+ \subseteq E$ of the edges are identified as *unit delay edges*. If $(p,p') \in E^+$, then p is to be assumed only up to time $t-1$ when proving p' at time t. The graph $(V, E \setminus E^+)$, obtained by deleting the unit delay edges, must be well founded (that is, have no infinite backward paths). In the finite case, this is equivalent to saying it is acyclic.

However, the complete proof graph (V, E) may in fact be cyclic. It is required only that every cycle contains at least one unit delay edge. Each strongly connected component in the graph corresponds to an application of the circular compositional rule. Thus, in particular, every formula on a cycle must be of the form Gp (so that we may apply the circular rule). Formulas not on a cycle may be of any form, however.

Now, suppose some proposition $Gp \in V$ is on a cycle. Let C be the strongly connected component containing Gp. Let Δ_p be the set of all propositions $Gp' \in C$ such that $(Gp', Gp) \in E$. Finally, let Θ_p be the subset of these such that $(Gp', Gp) \notin E^+$. That is, within a strongly connected component of the graph, a predecessor via a unit delay edge is assumed up to time $t-1$, while a predecessor via a normal edge is assumed up to time t. Finally we will denote by Γ_p the set of propositions q, such that $(q, Gp) \in E$ and $q \notin C$. That is, this is the set of predecessors not on a cycle with Gp. Finally, if a proposition q is not on a cycle, then let Γ_q be the set of q' such that $(q', q) \in E$.

Theorem 2. *Let $\mathcal{C}$ be the union of the strongly connected components of (V, E). If*

- *for all $Gp \in \mathcal{C}$: $\sigma \models \Gamma_p \Rightarrow \neg(\Delta_p\ U\ (\Theta_p \wedge \neg p))$*
- *for all $q \notin \mathcal{C}$: $\sigma \models \Gamma_q \Rightarrow q$*

then for all $p \in V$: $\sigma \models p$.

Proof. Let $\sqsubset$ be the transitive, irreflexive closure of E (that is, $p \sqsubset q$ iff there is a path $p \rightarrow q$ in (V, E) but no path $q \rightarrow p$). This is a well founded order. Now, suppose that $q \in V$ is not on a cycle. Then every predecessor q' of p satisfies $q' \sqsubset q$. Thus, by induction we may assume $\sigma \models \Gamma_q$, hence by *modus ponens*, $\sigma \models q$. Suppose on the contrary that $q = Gp$ is in strongly connected component C. Every predecessor q' of q not in C must satisfy $q' \sqsubset q$ (else there would be a path $q \rightarrow q'$, a contradiction). Thus, by induction we may assume $\sigma \models \Gamma_q$, hence by *modus ponens*, $\sigma \models \neg(\Delta_p U(\Theta_p \wedge \neg p))$. Now, for all $p', p'' \in C$, let $p' \prec p''$ iff $p'(E \setminus E^+)^* p''$. This is a well founded order satisfying the conditions of theorem 1. Thus $\sigma \models p$. □

In order to apply the circular compositional rule in SMV, we have only to supply the set of properties to be proved and the proof graph. From these, the above theorem can be used to construct a sufficient set of proof obligations in the form of LTL formulas.

4 Implementing the rule in SMV

In the SMV language, a property is specified in the following syntax:

```
<name> : assert <formula>;
```

When specifying the proof graph, properties are referred to by their names. An arc (p_1, p_2) in the proof graph is specified as follows:

```
using <n1> prove <n2>;
```

where `n1` and `n2` are the respective names of properties p_1 and p_2. A unit delay arc is specified by putting the assumption in parentheses, as follows:

```
using (<n1>) prove <n2>;
```

SMV verifies that every cycle in the proof graph is cut by a unit delay arc.

4.1 A simple example

As an example, consider a "token ring", where at each time step the token's owner passes the token to its successor in the ring. Here is a simple representation of this protocol in the SMV language:

```
forall(i in 0..(N-1)){
  t[i] : boolean;
  init(t[i]) := (i=0) ? 1 : 0;
  next(t[i]) := t[i-1 mod N];
}
```

Here, the boolean variable `t[i]` indicates that process `i` is holding the token. Initially, process 0 holds the token. Suppose we would like to prove compositionally that every process holds the token infinitely often. This can be specified in SMV as follows:

```
forall(i in 0..(N-1))
   p[i] : assert G F t[i];
```

This specifies a property `p[i]` for every process `i`, stating that `t[i]` is true infinitely often. Of course, any `t[i]` will eventually hold if its predecessor `t[i-1 mod N]` eventually holds. Thus, we could write the following proof graph:

```
forall(i in 0..(N-1))
  using t[i-1 mod N] prove t[i];
```

However, SMV would rightly complain that this proof graph is cyclic. We need to introduce unit delay edges in the graph in some way that breaks the cycle. Now, since passing the token involves one unit of delay, one might infer that it is sufficient to assume `F t[i-1 mod N]` at time $t-1$ to prove `F t[i]` at time t. Unfortunately, this argument works at all times except $t = 0$ (when there is no time $t-1$). We can only use a unit delay arc to prove a property if it holds at $t = 0$ without assumptions. Fortunately, that is true of process 0, which holds the token initially. That is, `F t[0]` is true initially, independent of what the other processes do. Also fortunately, a unit delay arc from `p[N-1]` to `p[0]` is sufficient to break the cycle. Thus, we use the following proof graph:

```
forall(i in 1..(N_1))
  using p[i-1] prove p[i];
using (p[N-1]) prove p[0];
```

The proof subgoals resulting from the circular compositional rule are:

$$\neg((F\ t[i-1])\ U\ (F\ t[i-1] \wedge \neg F\ t[i]))$$

for all $0 < i < N$, and

$$\neg((F\ t[N-1])\ U\ (\neg F\ t[0]))$$

Each of these can be verified locally, using just the definition of one variable `t[i]`, and abstracting away the other processes. Of course, this reduction is unnecessary in practice, since the number of states in the entire system is only N in this case. However, it demonstrates the principle of decomposing liveness properties using the circular compositional rule. In section 5, we'll see a less trivial example.

4.2 Checking the subgoals efficiently

The proof subgoals generated by the circular compositional rule are LTL formulas. A disadvantage of using LTL, as opposed to the branching time logic CTL, is that the model checking complexity for LTL is exponential in the formula size[1], whereas for CTL it is linear time. However, we will see in this section that for significant special cases, the circular compositional rule generates LTL proof subgoals that can be translated linearly into CTL. This is especially significant in the case of "refinement verification", in which the compositional rule is used to show that one system implements a more abstract model *vis à vis* a refinement relation. In this case, the formulas resulting from the circular compositional rule can be quite large, since the set of formulas P to be proved by the rule can consist of a large fraction of the system description itself.

Model checking for LTL The model checking problem for LTL is to determine whether every infinite path in a finite state graph is a model of a given formula. This can be done by adding a new boolean state variable to the model, corresponding to each occurrence of a temporal operator in the formula [BCM+92]. Note, the cost of this is to increase the state space size by a factor exponential in the number of temporal operators in the formula. Constraints over these variables are then added to the transition relation, corresponding to the fixed point characterizations of the temporal operators. For example, if we denote by v_p the state variable associate with subformula p, and if $p = Fq$, then we have:

$$v_{Fq} \Leftrightarrow q \vee X v_{Fq}$$

Note, Xv_{Fq} denotes the value of the state variable v_{Fq} at the next time. Thus, we can view the above equivalence as a constraint on the transition relation. In fact, the set of states satisfying Fq is the least set satisfying the above equivalence. This can be obtained by adding a "fairness constraint" $GF(v_{Fq} \Rightarrow q)$ to the model. This, in effect rules out the case where Fq is true forever, but q never holds. The above two constraints are easily shown to be equivalent to $v_{Fq} \Leftrightarrow Fq$. This means that, having added the constraints to the model, we can substitute v_{Fq} for every occurrence of Fq in the formula. Having performed this substitution for every temporal operator in the formula, there remains only a boolean formula, which must be true at the initial state. The original formula is true (for all paths in the original model) if and only if the derived boolean formula is true (initially) for all (fair) paths in the extended model. The existence of such a path can be determined in time linear in the size of the (extended) state graph, and in the number of fairness constraints. In fact, since the derived formula is trivially in CTL (since it contains no temporal operators), we can simply use a CTL model checker for this purpose. If we use a symbolic CTL model checker [BCM+92], this process is made even simpler, since the transition relation in this case is expressed as a boolean constraint on the current and next values of the state variables. Hence the transition constraints derived from the LTL formula have simply to be conjoined with the transition relation of the original model.

As noted above, this procedure is linear in the state graph size, but exponential in the formula size, since the model is expanded by an exponential factor in the formula size. It should be observed that the state graph itself is usually derived from a more compact representation, such as a program text, or a boolean formula characterizing its transition relation. Thus, in practice, both CTL and LTL model checking are exponential in the number of state variables in the model. However, it would be desirable, where possible, to eliminate the exponential dependence on formula size, and restrict the complexity of the problem to a function of the number of state variables. In certain common cases, including refinement verification, we can do this by translating the LTL formula linearly into a slightly extended version of CTL, and thus avoiding the above exponential construction.

[1] Exponential in practice. The problem is actually PSPACE complete.

Extending CTL As noted by Emerson [EH86], CTL can be viewed as subset of the general branching time logic CTL*. In CTL*, we define a *path formula* to be any LTL formula over state formulas, and a *state formula* to be either an atomic proposition, or a path formula prepended with a path quantifier, either E or A. A state formula beginning with E (respectively A) is true in a given state if and only if the given path formula is true for some path (respectively, for all paths) beginning with that state. CTL is the subset of this logic where the only path formulas allowed are Xp and $p\ U\ q$, where p and q are state formulas. However, we can extend the allowable path formulas slightly, without changing the model checking complexity of CTL.

In particular, let a *transition formula* be any boolean combination of state formulas and formulas of the form Xp, where p is a state formula. Thus, for example, if p and q are state formulas, then $p \Rightarrow Xq$ is a transition formula, while XXq is not a transition formula, since Xq is not a state formula. Every transition formula characterizes a set of transitions of the model. Note that, if the model contains non-boolean state variables, we can also extend the notion of transition formula to contain terms that refer to the "next" value v' of a variable v. Thus, for example, $v' = v + 1$ would be a valid transition formula, if logic includes the function symbol '+' and the predicate '='. This extension does not alter any of the results below.

Now, let's extend the logic CTL, by defining it to be the subset of CTL*, restricted to path formulas which are either transition formulas, or formulas of the form $p\ U\ q$, where p is a transition formula and q a state formula. We can easily see that model checking remains linear time with these extended modalities, and provide symbolic algorithms for them. First we rewrite a given transition formula p to a binary predicate $p(x, x')$ over states, where we replace every occurrence of state formula q by $q(x)$ and every occurrence of Xq by $q(x')$. Now, following the example of [BCM+92], we can use the relational Mu Calculus to characterize the set of states satisfying Ep and Ap, where p is a transition formula:

$$\begin{aligned} Ep &= \lambda x.\ \exists x'.\ (R(x, x') \wedge p(x, x')) \\ Ap &= \lambda x.\ \forall x'.\ (R(x, x') \Rightarrow p(x, x')) \end{aligned}$$

where R is the transition relation of the model. Clearly, these two sets can be computed in time linear in the size of the relation R and in the number of operators in p (given that we have already computed the sets $q(x)$ for all state formulas q in p). Further, the modalities $A(pUq)$ and $E(pUq)$, where p is a transition formula, obey the following fixed-point characterizations:

$$\begin{aligned} E(pUq) &= \mu Q.\ q \vee E(p \wedge Xq) \\ A(pUq) &= \mu Q.\ q \vee A(p \wedge Xq) \end{aligned}$$

Note that in the above $p \wedge Xq$ is a transition formula (since q is a state formula). Thus the sets $E(p \wedge Xq)$ and $A(p \wedge Xq)$ can be computed according to the above characterization. These fixed points give us a direct algorithm (using the Tarski fixed point theorem) for evaluating the "until" modalities. Although a naivë explicit-state application of this algorithm would be quadratic time, a linear time algorithm can be obtained by using the same idea as in the CES algorithm [CES86], or, in the problematic case of $A(pUq)$, simply using backward BFS, keeping a count of the number of outgoing edges of each state satisfying $p \wedge Xq$, so that each state need be visited only once. Thus, we have both symbolic and linear time explicit state algorithms for model checking the extended version of CTL.

Checking the circular compositional rule Each proof subgoal generated by the circular compositional rule is of the form:

$$\Gamma \Rightarrow \neg(\Delta\ U\ (\Theta \wedge \neg\phi))$$

Here, Γ, Δ and Θ are (the conjunctions of) sets of formulas. Intuitively, the formulas in Γ are assumed to hold *initially*, those in Δ are assumed to hold *up to time* $t-1$, and those in Θ are assumed to hold *up to time* t when proving ϕ at time t. We will consider the special case where the formulas in Δ and Θ are all transition formulas (as defined above), and the formulas in Γ are all either boolean combinations of state variables (*i.e.*, initial conditions), or formulas of the form Gp, where p is a transition formula (in other words, transition invariants). These are the typical forms of formulas that occur in refinement verification [McM98]. In this special case, we will see that the model checking problem for formulas of the above form can be translated linearly to a CTL model checking problem.

To begin with, we can eliminate the formulas in Γ, by simply converting them into constraints on the initial state or the transition relation of the model. That is, suppose we let $M_{I,R}$ denote a finite Kripke model whose initial states are characterized by the boolean formula I, and whose transition relation is characterized by the transition formula R. Then, if p is a boolean formula and q is a transition formula, we have

$$M_{I,R} \models (p \wedge Gq) \Rightarrow \psi \ \text{ iff } \ M_{I \wedge p, R \wedge q} \models \psi$$

After eliminating Γ in this way, we are left with the formula $\neg(\Delta\ U\ (\Theta \wedge \neg\phi))$. Note that this formula must be true for all paths in the model starting with an initial state. That is, implicitly, there is an A path quantifier in front of this formula. Pushing this A path quantifier through the initial negation, we obtain

$$\neg E(\Delta\ U\ (\Theta \wedge \neg\phi))$$

Note that, under the assumption that all the subformulas are transition formulas, this formula is in our extended version of CTL. Thus, it can be checked in time that is linear in the model size. In fact, we can do slightly better than this in practice, since when computing the fixed point for the EU modality, we can restrict the model to the states reachable from initial states by transitions satisfying Δ. Note that in general, in CTL model checking, we must restrict the computation to the set of states on infinite fair paths. However, in the case when the transition relation is known to be total, and where there are no fairness constraints, we know that every state is on an infinite fair path. In this case, checking the above formula reduces to reachability analysis (*i.e.*, the formula is true iff there is no path from an initial state to a state satisfying $\neg(\Theta \wedge \neg\phi)$ using only transitions satisfying Δ). Note also, that in the case where one of the subformulas is not a transition formula, we can reduce it to one by simply adding new state variables for the disallowed operators, and substituting the state variables for the corresponding subformulas. For example, if the set Δ contained the formula $p \Rightarrow XFq$, we could replace Fq by v_{Fq}, adding constraints $v_{Fq} \Leftrightarrow q \vee Xv_{Fq}$ and $GF(v_{Fq} \Rightarrow q)$ to the model, and substituting V_{Fq} for Fq in the formula, to obtain the transition formula $p \Rightarrow Xv_{Fq}$. Thus, even in the case where the proof subgoals do not exactly fit the linear time case, we may still substantially reduce the number of state variables added to the model (the basis of the exponential complexity of model checking LTL). All of these optimizations are implemented in SMV. As a result, when using the circular rule, the assignments that comprise the system description do not contribute to the exponential complexity in formula size associated with LTL model checking.

5 Verifying a version of Tomasulo's algorithm

We now consider a concrete example of how the circular compositional rule can be used to prove liveness of a system compositionally (that is, using only localized model checking). The functional verification of this design is described in [McM99]. Here, we consider only the verification of liveness.

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[2] is pictured in figure 2. Each instruction, as it arrives, fetches its two

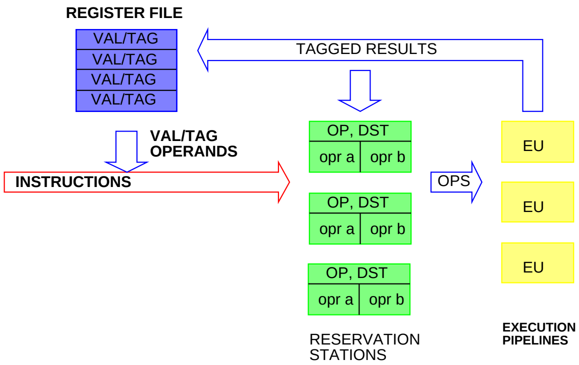

Fig. 2. Flow of instructions in Tomasulo's algorithm

operands from a special register file. Each register in this file holds either an actual value, or a "tag" indicating the reservation station that *will* produce the register value when it completes. The instruction and its operands (either values or tags) are stored in a reservation station (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

We would like to prove that an instruction in any given reservation station eventually terminates. As in the proof of functional correctness [McM99] the first step in the proof is to break the verification problem into two lemmas, to which we will apply the circular compositional rule. The first lemma

[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.`

states that the operands required by an instruction in a given reservation station eventually arrive at that reservation station. The second states that the result of an instruction in a given reservation station eventually arrives on the result bus (and hence the reservation station is eventually cleared). We use operand liveness lemma to prove result liveness and *vice versa*.

Here is the SMV specification for the operand liveness lemma: (for the `opra` operand):

```
forall (i in TAG)
  live1a[i] : assert G (st[i].valid -> F st[i].opra.valid);
```

Here, `TAG` is the type of reservation station indices. Thus, for all reservation stations i, the lemma `live1a[i]` states that if station i is `valid` (contains an instruction) then its `opra` operand will eventually be valid (hold a value and not a tag). A similar lemma is stated for the `oprb` operand.

The result liveness lemma is just as simply stated:

```
forall (i in TAG)
  live2[i] : assert G (st[i].valid -> F ~st[i].valid);
```

That is, for all reservation stations i, the lemma `live2[i]` states that if if station i is `valid` (contains an instruction), it is eventually cleared. Note that the reservation station is cleared when its result returns on the result bus.

5.3 Path splitting

We cannot apply model checking directly at this point to verify the two lemmas, because the state space is too large (except for very small instances of the design). The state space explosion in this case is chiefly due to the large arrays in the design, such as 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 taken by a given data item (operand or result) from its producer to its consumer.

As an example, consider a source operand of an instruction stored in reservation station k at a given time. We will call this reservation station the *consumer* of the operand. This source operand is also the destination operand of some previous instruction, assigned to some reservation station i, which we will call the *producer* of the operand. Note that the consumer reservation station holds a tag, which is, in fact, a pointer to the producer reservation station. Once the operand is produced, it is stored in some register j, which is both the destination register of the producer, and source register of the consumer.

This analysis suggests a case split which we will reduce the size of the verification problem to just two reservation stations and one register. To prove liveness of operands arriving at consumer reservation station k, we split cases based on the producer reservation station i (as indicated by the "tag" of the operand) and on the intermediate register j (as indicated by the the source operand index of the 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. 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 live1a[k][i][j] of live1a[k]
    for st[k].opra.tag = i & aux[i].srca = j;
```

For all consumer reservation stations k, this breaks the operand liveness 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 `aux[i].srca` to the model to remember the source operand register for reservation station i, since the implementation does not store this information (such use of auxiliary variables is described in [McM98]).

Naturally, in order to prove that operands of consumer reservation station k eventually arrive, we have to assume that the producer reservation station eventually produces a result. On the other hand, clearly we will have to assume that the operands of an instruction eventually arrive in order to prove that it eventually produces a result. This is where the circular compositional rule comes into play. Note that the producer instruction always enters the machine at least one time unit before the consumer instruction (in fact, this is how the consumer comes to hold a tag pointing to the producer). Thus to prove that the consumer operand eventually arrives for instructions arriving at time t, it is sufficient to assume that results eventually arrive for producer instructions arriving up to time $t-1$. Thus, in SMV, we add the following arcs to the proof graph (for the `opra` operand):

```
forall (i,k in TAG; j in REG)
  using (live2[i]) prove live1a[k][i][j];
```

That is, when proving `live1a[k][i][j]` at time t, we assume `live2[i]` up to time $t-1$.

Verifying each case requires only one register and two reservation stations in the model. In fact, default data type reductions [McM99] automatically eliminate all but reservation stations i, k and register j, and also reduce the types of register indices and tags to two and three values respectively. Thus, we have effectively broken the large data structures down into their components for verification purposes. As in [McM98], symmetry techniques automatically reduce the n^3 cases we need to verify to just 2 representative cases (for $i = k$ and $i \neq k$).

For the result liveness lemma, we again consider the possible paths of a data item from producer to consumer. In this case, operands are sent from reservation station i to some execution unit j, where a given operation is applied. The result is then returned on the result bus tagged for reservation station i, which in turn clears the reservation station. Thus, to prove liveness of results for reservation station i, we would like to split into cases based on the execution unit. This presents a problem, since at the time the instruction enters the reservation station, the execution unit is not yet determined. Nonetheless, it is possible to split cases on a future value of variable, using the following declaration:

```
forall(i in TAG; j in EU)
  subcase live2[i][j] of live2[i]
    for (aux[i].eu = j) when st[i].issued;
```

This creates one case of `live2[i]` for each execution unit j. Each case states that a result must eventually arrive only at those times when the instruction will be in execution unit j at the next time the reservation station is found to be in the `issued` state. Note that `when` is a derived temporal operator. The formula p `when` q is true iff p holds at the next time that q holds (and by definition p `when` q is true in the case when q never holds). The formula p `when` q is just an abbreviation for $\neg((\neg q)\ U\ (q \wedge \neg p))$. SMV recognizes that at any time `v = i when q` must be true for at least one value of i, and thus that the given set of cases is complete.

To prove that the instruction in a given execution unit eventually terminates, we must assume that its operands eventually arrive. Thus, we add the following arcs to the proof graph:

```
forall (i in TAG; j in EU)
  using live1a[i], live1b[i] prove live2[i][j];
```

Note that here, we assume the operand liveness lemma up to time t when proving the result liveness lemma up to time $t-1$. This is allowable, since all the cycles in the proof graph are broken by a unit delay arc.

5.4 Issues of fairness

Our implementation contains a number of arbiters, whose functions are, for example, to choose the next reservation station to issue to an execution unit, the next execution unit to use, and the next execution unit which is to complete, by returning its result on the result bus. When proving functional correctness of the design, the arbitration choices can be left completely free (in other words, we use an abstraction where the arbiter outputs are just free variables). However, to prove liveness, we require that the arbiters be fair (for example, if an instruction in a reservation station is ready to be issued, it must eventually be chosen for execution). While it would be possible to introduce an actual implementation of the arbiter into the proof, this would break the symmetry of reservation stations and execution units. Instead, we simply introduce an assumption of fairness. One way to do this in SMV is as follows:

```
forall (i in TAG)
  issue_fair[i] : assert G F (st_ready[i] -> st_issue[i]);
```

The assertion `issue_fair[i]` says that if reservation station i remains in the ready state, it eventually is issued to an execution unit. Because this assertion does not break the symmetry of the index types, it allows us to use symmetry and data type reductions to verify the design generically, for any number of reservation stations and execution units. We must, of course, also prove that any actual arbiter implementation satisfies the fairness assertion. This can be done locally, however, so that the cost of breaking the symmetry is not as high. In general, because of the symmetry breaking nature of arbiters, it appears to be most practical to abstract them away by stating fairness properties, and then verifying the arbiters separately.

To use the fairness assertion in proving result liveness, we add the following to the proof graph:

```
forall (i in TAG; j in EU)
  using issue_fair[i] prove live2[i][j];
```

Note that the fairness assertion `issue_fair[i]` does not enter into the circular compositional rule, since it is not on a cycle in the proof graph. Rather, it is assumed initially (hence, for all time, since it is a G property) when proving `live2[i][j]`.

5.5 Summary

The result of applying the above described proof decomposition is a set of proof subgoals that can be solved by model checking (and which, if false, will yield sequential counterexamples). Our implementation of Tomasulo's algorithm is shown to satisfy the given lemmas for an arbitrary (finite or infinite) number of registers, reservation stations, and execution units (assuming that the arbiters are fair).[3] All told, there are 5 model checking subgoals, with a maximum of 20 state variables. The overall verification time (including the generation of proof goals and model checking) is 3.2 CPU seconds (on 233MHz Pentium II processor).

6 Conclusions

We have seen that mutual dependence of liveness properties does occur in practice, for example in the case of processors with multiple execution units. In such a situation, the standard assumption/guarantee approach to compositional verification cannot be followed. However, we can use an

[3] The proof is also independent of the data word size and ALU functions supported, since liveness does not depend on the data path.

appropriate circular compositional rule, combined with model checking, to prove such mutually dependent liveness properties by induction over time. Such a rule was obtained by extending the technique for safety verification in [McM98]. Since the resulting proof goals are linear temporal formulas, some care must be taken to insure that assertions deriving from the system description itself do not contribute to an exponential state space explosion through the addition of "tableau" variables. This has been accomplished in part through an extension of the branching time logic CTL to include modalities containing transition formulas.

The circular compositional rule, as implemented in the SMV system, fits into a framework including temporal case splitting, symmetry reduction and data type reduction. This makes it possible to verify liveness of systems with large or unbounded arrays through model checking. To make this possible, the temporal case splitting facility in SMV was extended to allow case splitting on the future value of a variable.

In the case of Tomasulo's algorithm, the structure of the liveness proof, including the case splits, followed almost exactly that of the functional proof [McM99]. The main difference was that to prove liveness it was necessary to add fairness assumptions about arbiters. On the other hand, no reference to data path logic was necessary, since liveness did not depend on the data path. Note also that the proof itself is quite succinct, consisting of only a few one-line temporal assertions, with corresponding proof graph declarations and case splitting declarations. This is due to the fact that model checking was used to handle the details of the proof. This in turn was made possible by case splitting, and by using data type reduction, in effect a property-specific abstraction.

References

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

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

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

[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.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[EH86] E. A. Emerson and J. Y. Halpern. "Sometimes" and "Not Never" revisited: on brancheing versus linear time temporal logic. *J. ACM*, 33(1):151–178, 1986.

[GL91] Orna Grumberg and David E. Long. Model checking and modular verification. In J. C. M. Baeten and J. F. Groote, editors, *CONCUR 91*, volume 527 of *LNCS*. Springer-Verlag, August 1991.

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

[Lam91] L. Lamport. The temporal logic of actions. Research report 79, Digital Equipment Corporation, Systems Research Center, Dec. 1991.

[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.

[McM99] K. L. McMillan. Verification of infinite state systems by compositional model checking. http://www-cad.eecs.berkeley.edu/~kenmcmil/papers/1999-01.ps.gz, February 1999.

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

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