You Assume, We Guarantee: Methodology and Case Studies[*][**]

Thomas A. Henzinger Shaz Qadeer Sriram K. Rajamani

EECS Department, University of California at Berkeley, CA 94720-1770, USA
Email:{tah,shaz,sriramr}@eecs.berkeley.edu

Abstract. Assume-guarantee reasoning has long been advertised as an important method for decomposing proof obligations in system verification. Refinement mappings (homomorphisms) have long been advertised as an important method for solving the language-inclusion problem in practice. When confronted with large verification problems, we therefore attempted to make use of both techniques. We soon found that rather than offering instant solutions, the success of assume-guarantee reasoning depends critically on the construction of suitable abstraction modules, and the success of refinement checking depends critically on the construction of suitable witness modules. Moreover, as abstractions need to be witnessed, and witnesses abstracted, the process must be iterated. We present here the main lessons we learned from our experiments, in form of a systematic and structured discipline for the compositional verification of reactive modules. An infrastructure to support this discipline, and automate parts of the verification, has been implemented in the tool Mocha.

1 Introduction

Formal verification is a systematic approach for detecting logical errors in designs. The designer uses a language with mathematical semantics to describe the design, which is then analyzed for correctness with respect to a specification. We refer to the design being analyzed as the implementation. The verification problem is called *refinement checking* when the specification is a more abstract design. For a trace semantics, the refinement-checking problem is PSPACE-hard in the size of the implementation description and in the state space of the specification. Not surprisingly, algorithms for refinement checking are exponential in the size of the implementation description and doubly exponential in the size of the specification description.

There are two general classes of techniques for combating this state-explosion problem. Type-1 techniques focus on improving algorithms, often developing heuristics that target specific application domains, such as symbolic methods for synchronous hardware designs, and partial-order methods for asynchronous communication protocols. Type-2 techniques focus on dividing the verification task at hand into simpler tasks, often making use of the compositional structure of both implementation and specification, such as assume-guarantee methods for proof decomposition. While type-1 techniques can be applied fully automatically and improve the efficiency of formal verification, they need to be

[*] An abbreviated version of this paper appeared in the *Proceedings of the 10th International Conference on Computer-aided Verification* (CAV), Lecture Notes in Computer Science 1427, Springer-Verlag, 1998, pp. 440–451.

[**] This work is supported in part by ONR YIP award N00014-95-1-0520, by NSF CAREER award CCR-9501708, by NSF grant CCR-9504469, by ARO MURI grant DAAH-04-96-1-0341, and by the SRC contract 97-DC-324.041.

complemented by type-2 techniques in order to make the approach fully scalable. Type-2 techniques, however, require substantial assistance from human verification experts, and their systematic application in nontrivial situations remains somewhat of a black art.

We are developing a formal-verification tool, called Mocha [AHM+98], which is based on the system description language of *reactive modules* [AH96]. Reactive modules permit the modular and hierarchical description of heterogeneous systems, and have been designed explicitly to support type-2 techniques such as assume-guarantee reasoning. In this paper, we present the experiences and results of our attempts to make use of type-2 techniques within Mocha in a disciplined and systematic way. We report on a methodology that has led us to success in verifying a hardware circuit that implements Tomasulo's algorithm, and a sliding-window communication protocol. Since the description of these examples would require more space than is available in these proceedings, we illustrate our methodology, instead, on a circuit that implements a simple three-stage pipeline.

We now briefly outline our methodology, which approaches a refinement-checking problem of the form $P_1||P_2 \preceq Q$ (where $\preceq$ is the trace-containment relation) by introducing abstraction and witness modules. Suppose that the state space of the implementation $P_1||P_2$ is too large to be handled by exhaustive search algorithms. A naive compositional approach would attempt to prove both $P_1 \preceq Q$ and $P_2 \preceq Q$, and then conclude $P_1||P_2 \preceq Q$. Though sound, the naive approach often fails in practice, because P_1 usually refines Q only in a suitable constraining environment, and so does P_2. Hence we construct a suitable constraining environment A_2 for P_1, and similarly A_1 for P_2. Since A_1 describes the aspects of P_1 that are relevant to constraining P_2, and similarly A_2 is an abstract description of P_2, the two new modules A_1 and A_2 are called *abstraction modules*. By assume-guarantee reasoning, we conclude $P_1||P_2 \preceq Q$ from the two proof obligations $P_1||A_2 \preceq A_1||Q$ and $A_1||P_2 \preceq Q||A_2$.

Traditionally, the size of the implementation has been viewed as the main source of complexity for the refinement-checking problem. In our approach, we shift the focus to the size of the *gap* between the implementation and the specification. As an extreme case, if we are given two identical copies of a design, we ought to be able to verify that one is a valid refinement of the other, no matter how large the designs. We want the success rate of our methodology to increase if the designer invests effort in structuring the implementation and specification so as to expose more commonality between them. Abstraction modules form an intermediate layer between the implementation and the specification, and thus provide a systematic way of reducing the gap. In our case studies, we found that abstraction modules generally take the form of abstract definitions for hidden implementation variables. When composed with the original specification, which often specifies only relationships between primary inputs and outputs, the abstraction modules yield a richer specification that is closer to the implementation. Constructing good abstraction modules requires manual effort. Once constructed, our methodology automatically makes effective use of the abstraction modules to decompose the refinement check.

Even if the state space of the implementation becomes manageable as a result of proof decomposition, each remaining refinement check, say $P' = P_1||A_2 \preceq A_1||Q = Q'$, is still PSPACE-hard in the size of the specification state space. However, for the special case that all variables of Q' are also present in P' (in this case, we say that Q' is *projection refinable* by P'), the refinement check reduces to a transition-invariant check, which verifies that every move of P' can be mimicked by Q'. The complexity of this procedure is linear on the state spaces of both P' and Q'. If Q' is not projection refinable by P', our methodology advocates the introduction of a *witness module* W, which makes explicit how the hidden variables of the specification Q' depend on the state of the implementation P'. Then Q' is projection refinable by $P' \| W$, and it suffices to prove $P' \| W \preceq Q'$ in order to conclude $P' \preceq Q'$. The construction of witness modules also requires manual effort, but whenever the specification Q' simulates the implementation P', a suitable witness can be found.

Related work. The individual pieces of our methodology are not new; we simply advocate their disciplined use within the framework of reactive modules. In particular, assume-guarantee rules for various formalisms can be found in [Sta85,CLM89,GL94,AL95,McM97]; the rule used in this paper has been taken from [AH96]. Witnesses have appeared in various guises and forms (homomorphisms, refinement mappings, simulation relations, etc.) in different works [Lam83,LT87,AL91,BBLS92,CGL92,Kur94,LV95,McM97]. Also our choice of case studies is not new. Other correctness proofs for Tomasulo's algorithm can be found in [DP97,McM98]; the sliding-window protocol is taken from [Tan92].

2 A Verification Problem

Reactive modules. Reactive modules is a formalism for the modular description of systems with heterogeneous components. The definition of reactive modules can be found in [AH96]; here we give only a brief introduction. The state of a reactive module is determined by the values of three kinds of variables: the *external variables* are updated by the environment and can be read by the module; the *interface variables* are updated by the module and can be read by the environment; the *private variables* are updated by the module and cannot be read by the environment. The external and interface variables are called *observable*; the interface and private variables, *controlled*.

The state of a reactive module changes in a sequence of rounds. The first round is called the *initial round*, and determines initial values for all variables. Each subsequent round is called an *update round*, and determines new values for all variables. For external variables, the values in the initial and update rounds are left unspecified (i.e., chosen nondeterministically). For controlled variables, the values in the initial and update rounds are specified by (possibly nondeterministic) guarded commands. In each update round, the new value of a controlled variable may depend on the (latched) values of some variables from the previous round. In addition, in each round, the initial (or new) value of a controlled variable may depend on the initial (or new) values of some other variables from the same round; such a dependency between the values of variables within a single round is called an *await dependency*. In order to avoid inconsistent specifications, the await dependencies must be acyclic. In reactive modules, the acyclicity restriction is enforced statically, by partitioning the controlled variables into *atoms* that can be ordered such that in each round, the initial (or new) values for all variables of an atom can be determined simultaneously from the initial (or new) values of the external variables and the variables of earlier atoms.

Each round, therefore, consists of several subrounds—one for the external variables, and one per atom. Each atom has an *initial command*, which specifies the possible initial values for the variables of the atom, and an *update command*, which specifies the possible new values for the variables of the atom within each update round. In the update command, unprimed occurrences of variables refer to the latched values from the previous round; in both the initial and update commands, primed occurrences of variables refer to the initial (or new) values from the same round.

Example 1. Consider the simple instruction set architecture defined by the reactive module *ISA* of Figure 1. The module *ISA* has five external variables (inputs)—the operation

```
**module** $ISA$
  **external** $op, inp, src1, src2, dest$
  **interface** $out, stall$
  **private** $isaRegFile$
  **atom** $ISAStall$ **controls** $stall$
    **init update**
      $[]\ true \rightarrow stall' := nondet$
  **atom** $ISARegFile$ **controls** $isaRegFile$
    **init**
      $[]\ true \rightarrow$ **forall** $i$ **do** $isaRegFile'[i] := 0$
    **update**
      $[]\ \neg stall' \wedge op' = LOAD \rightarrow isaRegFile'[dest'] := inp'$
      $[]\ \neg stall' \wedge op' = AND \rightarrow isaRegFile'[dest'] := isaRegFile[src1'] \wedge isaRegFile[src2']$
      $[]\ \neg stall' \wedge op' = OR \rightarrow isaRegFile'[dest'] := isaRegFile[src1'] \vee isaRegFile[src2']$
  **atom** $ISAOut$ **controls** $out$
    **init update**
      $[]\ \neg stall' \wedge op' = STORE \rightarrow out' := isaRegFile[dest']$
```

Fig. 1. Instruction set architecture

op, the immediate operand *inp*, the source registers *src1* and *src2*, and the destination register *dest*. There are two interface variables (outputs)—the value *out* of a *STORE* instruction, and a boolean variable *stall*, which indicates if the current inputs have been accepted. If the value of *stall* is *true* in a round, then no instruction is processed in that round, and the environment is supposed to produce the same instruction again in the next round. Finally, there is one private variable—the register file *isaRegFile*.

A round of the module *ISA* consists of four subrounds. In the first subround of each update round, the environment chooses an operation, operands, and a destination, by assigning values to the external variables. In the second subround, the atom *ISAStall* decides nondeterministically if the current inputs are processed, by setting *stall* to *true* or *false*. The third subround belongs to the atom *ISARegFile*. If the updated value of *stall* is *false*, then the current instruction is processed appropriately. If the operation is *AND* or *OR*, it is performed on the source registers and the result is placed into the destination register. If the operation is *LOAD*, the immediate operand is assigned to the destination register. The fourth subround belongs to the atom *ISAOut*. If the updated value of *stall* is *false* and the current operation is *STORE*, then *out* is updated to the contents of the destination register from the previous round. Since both atoms *ISARegFile* and *ISAOut* wait, in each update round, for the new value of *stall*, they must be executed *after* the atom *ISAStall*, which produces the new value of *stall*. However, there are no await dependencies between the atoms *ISARegFile* and *ISAOut*, and therefore the third and fourth subrounds of each update round can be interchanged. □

Parallel composition. The composition operation combines two reactive modules into a single module whose behavior captures the interaction between the two component modules. Two modules P and Q are *compatible* if (1) the controlled variables of P and Q are disjoint, and (2) the await dependencies between the variables of P and Q are acyclic. If P and Q are two compatible modules, then the *composition* $P||Q$ is the module whose atoms are the (disjoint) union of the atoms from P and Q. The interface variables of $P||Q$ are the (disjoint) union of the interface variables of P and Q, and the private variables of $P||Q$ are the (disjoint) union of the private variables of P and Q. The external variables of $P||Q$ consist of the external variables of P that are not interface variables of Q, and the external variables of Q that are not interface variables of P.

Example 2. The module *ISA* from Figure 1 can be seen as the parallel composition of three modules. The module *ISAStall* has the interface variable *stall*; the module *ISARegFile* has the external variables *op*, *inp*, *src1*, *src2*, *dest*, and *stall*, and the interface variable *isaRegFile*; the module *ISAOut* has the external variables *op*, *dest*, *stall*, and *isaRegFile*, and the interface variable *out*. The operation **hide** makes the interface variable *isaRegFile* private:

$$\mathit{ISA} \; = \; \textbf{hide} \; \mathit{isaRegFile} \; \textbf{in} \; \mathit{ISAStall} || \mathit{ISARegFile} || \mathit{ISAOut} \qquad \square$$

Refinement. The notion that two reactive modules describe the same system at different levels of detail is captured by the refinement relation between modules. We define refinement as trace containment. A *state* of a module P is a valuation for the variables (external, interface, and private) of P. A state is *initial* if it can be obtained at the end of the initial round. Given two states s and t, we write $s \rightarrow_P t$ if when the state at the beginning of an update round is s, then the state at the end of the update round may be t. A *trajectory* of P is a finite sequence $s_0, \ldots, s_n$ of states such that (1) s_0 is an initial state of P, and (2) for $i \in \{0, 1, \ldots, n-1\}$, we have $s_i \rightarrow_P s_{i+1}$. The states that lie on trajectories are called *reachable*. An *observation* of P is a valuation for the observable variables (external and interface) of P. If s is a valuation to a set of variables, we use $[s]_P$ to denote the set of valuations from s restricted to the observable variables of P. For a state sequence $\overline{s} = s_0, \ldots, s_n$, we write $[\overline{s}]_P = [s_0]_P, \ldots, [s_n]_P$ for the corresponding observation sequence. If $\overline{s}$ is a trajectory of P, then the projection $[\overline{s}]_P$ is called a *trace* of P. The module Q is *refinable* by module P if (1) every interface variable of Q is an interface variable of P, and (2) every external variable of Q is an observable variable of P. The module P *refines* the module Q, written $P \preceq Q$, if (1) Q is refinable by P, and (2) for every trajectory $\overline{s}$ of P, the projection $[\overline{s}]_Q$ is a trace of Q.

Example 3. Consider the three-stage pipeline defined by the reactive module *PIPELINE* shown in Figures 2 and 3. In the first stage of the pipeline, the operands are fetched; in the second stage, the operations are performed; in the third stage, the result is written into the register file. The *PIPELINE* module is the parallel composition of seven modules. The first stage consists of the modules *Pipe1*, *Opr1*, and *Opr2*. Forwarding logic in *Opr1* and *Opr2* ensures that correct values are given to the second stage, even if the value in question has not yet been written into the register file. The second stage consists of the module *Pipe2*, which has an *ALU* atom that processes arithmetic operations using the operands from the first stage and writes the results into a write-back register called *wbReg*. The third stage is consists of the module *RegFile*, which copies *wbReg* into the appropriate register. The *PipeOut* module outputs a register value in response to a *STORE* instruction. The *Stall* module controls the *stall* signal, which is set to *true* whenever a *STORE* instruction cannot be accepted due to data dependencies.

Our goal is to show that *PIPELINE* is a correct implementation of the instruction set architecture *ISA*. This is the case if every sequence of instructions given to *PIPELINE* produces a sequence of outputs (and stalls) that is permitted by *ISA*. The module *ISA* is refinable by *PIPELINE*, so it remains to be shown that every trace of *PIPELINE* is a trace of *ISA*. □

```
module Opr1
   interface opr1
   external stall, pipe1.op, pipe2.op, pipe1.inp, wbReg, regFile, src1
   atom Opr1 controls opr1
      update
         [] $\neg stall' \rightarrow opr1' :=$
            if $src1' = pipe1.dest \wedge pipe1.op \neq NOP \wedge pipe1.op \neq STORE$
            then if $pipe1.op = LOAD$ then $pipe1.inp$ else $aluOut'$
            else if $src1' = pipe2.dest \wedge pipe2.op \neq NOP \wedge pipe2.op \neq STORE$
                 then $wbReg$ else $regFile[src1']$

module Opr2
   interface opr2
   external stall, pipe1.op, pipe2.op, pipe1.inp, wbReg, regFile, src2
   "Same as Opr1 with src1 replaced by src2"

module Pipe1
   interface pipe1.op, pipe1.inp, pipe1.dest
   external stall, inp, op, dest
   atom Pipe1 controls pipe1.op, pipe1.dest, pipe1.inp
      init
         [] $true \rightarrow pipe1.op' := NOP$
      update
         [] $true \rightarrow pipe1.op' :=$ if $stall'$ then $NOP$ else $op'$;
                  $pipe1.dest' := dest'$; $pipe1.inp' := inp'$
```

Fig. 2. Pipeline stage 1

3 Our Methodology

Witness modules. The problem of checking if $P \preceq Q$ is PSPACE-hard in the state space of Q. However, the refinement check is simpler in the special case in which all variables of Q are observable. The module Q is *projection refinable* by the module P if (1) Q is refinable by P, and (2) Q has no private variables. If Q is projection refinable by P, then every variable of Q is observable in both P and Q. Therefore, checking if $P \preceq Q$ reduces to checking if for every trajectory $\overline{s}$ of P, the projection $[\overline{s}]_Q$ is a *trajectory* of Q. According to the following proposition, this can be done by a transition-invariant check, whose complexity is linear in the state spaces of both P and Q.

Proposition 1. *[Projection refinement] Consider two modules P and Q, where Q is projection refinable by P. Then $P \preceq Q$ iff (1) if s is an initial state of P, then $[s]_Q$ is an initial state of Q, and (2) if s is a reachable state of P and $s \rightarrow_P t$, then $[s]_Q \rightarrow_Q [t]_Q$.*

We make use of this proposition as follows. Suppose that Q is refinable by P, but not projection refinable. This means that there are some private variables in Q. Define Q^u to be the module obtained by making every private variable of Q an interface variable. If we compose P with a module W whose interface variables include the private variables of Q, then Q^u is projection refinable by the composition $P||W$. Moreover, if W does not constrain any external variables of P, then $P||W \preceq Q^u$ implies $P \preceq Q$ (in fact, P is

```
module $Pipe2$
  interface $pipe2.op, pipe2.dest, wbReg, aluOut$
  external $pipe1.op, pipe1.inp, pipe1.dest, opr1, opr2$
  atom $ALU$ controls $aluOut$
    update
      [] $pipe1.op = AND \rightarrow aluOut' := opr1 \ \wedge \ opr2$
      [] $pipe1.op = OR \rightarrow aluOut' := opr1 \ \vee \ opr2$
  atom $Pipe2$ controls $pipe2.op, pipe2.dest$
    init
      [] $true \rightarrow pipe2.op' := NOP$
    update
      [] $true \rightarrow pipe2.op' := pipe1.op; \ pipe2.dest' := pipe1.dest$
  atom $WbReg$ controls $wbReg$
    update
      [] $pipe1.op = AND \ \vee \ pipe1.op = OR \rightarrow wbReg' := aluOut'$
      [] $pipe1.op = LOAD \rightarrow wbReg' := pipe1.inp$

module $RegFile$
  interface $regFile$
  external $pipe2.op, pipe2.dest, wbReg, aluOut$
  atom $RegFile$ controls $regFile$
    init
      [] $true \rightarrow$ forall $i$ do $regFile'[i] := 0$
    update
      [] $pipe2.op = AND \ \vee \ pipe2.op = OR \ \vee \ pipe2.op = LOAD \rightarrow$
         forall $i$ do $regFile'[i] :=$ if $pipe2.dest = i$ then $wbReg$ else $regFile[i]$

module $PipeOut$
  interface $out$
  external $op, regFile, dest$
  atom $Out$ controls $out$
    update
      [] $\neg stall' \ \wedge \ op' = STORE \rightarrow out' := regFile[dest']$

module $Stall$
  interface $stall$
  external $op, dest, pipe1.op, pipe1.dest, pipe2.op, pipe2.dest$
  atom $Stall$ controls $stall$
    update
      [] $op' = STORE \ \wedge \ pipe1.op \neq NOP \ \wedge \ pipe1.op \neq STORE \ \wedge \ dest' = pipe1.dest \rightarrow$
         $stall' := true$
      [] $op' = STORE \ \wedge \ pipe2.op \neq NOP \ \wedge \ pipe2.op \neq STORE \ \wedge \ dest' = pipe2.dest \rightarrow$
         $stall' := true$
      [] default $\rightarrow stall' := false$
```

Fig. 3. Pipeline stages 2 and 3, output, and stall

simulated by Q). Such a module W is called a *witness* to the refinement $P \preceq Q$. The following proposition states that in order to check refinement, it is sufficient to first find a witness module and then check projection refinement.

Proposition 2. *[Witness modules] Consider two modules P and Q such that Q is refinable by P. Let W be a module such that (1) W is compatible with P, and (2) the interface variables of W include the private variables of Q, and are disjoint from the external variables of P. Then (1) Q^u is projection refinable by $P \| W$, and (2) $P \| W \preceq Q^u$ implies $P \preceq Q$.*

Furthermore, it can be shown that if P does not have any private variables, and P is simulated by Q, then a witness to the refinement $P \preceq Q$ does exist. In summary, the creativity required from the human verification expert is the construction of a suitable witness module, which makes explicit how the private state of the specification Q depends on the state of the implementation P.

Assume-guarantee reasoning. The state space of a module may be exponential in the size of the module description. Consequently, even checking projection refinement may not be feasible. However, typically both the implementation P and the specification Q consist of the parallel composition of several modules, in which case it may be possible to reduce the problem of checking if $P \preceq Q$ to several subproblems that involve smaller state spaces. The assume-guarantee rule for reactive modules [AH96] allows us to conclude $P \preceq Q$ as long as each component of the specification Q is refined by the corresponding components of the implementation P within a suitable environment. The following proposition gives a slightly generalized account of the assume-guarantee rule.

Proposition 3. *[Assume-guarantee rule] Consider two composite modules $P = P_1 \| \cdots \| P_m$ and $Q = Q_1 \| \cdots \| Q_n$, where Q is refinable by P. For $i \in \{1, \ldots, n\}$, let Γ_i be the composition of arbitrary compatible components from P and Q with the exception of Q_i. If $\Gamma_i \preceq Q_i$ for every $i \in \{1, \ldots, n\}$, then $P \preceq Q$.*

We make use of this proposition as follows. First we decompose the specification Q into its components $Q_1 \| \cdots \| Q_n$. Then we find for each component Q_i of the specification a suitable module Γ_i (called an *obligation module*) and check that $\Gamma_i \preceq Q_i$. This is beneficial if the state space of Γ_i is smaller than the state space of P. The module Γ_i is the parallel composition of two kinds of modules—*essential modules* and *constraining modules*. The essential modules are chosen from the implementation P so that every interface variable of Q_i is an interface variable of some essential module. There may, however, be some external variables of Q_i that are not observable for the essential modules. In this case, to ensure that Q_i is refinable by Γ_i, we need to choose constraining modules from either from the implementation P or from the specification Q (other than Q_i). Once Q_i is refinable by Γ_i, if the refinement check $\Gamma_i \preceq Q_i$ goes through, then we are done. Typically, however, the external variables of Γ_i need to be constrained in order for the refinement check to go through. Until this is achieved, we must add further constraining modules to Γ_i.

It is preferable to choose constraining modules from the specification, which is less detailed than the implementation and therefore gives rise to smaller state spaces (in the undesirable limit, if we choose $\Gamma_i = P$, then the proof obligation $\Gamma_i \preceq Q_i$ involves the state space of P and is no simpler than the original proof obligation $P \preceq Q$). Unfortunately, due to lack of detail, the specification often does not supply a suitable choice of constraining modules. According to the following simple property of the refinement relation, however, we can arbitrarily "enrich" the specification by composing it with new modules.

Proposition 4. *[Abstraction modules] For all modules P, Q, and A, if $P \preceq Q \| A$ and Q is refinable by P, then $P \preceq Q$.*

So, before applying the assume-guarantee rule, we may add modules to the specification and prove $P \preceq Q \| A_1 \| \cdots \| A_k$ instead of $P \preceq Q$. The new modules $A_1, \ldots, A_k$ are called *abstraction modules*, as they usually give high-level descriptions for some implementation components, in order to provide a sufficient supply of constraining modules. In summary, the creativity required from the human verification expert is the construction of suitable abstraction modules, which on one hand, need to be as detailed as required to serve as constraining modules in assume-guarantee reasoning, and on the other hand, should be as abstract as possible to minimize their state spaces.

While witness modules are introduced "on the left" of a refinement relation, abstraction modules are introduced "on the right." So it may be necessary to iterate both processes, providing witnesses for abstractions, and abstractions for witnesses. An example of this will appear in the next section.

4 Our Solution

We prove that *PIPELINE* $\preceq$ *ISA* using Propositions 1, 2, 3, and 4. We note that *ISA* is refinable by *PIPELINE*, but not projection refinable. This is because *isaRegFile* in *ISA* is a private variable. We claim that the module *ISARegFile* is a witness module for *isaRegFile*. We then use Proposition 2 to reduce the proof obligation *PIPELINE* $\preceq$ *ISA* to *ISARegFile*$\|$*PIPELINE* $\preceq$ ISA^u. This proof obligation can be expanded in terms of component modules to

$$\begin{array}{r} \mathit{ISARegFile} \| \mathit{RegFile} \| \mathit{Opr1} \| \mathit{Opr2} \| \\ \mathit{Pipe1} \| \mathit{Pipe2} \| \mathit{PipeOut} \| \mathit{Stall} \end{array} \preceq \mathit{ISARegFile} \| \mathit{ISAOut} \| \mathit{ISAStall}.$$

Let us start by identifying *ISAOut* with Q_1. We need to find an obligation module Γ_1, such that $\Gamma_1 \preceq$ *ISAOut*. There is only one interface variable for *ISAOut*, namely *out*. The component of *PIPELINE* that generates *out* is *PipeOut*. Thus *PipeOut* is the only essential module for Γ_1. However, the proof obligation

$$\Gamma_1 = \mathit{PipeOut} \preceq Q_1 = \mathit{ISAOut}$$

fails trivially, because *ISAOut* is not refinable by *PipeOut*. The module *ISAOut* has an external variable *isaRegFile* that is not present in *PipeOut*. To achieve refinability, we add *ISARegFile*, the module controlling *isaRegFile*, to Γ_1 and try to prove

$$\Gamma_1 = \mathit{ISARegFile} \| \mathit{PipeOut} \preceq Q_1 = \mathit{ISAOut}.$$

This fails because the input *regFile* to *PipeOut* is not constrained. We add *RegFile* to constrain *regFile*, but in vain, because the check

$$\Gamma_1 = \mathit{ISARegFile} \| \mathit{RegFile} \| \mathit{PipeOut} \preceq Q_1 = \mathit{ISAOut}$$

also fails. The reason now is that the inputs to *RegFile* are not constrained. We add *Pipe2* for this purpose, and then *Pipe1*, *Opr1*, *Opr2*, and *Stall* to constrain the inputs to *Pipe2*. At last, we are able to prove the proof obligation

$$\Gamma_1 = \mathit{ISARegFile} \| \mathit{RegFile} \| \mathit{Pipe1} \| \mathit{Pipe2} \| \mathit{Opr1} \| \mathit{Opr2} \| \mathit{Stall} \| \mathit{PipeOut} \preceq Q_1 = \mathit{ISAOut}.$$

Now, according to Proposition 3, the assume-guarantee proof looks as follows:

$$\frac{\begin{array}{r@{\;}l} \mathit{ISARegFile}\|\mathit{RegFile}\|\mathit{Pipe1}\|\mathit{Pipe2}\| & \\ \mathit{Opr1}\|\mathit{Opr2}\|\mathit{Stall}\|\mathit{PipeOut} & \preceq \mathit{ISAOut} \\ \mathit{ISARegFile} & \preceq \mathit{ISARegFile} \\ \mathit{Stall} & \preceq \mathit{ISAStall} \end{array}}{\begin{array}{r@{\;}l} \mathit{ISARegFile}\|\mathit{RegFile}\|\mathit{Pipe1}\|\mathit{Pipe2}\| & \\ \mathit{Opr1}\|\mathit{Opr2}\|\mathit{Stall}\|\mathit{PipeOut} & \preceq \mathit{ISAOut}\|\mathit{ISARegFile}\|\mathit{ISAStall} \end{array}}$$

However, notice that the biggest module on the left side above the line is exactly the same as the module on the left side below the line. Hence, the compositional approach did not yield much advantage.

So let us return to the *PIPELINE* module with the intent of adding abstraction modules. We will add three abstraction modules—*AbsOpr1*, *AbsOpr2*, and *AbsRegFile*, corresponding to *Opr1*, *Opr2*, and *RegFile*. Notice that whenever the required operand specified by *src1* is currently being produced by *ALU* or is in *wbReg*, module *Opr1* looks ahead and finds it. Otherwise, it gets the operand from the register file in *PIPELINE*. It is observed that the specification variable $\mathit{isaRegFile}[\mathit{src1}']$ contains the same value that will be produced by the forwarding logic. This observation can be used to write the following abstraction module for *Opr1*.

```
module AbsOpr1
  external isaRegFile, src1, stall
  interface opr1
  atom AbsOpr1 controls opr1
    update
      [] $\neg stall' \rightarrow opr1' := isaRegFile[src1']$
```

Note that the abstraction module leaves the value of *opr1* unspecified if *stall* is *true*. The implementation module *Opr1*, on the other hand, specifies a value for *opr1* in every round. Such incomplete specification is an essential characteristic of abstraction modules. A similar abstraction module *AbsOpr2* can be written for *Opr2*.

To write an abstraction module for the implementation register file, *regFile*, observe that the value of *regFile* in every round must be equal to the value of *isaRegFile* from two rounds earlier. Thus, we can write the abstraction module for *RegFile* as $\mathit{AbsRegFile}\|\mathit{ISARegFile}_d$, where *AbsRegFile* and $\mathit{ISARegFile}_d$ are given below.

```
module $ISARegFile_d$
  atom $ISARegFile_d$ controls $isaRegFile_d$
    init
      [] $true \rightarrow$ forall $i$ do $isaRegFile'_d[i] := 0$
    update
      [] $true \rightarrow$ forall $i$ do $isaRegFile'_d[i] := isaRegFile[i]$

module AbsRegFile
  atom AbsRegFile controls regFile
    init
      [] $true \rightarrow$ forall $i$ do $regFile'[i] := 0$
    update
      [] $true \rightarrow$ forall $i$ do $regFile'[i] := isaRegFile_d[i]$
```

On composing *AbsRegFile* and $\mathit{ISARegFile}_d$ with *ISA*, we find that the new specification is not projection refinable by $\mathit{ISARegFile}\|\mathit{PIPELINE}$, because of the new specification variable $isaRegFile_d$. To regain projection refinability, a witness module needs to be written for the abstraction module $isaRegFile_d$, and composed with *PIPELINE*. A suitable witness is simply the module $ISARegFile_d$. After adding the abstraction modules, according to Proposition 3, we obtain the following assume-guarantee proof:

$$\begin{array}{rcl}
\begin{array}{r} \mathit{PipeOut} \| \mathit{Pipe1} \| \mathit{Pipe2} \| \mathit{Stall} \| \\ \mathit{AbsRegFile} \| \mathit{ISARegFile}_d \| \mathit{ISARegFile} \end{array} & \preceq & \mathit{ISAOut} \\
\begin{array}{r} \mathit{Opr1} \| \mathit{AbsOpr2} \| \mathit{Pipe1} \| \mathit{Pipe2} \| \\ \mathit{AbsRegFile} \| \mathit{ISARegFile}_d \| \mathit{ISARegFile} \end{array} & \preceq & \mathit{AbsOpr1} \\
\begin{array}{r} \mathit{Opr2} \| \mathit{AbsOpr1} \| \mathit{Pipe1} \| \mathit{Pipe2} \| \\ \mathit{AbsRegFile} \| \mathit{ISARegFile}_d \| \mathit{ISARegFile} \end{array} & \preceq & \mathit{AbsOpr2} \\
\begin{array}{r} \mathit{AbsOpr1} \| \mathit{AbsOpr2} \| \mathit{Pipe1} \| \mathit{Pipe2} \| \\ \mathit{RegFile} \| \mathit{ISARegFile}_d \| \mathit{ISARegFile} \| \mathit{Stall} \end{array} & \preceq & \mathit{AbsRegFile} \| \mathit{ISARegFile}_d \\
\mathit{Stall} & \preceq & \mathit{ISAStall} \\
\mathit{ISARegFile} & \preceq & \mathit{ISARegFile} \\
\hline
\begin{array}{r} \mathit{ISARegFile} \| \mathit{ISARegFile}_d \| \mathit{RegFile} \| \\ \mathit{Pipe1} \| \mathit{Pipe2} \| \mathit{Opr1} \| \mathit{Opr2} \| \mathit{PipeOut} \| \mathit{Stall} \end{array} & \preceq & \begin{array}{l} \mathit{ISARegFile} \| \mathit{ISAOut} \| \mathit{ISAStall} \| \mathit{AbsOpr1} \| \\ \mathit{AbsOpr2} \| \mathit{AbsRegFile} \| \mathit{ISARegFile}_d \end{array}
\end{array}$$

All proof obligations above the line satisfy projection refinability, and involve smaller state spaces than the conclusion of the proof. Following Proposition 1, they can be discharged by a transition-invariant check. Let us now focus on the modules below the line. Notice that the composite module on the left side is *PIPELINE* $\|$ *ISARegFile* $\|$ $ISARegFile_d$, and the composite module on the right side is $ISA^u \| ISARegFile_d$ $\|$ $AbsOpr1 \| AbsOpr2 \| AbsRegFile$. By Proposition 4, we can remove $ISARegFile_d \| AbsOpr1$ $\|$ $AbsOpr2 \| AbsRegFile$ from the right side to obtain the refinement

$$\mathit{PIPELINE} \| \mathit{ISARegFile} \| \mathit{ISARegFile}_d \preceq \mathit{ISA}^u.$$

The module *ISARegFile* $\|$ $ISARegFile_d$ is a witness for the refinement *PIPELINE* $\preceq$ *ISA*. Hence, by Proposition 2, we conclude that *PIPELINE* $\preceq$ *ISA*.

5 Discussion

In the previous section, we presented an assume-guarantee proof of the fact that *PIPELINE* refines *ISA*. In this section, we would like to touch upon some of the issues and finer points that came up while we were developing this methodology.

Projection refinability. Our definition of projection refinability is stronger than necessary. A variable is *history-free* if no atom uses the (latched) value of the variable from the previous round. Otherwise, the variable is said to be a *latch variable*. For module Q to be projection refinable by module P, it is sufficient to require that every latch variable of Q is observable in both P and Q.

Trivial witnesses. An atom is *deterministic* if two distinct guards of the initial command cannot be true in any given round, and the same is true for the update command. A module is *deterministic* if all its atoms are deterministic. If a private variable of the specification is controlled by a deterministic module, and all variables on which it depends are already present in the implementation, then the witness module for this variable can be easily constructed by copying the initial and update commands of the controlling module. This phenomenon can be noticed in the case study of Section 4, where we claimed *ISARegFile* as the witness for the variable *isaRegFile*. Notice also that this simplicity comes at a price. The module *ISARegFile* has latch variables, and so we have increased the number of state bits in the module over which we perform the transition-invariant check. Alternatively, a more complex witness for *isaRegFile*, which does not have any latch variables, can be produced [HQR98].

Choice of constraining modules. An important problem one faces in a compositional proof is the choice of a mimimal set of constraining modules, preferably with small state spaces. Consider one proof obligation ("lemma") $\Gamma_i \preceq Q_i$ in the compositional proof of $P \preceq Q$ using Proposition 3. Starting from the essential modules, our implementation chooses progressively larger obligation modules Γ_i in two steps. First, sufficient constraining modules are added to make Q_i refinable by Γ_i. Second, additional constraining modules are chosen according to a heuristics that looks at the data dependencies in the specification and implementation, until Γ_i refines Q_i. The constraining modules are chosen preferably from the specification Q, rather than from the implementation P. Alternatively, the user can force specific submodules of P or Q into Γ_i.

Fairness. Though not discussed here, our methodology also supports fairness conditions on the specification and implementation [HQR98].

Refinement Check	Latches
Monolithic	110*
msgP, indexP	35
msgBuffer	39
msgC, indexC	59
windowS	75
seqS	35
seqR	59
ackWait	15
seqX	55
msgX	55
ackX	51
busy	75
recvd	56
msgBufferR	68

Refinement Check	Latches
Monolithic	67*
Data Out	12
Bus valid bit	0
Bus value	32
Bus tag	0
Register[0] valid bit	4
Register[0] tag	4
Register[0] value	20
Reservation Station[0] valid bit	4
Reservation Station[0] aVal valid bit	22
Reservation Station[0] aVal tag	10
Reservation Station[0] aVal value	35

Table 1. Lemmas in the proof of sliding-window protocol (left) and Tomasulo's algorithm (right)

Other case studies. We used the methodology outlined in Section 3 to verify implementations of a sliding-window protocol and of Tomasulo's algorithm. Space does not permit us to describe these case studies in detail; a detailed description can be found in [HQR98]. The results of our experiments are summarized in Table 1. The table on the left gives the results for the sliding-window protocol with window size 12. The table on the right gives the results for Tomasulo's algorithm with 4 registers and 4 reservation stations. The tables enumerate the lemmas that were proved to conclude that the implementation refines the specification. There is a lemma for each component of the specification, and a lemma for each abstraction module that is composed with the specification. The second column gives the number of boolean latch variables that encode the state space of the corresponding obligation module. In all proofs, most obligation modules contained components from the specification or abstraction modules. These components are typically very abstract, with much nondeterminism and small state spaces. The row labeled "monolithic" refers to a noncompositional proof, where the transition-invariant check is performed on the full state space of the implementation. The superscript * indicates an unsuccessful verification attempt.

Acknowledgments. We thank Ken McMillan and Amir Pnueli for inspiring this work.

References

[AH96] R. Alur and T.A. Henzinger. Reactive modules. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science*, pages 207–218. IEEE Computer Society Press, 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. Hu and M. Vardi, editors, *CAV 98: Computer Aided Verification*, Lecture Notes in Computer Science. Springer-Verlag, 1998.

[AL91] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[AL95] M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, 1995.

[BBLS92] S. Bensalem, A. Bouajjani, C. Loiseaux, and J. Sifakis. Property-preserving simulations. In G. von Bochmann and D.K. Probst, editors, *CAV 92: Computer Aided Verification*, Lecture Notes in Computer Science 663, pages 260–273. Springer-Verlag, 1992.

[CGL92] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. In *Proceedings of the 19th Annual Symposium on Principles of Programming Languages*, pages 343–354. ACM Press, 1992.

[CLM89] E.M. Clarke, D.E. Long, and K.L. McMillan. Compositional model checking. In *Proceedings of the 4th Annual Symposium on Logic in Computer Science*, pages 353–362. IEEE Computer Society Press, 1989.

[DP97] W. Damm and A. Pnueli. Verifying out-of-order executions. In *Proceedings of the IFIP Working Conference on Correct Hardware Design and Verification Methods, CHARME*, 1997.

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

[HQR98] T.A. Henzinger, S. Qadeer, and S.K. Rajamani. You assume, we guarantee: Methodology and case studies. Technical report, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, 1998.

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

[Lam83] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, 1983.

[LT87] N.A. Lynch and M.R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the 6th Annual Symposium on Principles of Distributed Computing*, pages 137–151. ACM Press, 1987.

[LV95] N.A. Lynch and F. Vaandrager. Forward and backward simulations, Part I: Untimed systems. *Information and Computation*, 121(2):214–233, 1995.

[McM97] K.L. McMillan. A compositional rule for hardware design refinement. In O. Grumberg, editor, *CAV 97: Computer Aided Verification*, Lecture Notes in Computer Science 1254, pages 24–35. Springer-Verlag, 1997.

[McM98] K.L. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In A. Hu and M. Vardi, editors, *CAV 98: Computer Aided Verification*, Lecture Notes in Computer Science. Springer-Verlag, 1998.

[Sta85] E. W. Stark. A proof technique for rely/guarantee properties. In *Proceedings of the 5th Conference on Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Computer Science 206, pages 369–391. Springer-Verlag, 1985.

[Tan92] Andrew S. Tanenbaum. *Computer Networks.* Prentice-Hall Inc., 1992.