Decomposing Refinement Proofs using Assume-Guarantee Reasoning*

Thomas A. Henzinger†
University of California, Berkeley
tah@eecs.berkeley.edu

Shaz Qadeer
Compaq Systems Research Center
qadeer@pa.dec.com

Sriram K. Rajamani
Microsoft Research
sriram@microsoft.com

Abstract. Model-checking algorithms can be used to verify, formally and automatically, if a low-level description of a design conforms with a high-level description. However, for designs with very large state spaces, prior to the application of an algorithm, the refinement-checking task needs to be decomposed into subtasks of manageable complexity. It is natural to decompose the task following the component structure of the design. However, an individual component often does not satisfy its requirements unless the component is put into the right context, which constrains the inputs to the component. Thus, in order to verify each component individually, we need to make assumptions about its inputs, which are provided by the other components of the design. This reasoning is circular: component A is verified under the assumption that context B behaves correctly, and symmetrically, B is verified assuming the correctness of A. The assume-guarantee paradigm provides a systematic theory and methodology for ensuring the soundness of the circular style of postulating and discharging assumptions in component-based reasoning.

We give a tutorial introduction to the assume-guarantee paradigm for decomposing refinement-checking tasks. To illustrate the method, we step in detail through the formal verification of a processor pipeline against an instruction set architecture. In this example, the verification of a three-stage pipeline is broken up into three subtasks, one for each stage of the pipeline.

1 Introduction

In refinement checking we wish to verify that a low-level description of a design conforms with a high-level description. The assume-guarantee paradigm provides a method for decomposing a refinement-checking task into subtasks of manageable complexity, which can then be discharged automatically by model-checking algorithms.

We give a tutorial introduction to the assume-guarantee paradigm for decomposing refinement proofs. In Section 2, we present the assume-guarantee principle within the formalism of reactive

*A preliminary version of this paper appeared in the *Proceedings of the IEEE/ACM International Conference on Computer-aided Design* (ICCAD 00), IEEE Computer Society Press, 2000, pp. 245–252.

†Supported in part by DARPA Information Technology Office, by the MARCO Gigascale Silicon Research Center, and by the National Science Foundation.

systems. A system is a function from input signals to output signals. We call a system reactive if the input-output function is defined by induction over time using synchronous gates and latches without combinational cycles. In this model, all nondeterminism is confined to the input signals. In Section 3, we outline a systematic methodology for using the assume-guarantee principle in refinement proofs. In Section 4, we apply the methodology to verify a processor pipeline against an instruction set architecture.

We are aware of two model checkers that provide explicit tool support for refinement checking using assume-guarantee reasoning —MOCHA [AHM+98] and SMV [McM97, McM98, McM99]. Assume-guarantee refinement checking has been used successfully for verifying the correctness of algorithms, such as Tomasulo's algorithm [McM98]. It has also been used successfully for verifying real-world hardware designs against abstract specifications. Two such examples are (1) the verification of a pipelined implementation of a directory-based coherence protocol that is an integral part of the Origin 2000 servers from Silicon Graphics, using SMV [Eir98], and (2) the verification of a VGI dataflow processor array that was designed by the Infopad project at the University of California, Berkeley, using MOCHA [HLQR99].

2 The Formalism

2.1 Signals and systems

A *signal* is a function x: $T \to V$ from a time domain T to a value set V. Given a time instant $t \in T$, we write $x(t)$ for the value of the signal x at time t. The signal x is *discrete* if the time domain T is the set $\mathbb{N}$ of nonnegative integers. The signal x is *boolean* if the value set V is the set $\mathbb{B}$ of booleans.

Let P be a set of typed ports. The *type* of a port p consists of a time domain T_p and a value set V_p. A *snapshot at* P is a function s that maps each port $p \in P$ to a value $s_p \in V_p$. A *behavior at* P is a function x that maps each port $p \in P$ to a signal x_p: $T_p \to V_p$. If all ports in P agree on the time domain T, then the behavior x at P can be viewed as a function that maps each time instant $t \in T$ to a snapshot $x(t)$ at P. Given a second set Q of ports, the *projection onto* Q of the snapshot s at P, denoted $s|_Q$, is the snapshot at $P \cap Q$ which restricts the function s to the domain $P \cap Q$. Similarly, the projection $x|_Q$ of the behavior x at P is the behavior at $P \cap Q$ which restricts x to the domain $P \cap Q$. Given two disjoint sets P and Q of ports, a snapshot s at P and a snapshot u at Q, the *join* $s \bowtie u$ is the snapshot at $P \cup Q$ such that $(s \bowtie u)|_P = s$ and $(s \bowtie u)|_Q = u$. Similarly, the join $x \bowtie y$ of a behavior x at P and a behavior y at Q is the behavior at $P \cup Q$ for which $(x \bowtie y)|_P = x$ and $(x \bowtie y)|_Q = y$.

A *system* $S[P; Q]$ consists of (1) a set P of typed input ports, (2) a set Q of typed output ports disjoint from P, and (3) an input-output function S which maps each behavior x at P to a behavior y at Q. We call x an input behavior, $y = S(x)$ the corresponding output behavior, and $x \bowtie y$ the resulting input-output behavior.

If we compare two systems, then we compare the signals at the common ports. The system $S[P; Q]$ *refines* the system $S'[P'; Q']$ if for each input behavior x at P there is an input behavior x' at P' such that the input-output behaviors $x \bowtie S(x)$ and $x' \bowtie S'(x')$ agree on the common ports —that is, $(x \bowtie S(x))|_{P' \cup Q'} = (x' \bowtie S'(x'))|_{P \cup Q}$. For nontrivial value sets, this implies that the set P of input ports is disjoint from the set Q' of output ports. The refinement relation on systems is a preorder (i.e., reflexive and transitive).

2.2 Reactive systems

We restrict our attention to discrete signals. A system $S[P;Q]$ is *discrete* if the time domain of all input and output ports in $P \cup Q$ is $\mathbb{N}$. We are interested in discrete systems $S[P;Q]$ whose input-output function S is defined by induction on the time domain $\mathbb{N}$. From now on we assume that $T = \mathbb{N}$.

Consider a discrete system $S[P;Q]$ with two output ports $Q = \{1, 2\}$, and consider an input behavior x at P. The corresponding output behavior $y = S(x)$ is uniquely determined if (a) we define the initial output snapshot $y(0)$ from the inital input snapshot $x(0)$, and (b) we define, for all $t \in T$, the output snapshot $y(t+1)$ at time $t+1$ from the preceding output snapshot $y(t)$ and the concurrent input snapshot $x(t+1)$. This is called *Moore induction*. The following, more general way of defining the two interdependent output signals y_1 and y_2 is referred to as *reactive induction*. The output behavior $y = S(x)$ is uniquely determined if (a1) we define $y_1(0)$ from $x(0)$, (a2) we define $y_2(0)$ from $x(0)$ and $y_1(0)$, (b1) we define $y_1(t+1)$ from $y(t)$ and $x(t+1)$, and (b2) we define $y_2(t+1)$ from $y(t)$ and $x(t+1)$ and $y_1(t+1)$. In this case we say that the output port 2 *combinationally depends* on the output port 1. Alternatively, output port 1 could combinationally depend on output port 2. Reactive induction is a restricted form of *Mealy induction*. However, general Mealy induction, where both output ports 1 and 2 depend combinationally on each other, may not determine a unique output behavior. For example, for boolean values, the equations (b1) $y_1(t+1) = y_2(t+1)$ and (b2) $y_2(t+1) \neq y_1(t+1)$ have no solution; the equations (b1) $y_1(t+1) = y_2(t+1)$ and (b2) $y_2(t+1) = y_1(t+1)$ have multiple solutions.

Reactive induction can be generalized to more than two output ports as follows. A discrete system $S[P;Q]$ is *reactive* if there is an acyclic binary relation $\succ$ between the set Q of output ports and the set $P \cup Q$ of all ports such that for all $q \in Q$, (a) there is a function $First_q$ that, given the input values $\{x_p(0) \mid p \in P \text{ and } q \succ p\}$ and the output values $\{y_r(0) \mid r \in Q \text{ and } q \succ r\}$, returns the output value $y_q(0)$, and (b) there is a function $Next_q$ that, for all $t \in T$, given an output snapshot $y(t)$, the input values $\{x_p(t+1) \mid p \in P \text{ and } q \succ p\}$ and the output values $\{y_r(t+1) \mid r \in Q \text{ and } q \succ r\}$, returns the output value $y_q(t+1)$. The relation $\succ$ is called the *combinational dependency relation* of $S[P;Q]$. The acyclicity of $\succ$ (i.e., the transitive closure of $\succ$ is irreflexive) ensures that for each input behavior x at P, there is a unique output behavior y at Q which conforms with the functions *First* and *Next*. For each output port $q \in Q$, the function $First_q$ is called the *initialization function* of q, and $Next_q$ is the *transition function* of q. The pair $(First_q, Next_q)$ is called the *reactive definition* of q, and denoted S_q. If $S[P;Q]$ is a reactive system, then we assume that the input-output function S is given by reactive definitions for the output ports; that is, with slight abuse of notation, from now on we assume that $S = \{S_q \mid q \in Q\}$.

Consider a reactive system $S[P;Q]$. Given an output port $q \in Q$, we write $\prec q$ for the set of ports $p \in P \cup Q$ such that $q \succ p$. Given an input snapshot *in* at P, we write $First_S(in)$ for the output snapshot *out* at Q such that $out_q = First_q(in|_{\prec q}, out|_{\prec q})$ for all $q \in Q$. Given an output snapshot *prvout* at Q and an input snapshot *in* at P, we write $Next_S(prvout, in)$ for the output snapshot *out* at Q such that $out_q = Next_q(prvout, in|_{\prec q}, out|_{\prec q})$ for all $q \in Q$.

2.3 Solving witnessed refinement

Consider two reactive systems $S[P;Q]$ and $S'[P';Q']$. Then $S = \{S_q \mid q \in Q\}$ is a set of reactive definitions for the ports in Q, and $S' = \{S'_q \mid q \in Q'\}$ is a set of reactive definitions for the ports in Q'. For each port $q \in Q \cap Q'$, the two reactive definitions S_q and S'_q may be different. We want to know if $S[P;Q]$ refines $S'[P';Q']$. In this context, we call $S[P;Q]$ the *implementation* system, and $S'[P';Q']$ is the *specification* system.

Algorithm *Synchronized Search*
Input: two reactive systems $S[P;Q]$ and $S'[P';Q']$ with $P' \subseteq P \cup Q$ and $Q' \subseteq Q$.
Output: does $S[P;Q]$ refine $S'[P';Q']$?

Done := $\emptyset$;
for each snapshot *in* at P do
  out := $First_S(in)$;
  if $out|_{Q'} \neq First_{S'}((in \bowtie out)|_{P'})$ then return No fi;
  Todo := *Todo* $\cup \{out\}$
  od;
while *Todo* $\neq \emptyset$ do
  choose *prvout* from *Todo*;
  for each snapshot *in* at P do
    out := $Next_S(prvout, in)$;
    if $out|_{Q'} \neq Next_{S'}(prvout|_{Q'}, (in \bowtie out)|_{P'})$ then return No fi;
    if *out* $\notin$ *Done* then *Todo* := *Todo* $\cup \{out\}$ fi
    od;
  Done := *Done* $\cup \{prvout\}$;
  Todo := *Todo*$\backslash\{prvout\}$
  od;
return Yes.

Figure 1: Checking witnessed refinement

Of particular interest is the case in which all specification ports are implementation ports; specifically, $P' \subseteq P \cup Q$ and $Q' \subseteq Q$. If furthermore, $q \succ p$ in $S'[P';Q']$ implies $q \succ p$ in $S[P;Q]$ for all ports $q \in Q'$ and $p \in P' \cup Q'$, then we say that the specification is *fully witnessed* by the implementation, and we refer to the refinement question as fully witnessed. Fully witnessed refinement questions can be solved more efficiently than general refinement questions for reactive systems. In particular, the algorithm *Synchronized Search* of Figure 1 answers the fully witnessed refinement questions. The algorithm searches through the snapshots at Q which occur along some output behavior of $S[P;Q]$, and checks if all implementation successors of these snapshots agree with the corresponding specification successors. During the search, the variable *Done* contains the set of snapshots at Q whose successors have been explored, and the variable *Todo* contains the set of snapshots at Q whose successors need to be explored. The algorithm *Synchronized Search* can be implemented using either depth-first search (represent *Todo* as a stack) or breadth-first search (represent *Todo* as a queue). For boolean systems, binary decision diagrams may be used in breadth-first implementations.

If all ports are boolean, then the time complexity of the algorithm *Synchronized Search* is $2^{O(|Q|+|P|)}$ and the space complexity is $O(2^{|Q|} + |P|)$. Hence, the main source of complexity is the number of output ports of the implementation. We will reduce this number by decomposing the refinement check into subproblems.

2.4 Decomposing reactive systems

We define two operations on reactive systems: composition and slicing. Composition takes the union of the output ports of two systems; slicing removes some output ports from a system.

Consider two reactive systems $S_1[P_1; Q_1]$ and $S_2[P_2; Q_2]$ with the combinational dependency relations $\succ_1$ and $\succ_2$, respectively. The two systems $S_1[P_1; Q_1]$ and $S_2[P_2; Q_2]$ are *composable* if (1) the output ports are disjoint —that is, $Q1 \cap Q2 = \emptyset$— and (2) the union $\succ_1 \cup \succ_2$ of the combinational dependency relations is acyclic. If $S_1[P_1; Q_1]$ and $S_2[P_2; Q_2]$ are composable, then

$$S[P; Q] \;=\; (S_1 \cup S_2)[(P1 \backslash Q2) \cup (P2 \backslash Q1); Q1 \cup Q2]$$

is again a reactive system. We call $S[P; Q]$ the *composition* of $S_1[P_1; Q_1]$ and $S_2[P_2; Q_2]$.

If $S[P; Q]$ is a reactive system and $Q' \subseteq Q$, then

$$S[P'; Q'] \;=\; \{S_q \mid q \in Q'\}[P \cup (Q \backslash Q'); Q']$$

is again a reactive system. We call $S[P'; Q']$ the Q' *slice* of $S[P; Q]$. The Q' slice may differ from $S[P; Q]$ in the output signals at the ports in Q', because the Q' slice treats the ports in $Q \backslash Q'$, whose signals may influence the signals at Q', as unconstrained input ports. Composition and slicing are inverses: for all $Q' \subseteq Q$, the reactive system $S[P; Q]$ results from composing the Q' slice of $S[P; Q]$ with the $Q \backslash Q'$ slice of $S[P; Q]$.

2.5 Decomposing witnessed refinement

In the following, consider two reactive systems $S[P; Q]$ and $S'[P'; Q']$ with $Q' = \{q_1, \ldots, q_n\}$.

> **First weak decomposition rule.** If $S[P; Q]$ fully witnesses $S'[P'; Q']$, then in order to prove that $S[P; Q]$ refines $S'[P'; Q']$, it suffices to prove for all $1 \leq i \leq n$ that the $\{q_i\}$ slice of $S'[P'; Q']$ is refined by $S[P; Q]$. ∎

This rule does not slice the implementation, and thus gives no savings in the complexity of refinement checking.

> **Second weak decomposition rule.** If $S[P; Q]$ fully witnesses $S'[P'; Q']$, then in order to prove that $S[P; Q]$ refines $S'[P'; Q']$, it suffices to prove for all $1 \leq i \leq n$ that the $\{q_i\}$ slice of $S'[P'; Q']$ is refined by the $\{q_i\}$ slice of $S[P; Q]$. ∎

This rule slices the implementation too much: the premises usually do not hold even in cases where the conclusion is true. This is because the implementation signal at q_i may agree with the specification signal at q_i only if the implementation signals on which q_i depends are taken into account. In other words, the $\{q_i\}$ slice of the specification may be refined by the $\{q_i\}$ slice of the implementation only if its input ports $Q \backslash \{q_i\}$ are constrained. The following rule permits any set of constraints on $Q \backslash \{q_i\}$ to be taken from the implementation.

> **Third weak decomposition rule.** If $S[P; Q]$ fully witnesses $S'[P'; Q']$, then in order to prove that $S[P; Q]$ refines $S'[P'; Q']$, it suffices to prove for all $1 \leq i \leq n$ that the $\{q_i\}$ slice of $S'[P'; Q']$ is refined by some R slice of $S[P; Q]$ with $q_i \in R$. ∎

This rule generalizes the first two weak decomposition rules; it provides greater flexibity, as the slice $R \subseteq Q$ can be chosen freely. Still, a more general rule is possible: in order to show that the $\{q_i\}$ slice of the specification is refined by the $\{q_i\}$ slice of the implementation we may constrain its input ports $Q \backslash \{q_i\}$ not only with implementation signals but also with specification signals.

Strong (assume-guarantee) decomposition rule. If $S[P;Q]$ fully witnesses $S'[P';Q']$, then in order to prove that $S[P;Q]$ refines $S'[P';Q']$, it suffices to prove for all $1 \leq i \leq n$ that the $\{q_i\}$ slice of $S'[P';Q']$ is refined by the composition of (1) some R slice of $S[P;Q]$ with $q_i \in R$, and (2) some R' slice of $S'[P';Q']$ with $q_i \notin R'$. ∎

This rule offers greater flexibility than the weak decomposition rules, as both slices $R \subseteq Q$ and $R' \subseteq Q'$ can be chosen freely. Unlike the weak decomposition rules, the assume-guarantee rule appears to be circular in the sense that, say, the $\{q_1\}$ slice of the specification may be refined by an R'_1 slice of the specification with $q_2 \in R'_1$, while independently the $\{q_2\}$ slice of the specification may be refined by an R'_2 slice of the specification with $q_1 \in R'_2$. In other words, "we may assume that the specification holds at q_2 to guarantee it holds also at q_1, and independently assume that the specification holds at q_1 to guarantee it holds also at q_2." The apparent circularity is resolved by induction over time following the reactive definitions of the output signals.

Apparently circular proof rules whose soundness relies on induction over time can be traced back to [MC81]. A strong decomposition rule for asynchronous systems was given in [AL93, AL95], and for synchronous reactive systems, in [AH95, AH96]. Proof methodologies for applying strong decomposition rules were developed in [McM97] and [HQR98]. The strong decomposition rules and proof methodologies were recently generalized in many ways, for example, to accomodate multiple constraints on a single output port [McM98], branching-time refinement [HQRT98], different implementation and specification time scales [HQR99], and liveness constraints [McM99].

2.6 Witnessing refinement

The algorithm from Section 2.3 for refinement checking, as well as the decomposition rules from Section 2.5, require that the refinement question to be solved is fully witnessed. Based on the following rule, this can be achieved by adding specification ports to the implementation.

Implementation strengthening (witnessing) rule. Let $S[P;Q]$, $S'[P';Q']$, and $S^w[P^w;Q^w]$ be three reactive systems. If $Q^w \cap P = \emptyset$, then in order to prove that $S[P;Q]$ refines $S'[P';Q']$, it suffices to prove that $S'[P';Q']$ is refined by the composition of $S[P;Q]$ and $S^w[P^w;Q^w]$. ∎

If the premise of the implementation strengthening rule is witnessed and true, then we call the reactive system $S^w[P^w;Q^w]$ a *witness* to the question if $S[P;Q]$ refines $S'[P';Q']$. Note that a witness is not permitted to constrain the input ports of the implementation, but it may constrain the input ports of the specification. The construction of a witness sometimes requires creativity, but often is suggested by the refinement question at hand. In particular, for output ports $q \in Q' \backslash Q$ of the specification, it is usually appropriate to choose a witness with $q \in Q^w$ and $S^w_q = S'_q$; that is, the reactive definition of q in the witness is identical to the reactive definition of q in the specification. Input ports $p \in P' \backslash (P \cup Q)$ of the specification sometimes can be left unconstrained in the witness (then $p \in P^w$), and sometimes need to be defined in terms of the implementation ports (then $p \in Q^w$) in order for the premise of the witnessing rule to hold. It is the construction of S^w_p in the latter case which may require creative insight.

3 The Methodology

Recall the strong decomposition rule from Section 2.5 for proving that $S[P;Q]$ refines $S'[P';Q']$. The power of the rule stems from the fact that in order to refine the $\{q_i\}$ slice of the specification, we

can constrain the input ports of the $\{q_i\}$ slice of the implementation with specification signals (the slice R') —rather than implementation signals (the slice R), as in weak decomposition. Specification signals are generally preferable to implementation signals, because they tend to be more abstract and depend on fewer other signals. We therefore wish to keep the R slice as small as possible (preferably $R = \{q_i\}$), usually at the cost of enlarging R'. However, in practice, the flexibility offered by the strong decomposition rule is often irrelevant simply because many output ports of the implementation do not occur in the specification, and thus we have no abstract definitions of the signals at these ports to choose for R'. Based on the following rule, we can add to the specification abstract definitions for the signals at $Q \backslash Q'$.

> **Specification strengthening (abstraction) rule.** Let $S[P;Q]$, $S'[P';Q']$, and $S^a[P^a;Q^a]$ be three reactive systems. In order to prove that $S[P;Q]$ refines $S'[P';Q']$, it suffices to prove that $S[P;Q]$ refines the composition of $S'[P';Q']$ and $S^a[P^a;Q^a]$. ∎

In practice, the construction of suitable abstract definitions S^a_q for implementation output ports $q \in Q \backslash Q'$ is the activity which consumes the most creative energy during an assume-guarantee proof. To see this, we illustrate how a typical assume-guarantee proof may proceed. In the following, we write $S[P;Q] \Rightarrow S'[P';Q']$ to denote that $S[P;Q]$ refines $S'[P';Q']$, and we write $[P]\{S_q \mid q \in Q\}$ short for $\{S_q \mid q \in Q\}[P;Q]$ to avoid the repeated enumeration of Q.

Suppose that we want to show

(1) $[a,b]\{S_x, S_y, S_u, S_v, S_w\} \Rightarrow [a,c]\{S'_x, S'_y, S'_z\}$.

Since this refinement question is not fully witnessed, we must add the ports c and z to the implementation. Suppose we can give a reactive definition S_c of the signal at c in terms of implementation signals. Then our proof obligation becomes

(2) $[a,b]\{S_x, S_y, S_u, S_v, S_w, S_c, S'_z\} \Rightarrow [a,c]\{S'_x, S'_y, S'_z\}$.

If the algorithm *Synchronized Search* succeeds in answering this fully witnessed refinement question with 7 implementation output ports, then we are done. If not, then we can apply weak refinement decomposition and attempt to show the three proof obligations

(4) $[a,b,y,u,v,w]\{S_x, S_c, S'_z\} \Rightarrow [a,c,y,z]\{S'_x\}$,
(5) $[a,b,x,u,v,w]\{S_y, S_c, S'_z\} \Rightarrow [a,c,x,z]\{S'_y\}$,
(6) $[a,b,c,x,y,u,v,w]\{S'_z\} \Rightarrow [a,c,x,y]\{S'_z\}$.

Obligation (6) holds trivially. However, suppose that (4) and (5) do not hold, because S_x depends on y, and S_y depends on x and u. In this case, we must apply strong refinement decomposition to (2), and replace (4) and (5) by

(7) $[a,b,u,v,w]\{S_x, S'_y, S_c, S'_z\} \Rightarrow [a,c,y,z]\{S'_x\}$,
(8) $[a,b,v,w]\{S'_x, S_y, S_u, S_c, S'_z\} \Rightarrow [a,c,x,z]\{S'_y\}$.

Note that we have chosen the specification signal S'_y to constrain y, because unlike S_y, the specification signal may not depend on u, in which case (7) holds. However, suppose that (8) does not hold, because S_u in turn depends on v, and furthermore, S_v depends on w. Hence we need to replace (8) by

(9) $[a,b]\{S'_x, S_y, S_u, S_v, S_w, S_c, S'_z\} \Rightarrow [a,c,x,z]\{S'_y\}$

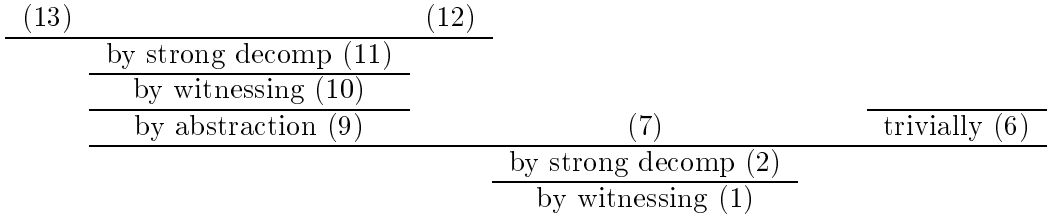

Figure 2: Sample assume-guarantee proof

and have thus arrived again at 7 implementation output ports, which represents no savings over (2). The problem is that we have no specification signal at u which is more abstract than S_u and does not depend on v. However, the abstraction rule allows us to add a reactive definition S'_u to the specification. This step usually requires a deep understanding of the implementation. Suppose that the abstract definition S'_u is nondeterministic and depends on an additional input port d. Now (9) follows from

(10) $[a, b]\{S'_x, S_y, S_u, S_v, S_w, S_c, S'_z\} \Rightarrow [a, c, d, x, z]\{S'_y, S'_u\}$.

To fully witness (10), we add a reactive definition S_d of the signal at d to the implementation. Then it suffices to show

(11) $[a, b]\{S'_x, S_y, S_u, S_v, S_w, S_c, S_d, S'_z\} \Rightarrow [a, c, d, x, z]\{S'_y, S'_u\}$,

which can be assume-guarantee decomposed to

(12) $[a, b, v, w]\{S'_x, S_y, S_c, S_d, S'_z, S'_u\} \Rightarrow [a, c, d, x, z, u]\{S'_y\}$,
(13) $[a, b, x, y]\{S_u, S_v, S_w, S_c, S_d, S'_z\} \Rightarrow [a, c, d, x, y, z]\{S'_u\}$.

Obligation (12) removes the dependencies on v and w by using the abstract definition S'_u instead of S_u. Obligation (13) verifies that S'_u is indeed an abstract definition of S_u. Both (12) and (13) have fewer implementation output ports than (9), and thus stand a better chance of being discharged automatically.

The proof is summarized in Figure 2. We have decomposed the original proof task (2) into three subtasks, namely, subtask (7) corresponding to the implementation port x, subtask (12) corresponding to the implementation port y, and subtask (13) corresponding to the implementation ports u, v, and w. While the savings in this example are minimal, the three subtasks could refer to implementation parts of arbitrary size, in which case the reduction would be substantial.

4 The Example

4.1 The system descriptions

Consider the simple instruction set architecture described by the reactive system *ISA* of Figure 4, and a simple three-stage pipeline described by the reactive system *PIPELINE* of Figure 5. We define initialization and transition functions using if-then-else syntax. In the definition of transition functions, we use the primed version p' of a port to denote the value of the signal at p at time $t+1$, and we use the unprimed version p to denote the value of the signal at p at time t. In the definition of initialization functions, we use the primed version p of a port to denote the value of the signal

type *opType*: {*AND*, *OR*, *STORE*, *NOP*}
type *regType*: **bitvector** [*WORDLENGTH*]
type *regIndexType*: (0..*NUMREGS*-1)
type *regFileType*: **array** *regIndexType* **of** *regType*
type *pipe1Type*: **record of**
    op: *opType*;
    dest: *regIndexType*;
    inp: *regType*;
    opr1: *regType*;
    opr2: *regType*
type *pipe2Type*: **record of**
    op: *opType*;
    dest: *regIndexType*;
    res: *regType*

Figure 3: Data types used in *ISA* and *PIPELINE*

at p at time 0. It follows that the left-hand-sides of all definitions are primed versions of output ports; the right-hand-sides of all definitions may contain unprimed and primed versions of output ports, and primed versions of input ports. The types of the input and output ports used in this example are defined in Figure 3. We use arrays for modeling register files, and records for modeling pipeline stages.

The reactive system *ISA* has six input ports —the operation *op*, the immediate operand *inp*, the source registers *src1* and *src2*, the destination register *dest*, and the signal *stall*, which indicates if the current inputs should be processed. If the value of *stall* is *true*, then no instruction is processed, and the environment is expected to produce the same instruction again at the next time instant. There are two output ports —the value *out* of a *STORE* instruction, and the register file *isaRegFile*.

The reactive system *PIPELINE* is a piplelined implementation of the described instruction set architecture. 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. All input ports of the *ISA* system with the exception of *stall* are input ports of the *PIPELINE* system as well, and *stall* is an output port of *PIPELINE*. The *PIPELINE* system has five other output ports —the register file *regFile*, the result *pipe1* of the first stage of the pipeline, the ALU output *aluOut*, the result *pipe2* of the second stage of the pipeline, and *out*. The output *pipe1* is a record $\{op, dest, inp, opr1, opr2\}$. In the first stage of the pipeline, the *op*, *dest*, and *inp* fields of *pipe1* store the corresponding inputs; the *opr1* and *opr2* fields store the generated operands. Forwarding logic ensures that correct operand values are generated even if the values have not yet been written into the register file. The output *aluOut* keeps the result of ALU operation in the second stage. The output *pipe2* is a record $\{op, dest, res\}$. The *op* and *dest* fields are copied from *pipe1*, and the field *res* stores the value to be written back into the register *pipe2.dest*. The third stage copies *pipe2.res* into the register *pipe2.dest*. The signal *out* outputs a register value in response to a *STORE* instruction. The signal *stall* is *true* whenever a *STORE* instruction cannot be accepted due to data dependencies.

We use the keyword **nondet** as an abbreviation denoting an unnamed input port of the appropriate type (depending on the context). For example, the definition of the transition function for

reactive system *ISA*
input *op*: *opType*; *inp*: *regType*; *stall*: **bool**; *src1*,*src2*,*dest*: *regIndexType*
output *out*: *regType*; *isaRegFile*: *regFileType*
initialization
  forall *i* **do** $isaRegFile'[i] := 0$
transition
  $isaRegFile'[dest'] :=$
    if $stall'$ **then** $isaRegFile[dest']$
    elsif $op' = LOAD$ **then** inp'
    elsif $op' = AND$ **then** $isaRegFile[src1'] \ \wedge \ isaRegFile[src2']$
    elsif $op' = OR$ **then** $isaRegFile[src1'] \ \vee \ isaRegFile[src2']$
    else $isaRegFile[dest']$
  $out' :=$ **if** $\neg stall' \ \wedge \ op' = STORE$ **then** $isaRegFile[dest']$ **else nondet**

Figure 4: Instruction set architecture

the signal *out* of *ISA* makes use of the keyword **nondet**. This is equivalent to having an input port *outchoice* of type *regType* and stating the transition function as

$$out' := \textbf{if} \ \neg stall' \ \wedge \ op' = STORE \ \textbf{then} \ isaRegFile[dest'] \ \textbf{else} \ outchoice'.$$

Since there is no other purpose for the port *outchoice*, we prefer to use the syntactic sugar **nondet** in such situations. If **nondet** is used in the specification, then the corresponding input port has to be witnessed in the implementation. This can be achieved simply by

$$outchoice' := out'.$$

When we make multiple uses of **nondet**, then each use refers to a different unnamed input port of the appropriate type.

4.2 The refinement proof

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*. Formally, we would like to prove that

$$PIPELINE \Rightarrow ISA.$$

The first step is to satisfy the requirement that the input and output ports of *ISA* are also present in *PIPELINE*. Except for *isaRegFile* and the unnamed input ports that correspond to nondeterministic choices (which are witnessed as described above), this already holds. We simply add the port *isaRegFile* and its reactive definition from *ISA* to *PIPELINE*, thus obtaining the system $PIPELINE^w$ and the fully witnessed refinement question

$$PIPELINE^w \Rightarrow ISA.$$

Note that *stall*, which is an input port of *ISA*, is witnessed by an output port of $PIPELINE^w$. At this point we could make use of the algorithm from Figure 1 to check the desired refinement.

reactive system *PIPELINE*
input *op*: *opType*; *inp*: *regType*; *src1*,*src2*,*dest*: *regIndexType*
output *out*,*aluOut*: *regType*; *pipe1*: *pipe1Type*; *pipe2*: *pipe2Type*; *regFile*: *regFileType*; *stall*: **bool**
initialization
  $pipe1.op' := NOP$
  $pipe2.op' := NOP$
  forall i **do** $regFile'[i] := 0$
transition
  $pipe1.opr1' :=$
    if $stall'$ **then** $pipe1.opr1$
    elsif $src1' = pipe1.dest \ \wedge \ pipe1.op \neq NOP \ \wedge \ pipe1.op \neq STORE$ **then**
      (**if** $pipe1.op = LOAD$ **then** $pipe1.inp$ **else** $aluOut'$)
    elsif $src1' = pipe2.dest \ \wedge \ pipe2.op \neq NOP \ \wedge \ pipe2.op \neq STORE$ **then** $pipe2.res$
    else $regFile[src1']$
  $pipe1.opr2' :=$
    if $stall'$ **then** $pipe1.opr2$
    elsif $src2' = pipe1.dest \ \wedge \ pipe1.op \neq NOP \ \wedge \ pipe1.op \neq STORE$ **then**
      (**if** $pipe1.op = LOAD$ **then** $pipe1.inp$ **else** $aluOut'$)
    elsif $src2' = pipe2.dest \ \wedge \ pipe2.op \neq NOP \ \wedge \ pipe2.op \neq STORE$ **then** $pipe2.res$
    else $regFile[src2']$
  $pipe1.op' :=$ **if** $stall'$ **then** NOP **else** op'
  $pipe1.dest' := dest'$
  $pipe1.inp' := inp'$
  $aluOut' :=$
    if $pipe1.op = AND$ **then** $opr1 \ \wedge \ opr2$
    elsif $pipe1.op = OR$ **then** $opr1 \ \vee \ opr2$
    else nondet
  $pipe2.op' := pipe1.op$
  $pipe2.dest' := pipe1.dest$
  $pipe2.res' :=$
    if $pipe1.op = AND \ \vee \ pipe1.op = OR$ **then** $aluOut'$
    elsif $pipe1.op = LOAD$ **then** $pipe1.inp$
    else $pipe2.res$
  $regFile'[pipe2.dest] :=$
    if $pipe2.op = AND \ \vee \ pipe2.op = OR \ \vee \ pipe2.op = LOAD$ **then** $pipe2.res$
    else $regFile[pipe2.dest]$
  $out' := regFile[dest']$
  $stall' :=$
    $(op' = STORE \ \wedge \ pipe1.op \neq NOP \ \wedge \ pipe1.op \neq STORE \ \wedge \ dest' = pipe1.dest) \ \vee$
    $(op' = STORE \ \wedge \ pipe2.op \neq NOP \ \wedge \ pipe2.op \neq STORE \ \wedge \ dest' = pipe2.dest)$

Figure 5: Three-stage pipeline

transition
$pipe1.opr1' :=$ **if** $\neg stall'$ **then** $isaRegFile[src1']$ **else nondet**
$pipe1.opr2' :=$ **if** $\neg stall'$ **then** $isaRegFile[src2']$ **else nondet**
$pipe2.res' :=$ **if** $pipe2.op' \in \{AND, OR, LOAD\}$ **then** $isaRegFile[pipe2.dest']$ **else nondet**

Figure 6: Abstract definitions of pipeline signals

However, we are interested in avoiding state explosion by decomposing the refinement question into questions of less complexity using the assume-guarantee rule.

In the following, for a set Q of ports, we denote the Q slice of a reactive system S by $S[Q]$, and we write $\cup$ for the composition of reactive systems. Since *out* is the output port in common between $PIPELINE^w$ and *ISA*, we attempt to prove the correctness of the slice $ISA[out]$. We use the second weak decomposition rule and attempt to show that

$$PIPELINE^w[out] \Rightarrow ISA[out].$$

However, the proof fails because *out* depends on *regFile* in $PIPELINE^w$ and on *isaRegFile* in *ISA*. Both *regFile* and *isaRegFile* are inputs to the slice $PIPELINE^w[out]$, and therefore vary independently in a nondeterministic way. We add the definition of *regFile* and *isaRegFile* to the left-hand-side using the third weak decomposition rule, but in vain, because the check

$$PIPELINE^w[out, regFile, isaRegFile] \Rightarrow ISA[out]$$

also fails. The reason now is that *regFile* depends on the variables of the second pipeline stage, which are not constrained. We add the definition of *pipe2* for this purpose, but *pipe2* depends on *pipe1* and *aluOut*, and *pipe1* in turn depends on *stall*. Therefore, we end up having to add the whole implementation and prove that

$$PIPELINE^w[out, regFile, isaRegFile, pipe1, pipe2, aluOut, stall] \Rightarrow ISA[out].$$

Hence, this approach still explores the entire state space of the $PIPELINE^w$ system and does not yield any advantage.

So let us return to the *ISA* system with the intent of constructing abstract definitions of some of its outputs, and using these to break up the proof according to the strong decomposition rule. We strengthen the system *ISA* using the abstraction rule, and add abstract definitions for three outputs of *PIPELINE* —*pipe1.opr1*, *pipe1.opr2*, and *pipe2.res*. The transition functions for these definitions are shown in Figure 6. The corresponding initialization functions are nondeterministic; they do not constrain the defined signals. Let ISA^a be the resulting specification, and as usual, add witnesses for the new unnamed input ports to the implementation. The intuition behind the abstract definitions is as follows. Both *pipe1* and *pipe2* contain control fields that include relevant parts of the instruction to be processed, and data fields that contain data on which the instruction operates. We write reactive definitions for the data fields in terms of the control fields and the instruction register file *isaRegFile*. For example, if the instruction in the third stage of the pipeline is going to update the register file, then the update must already have happened in the ISA when this instruction was in the first stage of the pipeline. Therefore, the value to be written back into *pipe2.res* can be obtained directly from *isaRegFile*. Similar intuitions guide the abstract definitions of *pipe1.opr1* and *pipe1.opr2*. Note the incomplete nature of the abstract definitions.

1	$ISA^a[pipe2.res] \cup$ $PIPELINE^w[out, stall, regFile, isaRegFile, pipe1.op,$ $pipe1.dest, pipe1.inp, pipe2.op, pipe2.dest]$	$\Rightarrow$	$ISA^a[out]$	(by model checking)
2	$ISA^a[pipe1.opr1, pipe1.opr2] \cup$ $PIPELINE^w[stall, pipe1.op, pipe1.dest, pipe1.inp,$ $pipe2.op, pipe2.dest, pipe2.res, aluOut, isaRegFile]$	$\Rightarrow$	$ISA^a[pipe2.res]$	(by model checking)
3	$ISA^a[pipe1.opr2, pipe2.res] \cup$ $PIPELINE^w[stall, regFile, pipe1.op, pipe1.dest, isaRegFile,$ $pipe1.inp, pipe1.opr1, pipe2.op, pipe2.dest, aluOut]$	$\Rightarrow$	$ISA^a[pipe1.opr1]$	(by model checking)
4	$ISA^a[pipe1.opr1, pipe2.res] \cup$ $PIPELINE^w[stall, regFile, pipe1.op, pipe1.dest, isaRegFile,$ $pipe1.inp, pipe1.opr2, pipe2.op, pipe2.dest, aluOut]$	$\Rightarrow$	$ISA^a[pipe1.opr2]$	(by model checking)
	$PIPELINE^w$	$\Rightarrow$	ISA^a	(by strong decomposition)
	$PIPELINE^w$	$\Rightarrow$	ISA	(by abstraction)
	$PIPELINE$	$\Rightarrow$	ISA	(by witnessing)

Figure 7: Assume-guarantee proof that *PIPELINE* refines *ISA*

For example, the abstract definition for *pipe1.opr1* leaves the signal unspecified when *stall* is *true*. The implementation of the signal, on the other hand, specifies a value at all times.

The abstract definitions described above allow us to decompose the proof of the whole pipeline, using the assume-guarantee rule, into the three stages. The first proof obligation of Figure 7 shows the proof of *out*, in which the abstract definition of *pipe2.res* is used to constrain the input of the implementation register file. By using the abstract definition of *pipe2.res* we avoid including *aluOut*, *pipe1.opr1*, and *pipe1.opr2* in the proof of *out*. The implementation of *pipe2.res* depends on the signals *pipe1.opr1* and *pipe1.opr2*. The second proof obligation of Figure 7 shows how the abstract defintions of *pipe1.opr1* and *pipe1.opr2* are used to prove the abstract definition of *pipe2.res*. The abstract definition of *pipe1.opr1* is proved by using the abstract definitions of *pipe1.opr2* and *pipe2.res*. The proof of *pipe1.opr2* is symmetric. This is shown in the third and fourth proof obligations of Figure 7. Note the circularity in the last three proof obligations, where each signal on the right is used on the left to prove the other two signals.

All four proof obligations involve only portions of the $PIPELINE^w$ system. Thus we avoid exploring the state space of the whole pipeline implementation. Creative energy was required to come up with the abstract definitions in Figure 6. Once these definitions are written, the mechanics of decomposing the proof using the assume-guarantee rule can be automated. In particular, given the abstract definitions, the tool MOCHA [AHM+98] is able to automatically produce and prove the four proof obligations of Figure 7. For each subproof MOCHA chooses the slices on the left-hand-side using heuristics. MOCHA also provides facilities for manually overriding the automatic choices. However, no manual overrides are necessary in this example.

References

[AH95] R. Alur and T.A. Henzinger. Local liveness for compositional modeling of fair reactive systems. In P. Wolper, editor, *CAV 95: Computer-aided Verification*, Lecture Notes in Computer Science 939, pages 166–179. Springer-Verlag, 1995.

[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, pages 521–525. Springer-Verlag, 1998.

[AL93] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, 1993.

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

[Eir98] A.T. Eiriksson. The formal design of 1M-gate ASICs. In G. Gopalakrishnan and P. Windley, editors, *FMCAD 98: Formal Methods in Computer-aided Design*, Lecture Notes in Computer Science 1522, pages 49–63. Springer-Verlag, 1998.

[HLQR99] T.A. Henzinger, X. Liu, S. Qadeer, and S.K. Rajamani. Formal specification and verification of a dataflow processor array. In *Proceedings of the International Conference on Computer-aided Design*, pages 494–499. IEEE Computer Society Press, 1999.

[HQR98] T.A. Henzinger, S. Qadeer, and S.K. Rajamani. You assume, we guarantee: methodology and case studies. In A. Hu and M. Vardi, editors, *CAV 98: Computer-aided Verification*, Lecture Notes in Computer Science, pages 440–451. Springer-Verlag, 1998.

[HQR99] T.A. Henzinger, S. Qadeer, and S.K. Rajamani. Assume-guarantee refinement between different time scales. In N. Halbwachs and D. Peled, editors, *CAV 99: Computer-aided Verification*, Lecture Notes in Computer Science 1633, pages 208–221. Springer-Verlag, 1999.

[HQRT98] T.A. Henzinger, S. Qadeer, S.K. Rajamani, and S. Tasiran. An assume-guarantee rule for checking simulation. In G. Gopalakrishnan and P. Windley, editors, *FMCAD 98: Formal Methods in Computer-aided Design*, Lecture Notes in Computer Science 1522, pages 421–432. Springer-Verlag, 1998.

[MC81] J. Misra and K.M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, SE-7(4):417–426, 1981.

[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 1427. pages 110–121. Springer-Verlag, 1998.

[McM99] K.L. McMillan. Circular compositional reasoning about liveness. In L. Pierre and T. Kropf, editors, *CHARME 99: Correct Hardware Design and Verification*, Lecture Notes in Computer Science 1703, pages 342–345. Springer-Verlag, 1999.