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 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 InQpad project at the University of California, Berkeley, using Mocha [HLQR99].
the join \(x \triangleright y\) of a behavior \(x\) at \(P\) and a behavior \(y\) at \(Q\) is the behavior at \(P \cup Q\) for which \((x \triangleright y)|_P = x\) and \((x \triangleright y)|_Q = y\).

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 \(x \triangleright y\) 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 \(x \triangleright S(x)\) and \(x' \triangleright S'(x')\) agree on the common ports—that is, \((x \triangleright S(x))|_{P' \cup Q'} = (x' \triangleright S'(x'))|_{P' \cup Q'}\). For nontrivial value sets, this implies that the set 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 initial 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 \textit{Moore induction}. The following, more general way of defining the two interdependent output signals \(y_1\) and \(y_2\) is referred to as \textit{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 \textit{combinatorially depends} on the output port 1. Alternatively, output port 1 could combinationally depend on output port 2. Reactive induction is a restricted form of \textit{Mealy induction}. However, general Mealy induction, where both output ports 1 and 2 depend combinatorially 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 \(\triangleright\) 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 \(\text{First}_q\) that, given the input values \(\{x_p(0) \mid p \in P\) and \(q \triangleright p\}\) and the output values \(\{y_r(0) \mid r \in Q\) and \(q \triangleright r\}\), returns the output value \(y_r(0)\), and (b) there is a function \(\text{Next}_q\) that, for all \(t \in T\), given an output snapshot \(y(t)\), the input values \(\{x_p(t+1) \mid p \in P\) and \(q \triangleright p\}\) and the output values \(\{y_r(t+1) \mid r \in Q\) and \(q \triangleright r\}\), returns the output value \(y_r(t+1)\). The relation \(\triangleright\) is called the combinational dependency relation of \(S[P; Q]\). The acyclicity of \(\triangleright\) (i.e., the transitive closure of \(\triangleright\) 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 \(\text{First}\) and \(\text{Next}\). For each output port \(q \in Q\), the function \(\text{First}_q\) is called the initialisation function of \(q\), and \(\text{Next}_q\) is the transition function of \(q\). The pair \((\text{First}_q, \text{Next}_q)\) 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 \(q \triangleright p\). Given an input snapshot \(\text{inp}\) at \(P\), we write \(\text{First}_q(\text{inp})\) for the output snapshot \(\text{out}\) at \(Q\) such that \(\text{out}_q = \text{First}_q(\text{inp}\mid_{\prec q}, \text{out}\mid_{\prec q})\) for all \(q \in Q\). Given an output snapshot \(\text{prevout}\) at \(Q\) and an input snapshot \(\text{inp}\) at \(P\), we write \(\text{Next}_q(\text{prevout}, \text{inp})\) for the output snapshot \(\text{out}\) at \(Q\) such that \(\text{out}_q = \text{Next}_q(\text{prevout}, \text{inp}\mid_{\prec q}, \text{out}\mid_{\prec q})\) 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 \textit{implementation system}, and \(S'[P'; Q']\) the \textit{specification system}.

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, \(q \triangleright p\) in \(S'[P'; Q']\) implies \(q \triangleright p\) in \(S[P; Q]\) for all ports \(q \in Q'\) and \(p \in P' \cup Q'\), then we say that the specification is \textit{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 \textit{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 \textit{Done} contains the set of snapshots at \(Q\) whose successors have been explored, and the variable \textit{Todo} contains the set of snapshots at \(Q\) whose successors need to be explored. The algorithm \textit{Synchronized Search} can be implemented using either depth-first search (represent \textit{Todo} as a stack) or breadth-first search (represent \textit{Todo} as a queue). For boolean systems, binary decision diagrams may be used.
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
  $out := FirstS(in)$;
  if $out|Q \neq FirstS((in \uplus out)|P')$ then return No fi;
  $Todo := Todo \cup \{out\}$ od;
while $Todo \neq \emptyset$ do
  choose $prwout$ from $Todo$;
  for each snapshot $in$ at $P$ do
    $out := NextS(prwout, in)$;
    if $out|Q \neq NextS((prwout|Q', (in \uplus out)|P'))$ then return No fi;
  od;
if $out \notin Done$ then $Todo := Todo \cup \{out\}$ fi;
$Done := Done \cup \{prwout\}$;
$Todo := Todo \setminus prwout$ od;
return Yes.

Figure 1: Checking witnessed refinement

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, $Q_1 \cap Q_2 = \emptyset$ — 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

$$S[P; Q] = (S_1 \cup S_2)(P_1 \cup Q_2) \cup (P_2 \cup Q_1) \cup Q_1 \cup Q_2$$

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

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

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 $Q \setminus Q'$, whose signals may influence the signals at $Q'$, as uncontrolled 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 $Q \setminus Q'$ 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.

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 flexibility, 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 $Q \setminus \{q_i\}$ 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 \( q_i \in R^R \). In other words, “we may assume that the specification holds at \( q_i \) to guarantee it holds also at \( q_i \), and independently assume that the specification holds at \( q_i \) to guarantee it holds also at \( q_i \).” 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, AL93], 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 accommodate multiple constraints on a single output port [McM98], branching-time refinement [HQRRT98], 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 \( S''[P''; Q''] \) be three reactive systems. If \( Q'' \cap P = \emptyset \), 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 \( S''[P''; Q''] \).

If the premise of the implementation strengthening rule is witnessed and true, then we call the reactive system \( S''[P''; Q''] \) 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 \( q \in Q \setminus Q' \) of the specification, it is usually appropriate to choose a witness with \( q \in Q'' \) and \( S'' = S'_q \), that is, the reactive definition of \( q \) in the witness is identical to the reactive definition of \( q \) in the specification. Input ports \( p \in P \setminus (P \cup Q) \) of the specification sometimes can be left unconstrained in the witness (then \( p \in P'' \)), and sometimes need to be defined in terms of the implementation ports (then \( p \in Q'' \)) in order for the premise of the witnessing rule to hold. It is the construction of \( S'' \) 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 \( R^R \))—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 \( R^R \). 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 \( R^R \). Based on the following rule, we can add to the specification abstract definitions for the signals at \( Q \setminus Q' \).

**Specification strengthening (abstraction) rule.** Let \( S[P; Q] \), \( S'[P'; Q'] \), and \( S''[P''; Q''] \) 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 \( S''[P''; Q''] \). In practice, the construction of suitable abstract definitions \( S'' \) for implementation output ports \( q \in Q \setminus Q' \) 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 \( S[P; Q][q \in Q] \) short for \( S[q \in Q][P; Q] \) to avoid the repeated enumeration of \( Q \).

Suppose that we want to show

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

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 \( S'_z \) of the signal at \( c \) in terms of implementation signals. Then our proof obligation becomes

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

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

\[
(4) \quad [a, b, y, u, v, w]\{S_x, S_c, S'_z\} \Rightarrow [a, c, y, z]\{S'_x\},
\]

\[
(5) \quad [a, b, x, u, v, w]\{S_y, S_c, S'_z\} \Rightarrow [a, c, x, z]\{S'_y\},
\]

\[
(6) \quad [a, b, c, x, y, u, v, w]\{S'_z\} \Rightarrow [a, c, x, y]\{S'_z\}.
\]

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

\[
(7) \quad [a, b, u, v, w]\{S_x, S'_y, S_c, S'_z\} \Rightarrow [a, c, y, z]\{S'_x\},
\]

\[
(8) \quad [a, b, v, w]\{S'_x, S_y, S_c, S'_z\} \Rightarrow [a, c, x, z]\{S'_y\}.
\]

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 \( S_x \) in turn depends on \( v \), and furthermore, \( S_v \) depends on \( w \). Hence we need to replace (8) by
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

\[
\{a, b\} \{S'_x, S_y, S_u, S_v, S_w, S_e, S'_z\} \Rightarrow \{a, c, x, z\} \{S'_y, S'_e\}
\]

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

\[
\{a, b\} \{S'_x, S_y, S_u, S_v, S_w, S_e, S_d, S'_z\} \Rightarrow \{a, c, d, x, z\} \{S'_y, S'_e\}
\]

which can be assume-guarantee decomposed to

\[
\{a, b, v, w\} \{S'_x, S_y, S_u, S_v, S_w, S_e, S_d, S'_z\} \Rightarrow \{a, c, d, x, z\} \{S'_y, S'_e\}
\]

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, sub-task (7) corresponding to the implementation port \( x \), sub-task (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 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 pipelined 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 \( pipe2 \), etc.
of the second stage of the pipeline, and \( \text{out} \). The output \( \text{pipe1} \) is a record \{\( \text{op}, \text{dest}, \text{inp}, \text{opr1}, \text{opr2} \}\). In the first stage of the pipeline, the \( \text{op} \), \( \text{dest} \), and \( \text{inp} \) fields of \( \text{pipe1} \) store the corresponding inputs; the \( \text{opr1} \) and \( \text{opr2} \) 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 \( \text{aluOut} \) keeps the result of ALU operation in the second stage. The output \( \text{pipe2} \) is a record \{\( \text{op}, \text{dest}, \text{res} \}\}. The \( \text{op} \) and \( \text{dest} \) fields are copied from \( \text{pipe1} \), and the field \( \text{res} \) stores the value to be written back into the register \( \text{pipe2}.\text{dest} \). The third stage copies \( \text{pipe2}.\text{res} \) into the register \( \text{pipe2}.\text{dest} \). The signal \( \text{out} \) outputs a register value in response to a \( \text{STORE} \) instruction. The signal \( \text{stall} \) is true whenever a \( \text{STORE} \) instruction cannot be accepted due to data dependencies.

We use the keyword \text{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 the signal \( \text{out} \) of \( \text{ISA} \) makes use of the keyword \text{nondet}. This is equivalent to having an input port \text{outchoice} type \text{regType} and stating the transition function as

\[
\text{out' := if } \neg \text{stall'} \land \text{op' = STORE then } \text{isaRegFile[dest']}
\]

\[
\text{else outchoice'}. \\
\]

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

\[
\text{outchoice' := out'}. \\
\]

When we make multiple uses of \text{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 \( \text{PIPELINE} \) is a correct implementation of the instruction set architecture \( \text{ISA} \). This is the case if every sequence of instructions given to \( \text{PIPELINE} \) produces a sequence of outputs (and stalls) that is permitted by \( \text{ISA} \). Formally, we would like to prove that

\[
\text{PIPELINE} \Rightarrow \text{ISA}. \\
\]

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

\[
\text{PIPELINE}^{w} \Rightarrow \text{ISA}. \\
\]

Note that \( \text{stall} \), which is an input port of \( \text{ISA} \), is witnessed by an output port of \( \text{PIPELINE}^{w} \). At this point we could make use of the algorithm from Figure 1 to check the desired refinement. 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 \( U \) for the composition of reactive systems. Since \( \text{out} \) is the output port in common between \( \text{PIPELINE}^{w} \) and \( \text{ISA} \), we attempt to prove the correctness of the slice \( \text{ISA}[\text{out}] \). We use the second weak decomposition rule and attempt to show that

\[
\text{PIPELINE}^{w}[\text{out}, \text{regFile}, \text{isaRegFile}] \Rightarrow \text{ISA}[\text{out}]. \\
\]

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

\[
\text{PIPELINE}^{w}[\text{out}, \text{regFile}, \text{isaRegFile}] \Rightarrow \text{ISA}[\text{out}] \\
\]

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

\[
\text{PIPELINE}^{w}[\text{out}, \text{regFile}, \text{isaRegFile}, \text{pipe1}, \text{pipe2}, \text{aluOut}, \text{stall}] \\
\]

\[
\Rightarrow \text{ISA}[\text{out}]. \\
\]

Hence, this approach still explores the entire state space of the \( \text{PIPELINE}^{w} \) system and does not yield any advantage.

So let us return to the \( \text{ISA} \) 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 \( \text{ISA} \) using the abstraction rule, and add abstract definitions for three outputs of \( \text{PIPELINE} = \text{pipe1}.\text{opr1}, \text{pipe1}.\text{opr2}, \text{and } \text{pipe2}.\text{res} \). The transition functions for these definitions are shown in
The second proof obligation of Figure 7 shows how 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 \texttt{aluOut}, \texttt{pipe1.opr1}, and \texttt{pipe1.opr2} in the proof of \texttt{out}. The implementation of pipe2.res depends on the signals \texttt{pipe1.opr1} and \texttt{pipe1.opr2}. The second proof obligation of Figure 7 shows how the abstract definitions of \texttt{pipe1.opr1} and \texttt{pipe1.opr2} are used to prove the abstract definition of pipe2.res. The abstract definition of \texttt{pipe1.opr1} is proved by using the abstract definitions of \texttt{pipe1.opr2} and pipe2.res. The proof of \texttt{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 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.

\begin{proof}

\begin{figure}[h]
\begin{center}
\begin{verbatim}
ISA^{[pipe2.res]} \cup
1 PIPELINE^{[out, stall, regFile, isaRegFile, pipe1.opr, pipe1.dest, pipe2.op, pipe2.dest]} \Rightarrow ISA^{[out]}
   \hspace{1cm} \text{(by model checking)}

ISA^{[pipe1.opr1, pipe1.opr2]} \cup
2 PIPELINE^{[stall, pipe1.opr, pipe1.dest, pipe1.inp, pipe2.op, pipe2.dest, pipe2.res, aluOut, isaRegFile]} \Rightarrow ISA^{[pipe2.res]}
   \hspace{1cm} \text{(by model checking)}

ISA^{[pipe1.opr2, pipe2.res]} \cup
3 PIPELINE^{[stall, regFile, pipe1.opr, pipe1.dest, isaRegFile, pipe1.inp, pipe1.opr1, pipe2.op, pipe2.dest, aluOut]} \Rightarrow ISA^{[pipe1.opr1]}
   \hspace{1cm} \text{(by model checking)}

ISA^{[pipe1.opr1, pipe2.res]} \cup
4 PIPELINE^{[stall, regFile, pipe1.opr, pipe1.dest, isaRegFile, pipe1.inp, pipe1.opr2, pipe2.op, pipe2.dest, aluOut]} \Rightarrow ISA^{[pipe1.opr2]}
   \hspace{1cm} \text{(by model checking)}
\end{verbatim}
\end{center}
\end{figure}

Figure 7: Assume-guarantee proof that PIPELINE refines ISA
\end{proof}

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 \texttt{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 \texttt{aluOut}, \texttt{pipe1.opr1}, and \texttt{pipe1.opr2} in the proof of \texttt{out}. The implementation of pipe2.res depends on the signals \texttt{pipe1.opr1} and \texttt{pipe1.opr2}. The second proof obligation of Figure 7 shows how the abstract definitions of \texttt{pipe1.opr1} and \texttt{pipe1.opr2} are used to prove the abstract definition of pipe2.res. The abstract definitin of \texttt{pipe1.opr1} is proved by using the abstract definitions of \texttt{pipe1.opr2} and pipe2.res. The proof of \texttt{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 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.

\begin{thebibliography}{10}


\end{thebibliography}
Figure 6: Abstract definitions of pipeline signals

\[\text{transition} \]
\[
\text{pipe}_{1}. \text{opr}_{1} := \begin{cases} 
\text{if } \text{stall} \text{ then } & \text{pipe}_{1}. \text{opr}_{1} \\
\text{else if } & \text{src} \neq \text{NOP} \land \text{pipe}_{1}. \text{opr} \neq \text{NOP} \land \text{pipe}_{1}. \text{res} \neq \text{STOR} \text{ then } \text{pipe}_{1}. \text{res} \\
\text{else } & \text{isRegFile} \text{[src]} \\
\text{else } & \text{nondet}
\end{cases}
\]
\[
\text{pipe}_{1}. \text{opr}_{2} := \begin{cases} 
\text{if } \text{stall} \text{ then } & \text{pipe}_{1}. \text{opr}_{2} \\
\text{else if } & \text{src} \neq \text{NOP} \land \text{pipe}_{1}. \text{opr} \neq \text{NOP} \land \text{pipe}_{1}. \text{res} \neq \text{STOR} \text{ then } \text{pipe}_{1}. \text{res} \\
\text{else } & \text{isRegFile} \text{[src]} \\
\text{else } & \text{nondet}
\end{cases}
\]
\[
\text{pipe}_{2}. \text{res}' := \begin{cases} 
\text{if } \text{pipe}_{2}. \text{op} \in \{ \text{AND}, \text{OR}, \text{LOAD} \} \\
\text{then } & \text{isRegFile} \text{[pipe}_{2}. \text{dest}'] \\
\text{else } & \text{nondet}
\end{cases}
\]

Figure 5: Three-stage pipeline