A Foundation for Component Verification in RESOLVE

William Leal and Anish Arora

Department of Computer and Information Science,
The Ohio State University, Columbus, Ohio 43210, USA
{leal, arora}@cis.ohio-state.edu

Technical Report Number OSU-CISRC-4/99-TR11
April, 1999

Abstract. Components are an increasingly popular way of designing and implementing systems. Since components can be verified in isolation, the task of verifying a system is simplified. The RESOLVE methodology gives a syntax-based proof method for verifying that a concrete component implements an abstract one. In this paper, we show that this method is not only sound, in terms of a trace-based semantic definition of component implementation, but also that it is incomplete. We redress this with a more general verification method that we show to be both sound and relatively complete. Although focused on RESOLVE, this proof method and semantic definition in fact have applicability in other component methodologies, and in data refinement approaches such as behavioral subtyping.

1 Introduction

The idea of a component as a system building block is not new. Components are increasingly viewed as "*the* cornerstone of software in the years to come" [16, p. xiii]. Components offer many advantages in system design, implementation, deployment and verification. These advantages stem from the fact that system functions can be developed, tested and verified in isolation, and systems can be updated incrementally by replacing individual components rather than massively by replacing the entire system. Components designed with adequate generality can be deployed and reused in different systems, reducing the need to provide and maintain multiple versions of the same code.

In software engineering, a component is typically treated as an abstract specification and a concrete implementation. RESOLVE is a software engineering discipline for expressing the specification and implementation of components, but beyond that it provides an integrated framework "in which all the important, but sometimes conflicting, aspects of software design can be considered at once" [14]. At the heart of the RESOLVE discipline is the idea of modularity: components can be developed and verified in isolation [3]. Because components encapsulate data types, their interactions are strictly controlled and so components can be composed into a system while maintaining guarantees as to correct behavior.

The goal of this paper is to show that the standard RESOLVE verification method is incomplete, but that there is a sound and relatively complete method that serves not only for RESOLVE but also for other component disciplines such as Seuss [11, 12] and for data refinement approaches such as behavioral subtyping [9].

RESOLVE Components. In RESOLVE, the specification and the implementation are separated; each is considered as a component in its own right [15]. This permits an *abstract component* (variously referred to as *specification* or *concept*) to have several associated *concrete components* (variously, *implementation* or *realization*) which differ according to performance. Based on patterns of data access, a developer or a dynamic system can substitute one realization for another, thus improving performance without compromising correctness.

A RESOLVE component consists of a set of exported types with an associated set of data values for each type and a set of exported operations. For abstract components, the operations are given as precondition-postcondition pairs (*requires-ensures*). For concrete components, the operations are given as procedures that call operations of other, previously-defined components. There is also a *convention* predicate that holds for all values that a concrete component variable can be in during computation; for example, if a concrete component is used to compute square roots, only nonnegative numbers would be permitted by the convention.

In Figure 1, the concept of an abstract component consisting of bags of strings is realized by a concrete component consisting of queues of strings. That the queue "behaves like" the bag can be seen through the effect the operations have on the string variables. If for instance three strings are added to the queue and then removed, we want that same sequence of operations to be one of the ways strings could be added to the bag and then removed. By saying "one of the ways," we are recognizing that a bag can remove items in a different order than a queue since a queue's remove operation behaves deterministically but the bag's remove operation is nondeterministic. This leaves the door open to other concrete realizations, such as a stack of strings, whose pattern of adding and removing is different than that of a queue but is consistent with a bag. The string variables in this example are called *indicator variables* since they indicate how the component is behaving. The *Bag_of_Strings* and *Queue_of_Strings* variables are called the *self variables* for the abstract and concrete components, respectively.

Our interest in this paper is in the verification of components: how do we know that a concrete component in fact implements an abstract component? The RESOLVE verification approach [15], detailed in the following section, identifies a relation called the *pointwise correspondence* which is defined over concrete and abstract values. This relation assures us that if some operation of the concrete component is able to transition from a predecessor state to a successor state then, using the abstract states related to the concrete's predecessor and succes-

```
concept Bag_of_Strings (* where ⊕ is bag
  union, admitting multiple instances *)
  (* Bag(string) is math description *)
  type family Container ∈ Bag(String)
  exemplar b

  initialization
    ensures b = ∅


  operation Add (
      alters b: Bag_of_Strings
      preserves y: String )
    requires true
    ensures b = @b ⊕ {y} ;

  operation Remove (
      alters b: Bag_of_Strings
      produces y: String )
    requires @b ≠ ∅
    ensures @b = b ⊕ y;
end Bag_of_Strings
```

```
realization Queue_of_Strings for
  Bag_of_Strings
  type Container is represented by Array
  convention true;


  initialization
    (body of initialization code)
  end;

  procedure Add (
    alters q: Queue_of_Strings
    preserves y: String )
    (body of add code)
  end Add;

  procedure Remove (
    alters q: Queue_of_Strings
    produces y: String )
    (body of remove code)
  end Remove;
end Queue_of_Strings
```

Fig. 1. Bag concept component with queue realization component. $@b$ refers to the input value of parameter b, b to the output value. Other details of RESOLVE syntax are in [3, 5].

sor states, respectively, the matching operation of abstract component can make the same transition. The RESOLVE verification method is believed to be sound [3] but not known to be complete. In fact, as we shall see in Sections 2 and 4, the pointwise correspondence is sound but not complete. (For ease of exposition we make several simplifications that do not affect the results; these simplifications are described in Section 5. For a more detailed description of RESOLVE components see [3, 5].)

Contributions. The first key contribution of this paper is a novel method for verifying that a concrete component implements an abstract one, richer than those that have been previously proposed. The second key contribution is to give a novel definition of component implementation. It includes not only the by-now standard condition of behavioral conformity but also a new condition, that of service conformity. These contributions form a foundation for verification not only for RESOLVE but also for other component and data type systems. We also show that RESOLVE's pointwise correspondence relation is sound and incomplete.

Outline of the paper. We show the incompleteness of the standard RESOLVE verification method in Section 2. In Section 3 we give the correspondence relation as a more general verification method. We develop a semantic trace-based definition of component implementation in Section 4 and show the correspondence relation to be sound and relatively complete with respect to this definition. We also show that the standard RESOLVE method is sound. In Section 5 we discuss related work and give conclusions. We include additional examples in Appendix A and give theorem proofs and other technical material in Appendix B.

2 The RESOLVE Verification Method and Its Incompleteness

We begin by defining RESOLVE's pointwise correspondence relation with which we can show that a realization R implements a concept T. The definition is adapted from [15]. The signature of a component is the set of its operation interfaces. Two signatures are said to be the same if they differ only in the names given to the interface arguments.

Concept T consists of a set of values, $values(T)$, an initialization ensures predicate, $init.ensures$, and a set of operations, each given by an interface and a requires and an ensures predicate. For operation P, these predicates are $P.requires$ and $P.ensures$, optionally qualified with the component name as in $T.P.requires$. Simultaneous substitution of x_1 by r_1 and x_2 by r_2 in predicate exp is given by $exp[t_1/x_1, t_2/x_2]$.

Realization R consists of a set of values, $values(R)$, an initialization procedure $init$ and a set of operation procedures, each given by an interface and a body of program code. Let P be an operation of realization R with interface $(x_1 : self, x_2 : self, z : Q)$; the discussion is easily adapted to lesser or greater numbers of $self$ or indicator arguments and to $init$ whose interface is assumed to be $(x : self)$. Procedure program code is built from calls to operations of other abstract components along with control structures for conditionals and looping. To aid in semantic analysis, the *requires* and *ensures* predicates of each called operation are available to the component, and loops are given a loop invariant predicate.

Let $(@r_1, @r_2, @d) \in values(R) \times values(R) \times values(Q)$. A total correctness semantics $\mathcal{M}_{tot}[\![P]\!](@r_1, @r_2, @d)$ is given to P of realization R (and similarly to $init$) as in Apt and Olderog [2]. $\mathcal{M}_{tot}[\![P]\!](@r_1, @r_2, @d)$ is defined as follows. When P starts on $(@r_1, @r_1, @d)$, $\mathcal{M}_{tot}[\![P]\!](@r_1, @r_2, @d)$ gives all the final values of (x_1, x_2, z) when P terminates, combined with up to two other values. The first is the usual $\perp$ to denote non-termination of some execution of P. The second is a new value, $abort$, which indicates that in the course of an execution of P from the starting values, an operation precondition or loop invariant failed; this is discussed in [6]. Thus an attempted execution of division by zero would result not in $\perp$ but in $abort$ since the precondition of the divide operation was violated. Notationally, if A is a relation then $A[x] = \{y : (x, y) \in A\}$. A relation $A \subseteq B \times C$ is image-finite if $\{c : \exists b : (c, b) \in A\}$ is finite.

Definition: Pointwise Correspondence Relation. $A \subseteq values(R) \times values(T)$ is a pointwise correspondence relation provided A is image-finite and the following three conditions hold.

1. **Totality over convention.** $\forall r \in values(R) :$ (if $conv[r/x]$ then $A[r]$ is not null).

2. **Initialization.** For initialization, let $R_f = \{r : \forall @r \in \mathit{values}(R) : (r \in \mathit{values}(R) \wedge r \in \mathcal{M}_{tot}[\![\mathit{init}]\!](@r) \wedge \mathit{abort} \notin \mathcal{M}_{tot}[\![\mathit{init}]\!](@r)\}$. Then following must hold.
   (a) **Initialization convention invariant.** $\forall r \in R_f : (\mathit{conv}[r/x])$.
   (b) **Initial value subsetting.** $\forall r \in R_f : \forall t \in A[r] : \exists @t \in \mathit{values}(T) : (\mathit{init.ensures}[@t/@x, t/x])$.
3. Let P be an operation, where we assume that P's interface is $P(x_1 : \mathit{self}, x_2 : \mathit{self}, z : Q)$ for both C and T. Let
   $R_i = \{ (@r_1, @r_2, @d) : @r_1, @r_2 \in \mathit{values}(R) \wedge @d \in \mathit{values}(Q) \wedge \mathit{conv}[@r_1/x] \wedge \mathit{conv}[@r_2/x]\}$,
   $R_f = \{ (r_1, r_2, d) : \exists(@r_1, @r_2, @d) \in R_i : (r_1, r_2, d) \in \mathcal{M}_{tot}[\![P]\!](@r_1, @r_2, @d) \wedge \mathit{abort} \notin \mathcal{M}_{tot}[\![P]\!](@r_1, @r_2, @d)\}$,
   $T_i = \{ (t_1, t_2, d) : \exists(r_1, r_2, d) \in R_i : (t_1 \in A[r_1] \wedge t_2 \in A[r_2])\}$, and
   $T_f = \{ (t_1, t_2, d) : \exists(r_1, r_2, d) \in R_f : (t_1 \in A[r_1] \wedge t_2 \in A[r_2])\}$.
   Then the following hold.
   (a) **Convention invariant.** $\forall(r_1, r_2, d) \in R_f : (\mathit{conv}[r_1/x] \wedge \mathit{conv}[r_2/x])$.
   (b) **Service condition.** $\forall(@r_1, @r_2, @d) \in R_i :$
   (if
   $\forall @t_1 \in A[@r_1] : \forall @t_2 \in A[@r_2] : (P.\mathit{requires}[@t_1/x_1, @t_2/x_2, @d/z])$
   then
   $\mathit{abort} \notin \mathcal{M}_{tot}[\![P]\!](@r_1, @r_2, @d))$
   (c) **Backward condition.** $\forall(@r_1, @r_2, @d) \in R_i : \forall(r_1, r_2, d) \in R_f : \forall(t_1, t_2, d) \in T_f :$
   (if
   $\forall @t_1 \in A[@r_1] : \forall @t_2 \in A[@r_2] :$
   $(P.\mathit{requires}[@t_1/x_1, @t_2/x_2, @d/z]) \wedge$
   $(r_1, r_2, d) \in \mathcal{M}_{tot}[\![P]\!](@r_1, @r_2, @d) \wedge$
   $\mathit{abort} \notin \mathcal{M}_{tot}[\![P]\!](@r_1, @r_2, @d) \wedge$
   $t_1 \in A[r_1] \wedge t_2 \in A[r_2]$
   then
   $\exists(@t_1, @t_2, @d) \in T_i : (@t_1 \in A[@r_1] \wedge @t_2 \in A[@r_2] \wedge$
   $P.\mathit{ensures}[@t_1/@x_1, @t_2/@x_2, @d/@z, t_1/x_1, t_2/x_2, d/z])$.

Condition 3b is called a service condition because, informally, it guarantees that if the image of a realization point is accepted by the concept then beginning at that point, the operation terminates or diverges; that is, if the concept gives service then the realization does too. Condition 3c is similar to the backward simulation of [10].

Now we turn to the question of incompleteness. Figures 2 and 3 show an abstract concept *Eager_Coin* and a concrete realization *Lazy_Coin* with a diagram that gives partial computations in a client program for both. *Eager_Coin* keeps two coins synchronized: the *Make_Match* operation takes two edge coins and nondeterministically make them both heads or both tails. The *Check_Were_Heads* operation reports whether the coins are heads or tails. *Lazy_Coin* also keeps two coins synchronized, but *Make_Match* only causes the coins to not be on edge. It is the *Check_Were_Heads* operation that nondeterministically decides whether they were heads or tails.

```
Concept Eager_Coin;
  (* Assume thick coins which readily edge
    balance *)
  type family Coin_State = {Head, Tail,
    Edge};
  exemplar C;

  initialization
    ensures C = Edge;

  operation Make_Match( var C, D:
      Coin_State );
    requires C = D = Edge;
    ensures C = D and C ≠ Edge;

  operation Check_Were_Heads( var C, D:
      Coin_State; var Ans: Boolean );
    requires C = D and C  Edge;
    ensures Ans = ( @C = Head ) and C =
      D = Edge;
  end Eager_Coin;
```

```
Realization Lazy_Coin for Eager_Coin;
  var RN: Random_Num;
  type Coin_State = {On_Edge, Matched}

  initialization
    C := On_Edge;
  end;

  procedure Make_Match( var C, D:
      Coin_State );
    C := Matched;
    D := Matched;
  end Make_Match;

  procedure Check_Were_Heads( var C, D:
      Coin_State; var Ans: Boolean );
    alters RN;
    C := On_Edge;
    D := On_Edge;
    Randomize(RN);
    (* Randomly return 0-9 *)
    If RN > 4 then Ans := True
      else Ans := False;
  end Check_Were_Heads;
end Lazy_Coin;
```

Fig. 2. Eager coin flipper implemented by lazy coin flipper.

It should be clear that the behaviors of *Eager_Coin* and *Lazy_Coin* in a client program are identical; however, there is no pointwise correspondence relation that will show this. The details are given in Example A.1. Intuitively, the problem comes after the *Make_Match*$(\#1, \#2)$ operation since *Eager_Coin* does not permit outcomes of $< Head, Tail >$ or $< Tail, Head >$.

3 Verification with the Correspondence Relation

In this section we give a definition of component that is based on semantics rather than syntax and use this to give the definition of the correspondence relation, which generalizes the pointwise correspondence. In this and following sections, C, N and B refer to components.

The major limitation of the standard RESOLVE method for verification is that it depends on a pointwise relation between the values of the concrete and the abstract components. To solve the problem we break away from dealing with individual points and instead take the semantic view of relations over sets of values as they evolve in the course of executing a program. We will treat each

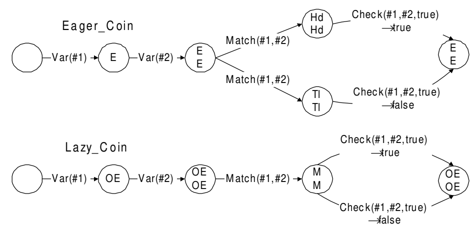


Fig. 3. Sample computations using eager and lazy coins. Abbreviations are used: "E" for "Edge", "OE" for "On_Edge", and so forth. Details of the notation are given in Sections 4 and 5.

component as a set of values and a collection of relations over the sets defined by their interface. For instance, the *Bag_of_Strings* component of Example 1.1 has as its set of values all bags of strings. Since the *Add* operation has an interface of (g: *self*, y: *string*), the relation is defined over a domain D of $\mathit{values}(\mathit{bag}) \times \mathit{values}(\mathit{string})$. By a slight abuse of notation, we treat *Add* not only as the name of the operation but as the operation itself, so $\mathit{Add} \subseteq D \times D$.

For concept components, the operation is calculated using the requires and ensures clauses. For *Bag_of_Strings* from Figure 1, $\mathit{Remove} = ((@x, @y), (x, y)) : @x \neq \emptyset \wedge @x = x \oplus \{y\}\}$. For concrete components, we can appeal to a semantic analysis of the code to give a relation for each operation. This reasoning gives us the following definition. (See Figure A.2,)

Definition: Component. A component C consists of the following.

1. $\mathit{values}(C)$ is a non-null set of values.
2. $\mathit{init}(C) \subseteq \mathit{values}(C)$ is a non-null subset of initial values.
3. $\mathit{ops}(C)$ is a set of operations, given as relations over the values of their arguments' components. For example, let T_1 and T_2 be components other than C and let Q be an operation in $\mathit{ops}(C)$ whose relation is over D and R where:
   (a) $D = \mathit{values}(C) \times \mathit{values}(T_1) \times \mathit{values}(C) \times \mathit{values}(T_2)$.
   (b) $R = (\mathit{values}(C) \times \mathit{values}(T_1) \times \mathit{values}(C) \times \mathit{values}(T_2)) \cup \{\perp\}$. □

The bottom state, $\perp$, indicates non-abortion divergence. Note that *abort* is not in the relation.

Now that we have each component represented in a common framework we can drop the requirement that abstract components be given by requires/ensures clauses and that concrete components be given by program code: components merely consist of value sets and relations. In the sequel, one component is labeled as abstract and the other as concrete only in the sense that we make the designation to discuss implementation. Any component could, at any time, be abstract or concrete.

Making this change opens up other opportunities as well. Since verification is no longer tied to syntax, we can verify whether one concept implements another. If we have a *Stack_of_Strings* concept and a *Bag_of_Strings* concept, we can verify whether *Stack_of_Strings* implements *Bag_of_Strings*. Suppose that a linked-list realization *LL_String_Stack* implements *Stack_of_Strings* which in turn implements *Bag_of_Strings*. Since the implementation relation is transitive we can conclude that *LL_String_Stack* implements *Bag_of_Strings* and hence could serve as a realization for it.

One way to tell whether one component implements another is to see if, when the first is substituted for the second in some program, the new program behaves like the old one up to nondeterminism. Intuitively, this is what we want in component implementation. "Behaves like" is given a more precise meaning in Section 4. For the moment, we merely note that carrying this out means looking at all programs and comparing the behaviors of those programs when used with the concrete component or when used with the abstract component. This in turn means dealing with an infinite number of programs, any of which could have an infinite number of infinite behaviors. We want a reasoning method that limits what we have to inspect and so makes the job easier, and this is what the general correspondence relation gives us. Under the correspondence we need only consider a state and its successor due to some concrete component operation and compare the images of those states under the correspondence relation.

Now we turn to the definition of the correspondence relation. The correspondence relation is defined over the computation state spaces of the concrete and abstract components, so vectors of concrete values are related to vectors of abstract values. Before giving the definition of the general correspondence we need to define several terms.

Definition: State. Given component C, a state is a finite, possibly null, vector of values of C or is $\bot$. If $s \neq \bot$ then $s^{(i)}$ refers to the ith position in the state s. The states of C are denoted by $states(C)$. $\langle\rangle$ denotes the null state. $\square$

Definition: Event. Let C be a component and P an operation with interface $(\mathit{self}, T1, \mathit{self}, T2)$ for components $T1$ and $T1$. There are two kinds of event. (a) An event e of C may be of the form $P(\#i, @a1, \#j, @a2) \rightarrow (a1, a2)$ or $P(\#i, @a1, \#j, @a2) \rightarrow \bot$, where $@a1$ and $a1$ are values of $T1$, and $@a2$ and $a2$ are values of $T2$. $\#i$ and $\#j$ refer to positions in a state space vector. The part before the arrow is called the invocation and the part afterwards is called the result. (b) An event may be $var(\#i)$ to indicate variable initialization. The invocation is the same as the event and there is no result. The events of C are denoted by $events(C)$. $\square$

Definition: Transition. Let C be a component, e be an event, and $@s$ and s be states. A transition $@s \xrightarrow{e}_C s$ of C has three forms. (a) If e is of the form $P(\#j, @a1, \#k, @a2) \rightarrow (a1, a2)$ then $@s$ and s are the same size and are at least as large as the larger of j and k, and $(s^(j), a1, s^(k), a2)$ is in $P[(@s^(j), @a1, @s^(k), @a2)]$. (b) If e is of the form $P(\#j, @a1, \#k, @a2) \rightarrow \bot$ then $@s$ must be at least as large as the larger of j and k and $s = \bot$. (c) If e is $var(\#j)$, defining a new variable, then $@s$ must be size $j-1$, s must be size j, and $s(j) \in init(C)$. In all cases, unreferenced elements of $@s$ and s are identical. The transitions of C are denoted by $transitions(C)$. $\square$

For example, $\langle 1, 9, 5\rangle$ is a state for a component whose values are the integers. *Check_Were_Heads*$(\#1, \#2, \mathit{true}) \rightarrow \mathit{false}$ is an event in Figure 2, which has a transition of

$$\langle \mathit{Tail}, \mathit{Tail}\rangle \xrightarrow{\mathit{Check\_Were\_Heads}(\#1,\#2,\mathit{true})}_{\mathit{Eager\_Coin}} \langle \mathit{Edge}, \mathit{Edge}\rangle.$$

Definition: Enablement. Given component C, event e and state $@s$, then e is enabled on $@s$ in C if there is some event e' and some state s s.t. $@s \xrightarrow{e'}_C s$ and the invocation parts of e and e' are the same. □

Since in general not every concrete state is of interest, we augment the concrete component with a *convention*, similar in spirit to the pointwise convention. The concrete convention must be a superset of the computable states of the component, where computable states are defined in Section 4.1. The correspondence relation need only be defined over the convention.

Definition: Correspondence. Suppose N and B have the same signature. Let $\gamma \subseteq \mathit{values}(N) \times \mathit{values}(B)$ be an image-finite relation that is total on the convention of N, rc. Then γ is a correspondence from N to B provided the following hold:

1. **Initial state subsetting.** $\gamma[\langle\rangle] = \{\langle\rangle\}$.
2. Let $@s$ be a state of rc and let e be an event that is enabled on each $@u$ of $\gamma[@s]$. In that case, the following both hold.
   (a) **Service simulation.** e is enabled on $@s$ in N.
   (b) **Backward simulation.** If there are s and u s.t. $@s \xrightarrow{e}_N s$ and $u \in \gamma[s]$ then there is $@u \in \gamma[@s]$ s.t. $@u \xrightarrow{e}_B u$.

If a correspondence can be exhibited for N and B, then we have verified that N implements B. In Example A.4, a correspondence is used to verify the example of Figure 2, which could not be verified with the standard method.

4 Soundness and Completeness

In this section we develop a trace-based definition of component implementation and show that both the general correspondence and the pointwise correspondence relations are sound with respect to this definition. We define finite invisible nondeterminism (fin) and show that the general correspondence relation is complete provided the abstract component has fin.

We will develop an operational definition of implementation by characterizing all the possible behaviors of a component and relating them to those of another component. Then we will show that under this definition of implementation, if there is a correspondence from one component to another then the first implements the second. Conversely, we will show that, with certain restrictions, if one component implements another then there is a correspondence from the first to the second.

The essential idea of component implementation is that of substitutability. Suppose a program is written using the component *Bag_of_Strings*. Then if we substitute *Queue_of_Strings* for *Bag_of_Strings*, the new program should behave like the original one, though perhaps with less nondeterminism, similar to the goals of Liskov and Wing [9] and Leavens [8]. For the new program to "behave like" the original program, a concrete component must meet two requirements. First, the component must give service: when a program calls on an abstract operation for some service, the concrete operation must also be prepared to give service. Second, the component must respect behavior: when a concrete operation executes, its behavior must be consistent with what the abstract operation would have done. We describe this precisely in Sections 4.1 and 4.2.

4.1 Scenarios

Scenarios give us a way of describing the "interesting" behavior of a component in a computation. In comparing the behavior of one component to another, such as *Bag_of_Strings* to *Queue_of_Strings*, we are not particularly interested in how the bags or queues are maintained, nor are we interested in comparing bags with queues directly. Instead, we want to know what the components can actually do in a client computation. For example, if a client program instantiates a bag, we want to know whether every item added can be removed, and only those items. Thus the sequence of defining a bag instance, adding the strings "a" and "b" to the bag and removing the string "a" would be correct but the same sequence that removes the string "c" would not be.

Our computational model is a state space with events that change states via transitions. Computations consist of sequences of transitions where the final state of one transition is the initial state of its successor. Traces project away the state and form a basis for comparing the behavior of two components. Not all traces are of interest since we want to consider only well-formed client programs, so scenarios are a well-formed subset of the traces. These terms are defined below.

Definition: Computation. Given component C, a computation of C is an interleaved sequence of states and events $\ll s_0 e_1 s_1 e_2 s_2 ... \gg$ that begins with a state, ends with a state if finite, and satisfies the following conditions:

1. The first state, s_0, is $\langle\rangle$.
2. If $s_{i-1} e_i s_i$ appears in the sequence then $s_{i-1} \xrightarrow{e_i}_C s_i$ must be a transition.

$\Box$

Definition: Trace. Given component C, a trace of C is a sequence of events $\ll e_1 e_2 ... \gg$ s.t. there is and interleaving with some sequence of states that forms a computation of C. λ stands for the null trace. $\Box$

Definition: Computable State of Scenario. Given component C and finite trace $\sigma = \ll e_1 e_2 ... e_n \gg$ then the states computable by σ in C is the set of all s_n s.t. there is an interleaving $\ll s_0 e_1 s_1 e_2 s_2 ... s_{n-1} e_n s_n \gg$ of state with σ that forms a computation of C. $\Box$

Not every trace is well-formed. Suppose we had a trace $\ll e_1 e_2 e_3 \gg$ where the states computable by $\ll e_1 e_2 \gg$ are S, and e_3 is enabled on some but not all states in S. This means we have a program that has executed $\ll e_1 e_2 \gg$ and now expects to do e_3, but in some nondeterministic cases is not able to, and in those cases would unexpectedly terminate. We consider such a program ill-formed since it may abort unexpectedly depending on nondeterministic results of operations. We eliminate those cases by restricting traces of interest to scenarios, defined next.

Definition: Scenario. Given component C, a scenario of C is inductively defined as a trace σ of C s.t. for every prefix σ' of σ,

1. σ' is λ or
2. σ' is $\sigma''; e$ where σ'' is a scenario of C and e is enabled on every state computable by σ'' in C. □

4.2 Definition of Component Implementation

Scenarios give us just the behaviors of interest for a component since they identify the sequence of operations that can take place on instances of the component and show the indicator inputs and outputs. Since they suppress the state of the instances themselves, two components with the same signature will have sets of scenarios that can be compared.

Service Guarantees. If a concrete component is to be an implementation of an abstract component, the first condition the concrete component must offer is service. Suppose there is a scenario σ' in common between the concrete and the abstract. Suppose for some event e, $\sigma'; e$ is also a scenario of the abstract. This means that the abstract was able to make some invocation and get some result. Then we require that the concrete also be able to make the same invocation, though perhaps getting a different result. That is, there must be an event f s.t. $\sigma'; f$ must be a scenario of the concrete, where the invocation of e is the same as the invocation of f. See Examples A.5 and A.6. These ideas are made precise in the following definitions. (See Example A.7.)

Definition: Scenario extension. Given component C and scenario σ' of C, $\sigma'; v$ is an extension of C if v is the invocation part of an event e s.t. $\sigma; e$ is a scenario of C. We write $\mathit{extensions}(N)$ to denote this set of extensions and $\mathit{inv}(e)$ to denote the invocation part of e. □

Definition: Required extensions. Let N and B be components and let $\sigma'; v$ be an extension of B. Then $\sigma'; v$ is a required extension of B with respect to N if σ' is a scenario of N. We write $\mathit{extensions}(B/N)$ to denote the set of required extensions. □

Definition: Service Requirement. The service requirement for component implementation of B by N is met if $\mathit{extensions}(B/N)$ are a subset of $\mathit{extensions}(N)$. □

Behavior Guarantees. Definitions of implementation are generally based on subsetting [10]. For us, it means that we want to ensure that the scenarios of an abstract component are a subset of the scenarios of a concrete component. This is not always possible. In Example A.3, operation *Remove* in *Queue_of_Strings* is total, which means *Remove* is defined even on empty queues. *Queue_of_Strings* has the scenario $\ll \mathit{Var}(\#1), \mathit{Remove}(\#1, "a") \rightarrow "a" \gg$, which is not a scenario of *Bag_of_Strings*. We do not want this fact to force us to reject *Queue_of_Strings* as an implementation. This totality should not be a problem since, if a program is correctly written using *Bag_of_Strings*, it will never invoke *Remove* on an empty bag; and when *Queue_of_Strings* is substituted as an implementation, it will never invoke *Remove* on an empty queue. The solution is to again make use of extensions.

Definition: Useful scenarios. Let σ be a scenario of N. Then σ is a useful scenario of N with respect to B if every extension prefix of σ is an extension of B. We write $scenarios(N/B)$ to denote the set of useful scenarios. □

Note that by this definition, $\ll Var(\#1), Remove(\#1, "a") \rightarrow "a" \gg$ is not a useful scenario of *Queue_of_Strings* with respect to *Bag_of_Strings* since $\ll Var(\#1), Remove(\#1, "a") \gg$ is not an extension of *Bag_of_Strings*.

Definition: Behavior Requirement. The behavior requirement for component implementation of B by N is met if $scenarios(N/B)$ are a subset of $scenarios(B)$. □

We observe that the scenarios of *Queue_of_Strings* with respect to *Bag_of_Strings* are in fact a subset of the scenarios of *Bag_of_Strings*.

Component implementation: service and behavior combined. We are now ready to give the definition of component implementation as a composite of service and behavior.

Definition: Component implementation. N implements B if the service and behavior requirements are met:

1. $extensions(B/N) \subseteq extensions(N)$
2. $scenarios(N/B) \subseteq scenarios(B)$ □

Note that *Queue_of_Strings* and *Bag_of_Strings* meet the service and behavior requirements, so *Queue_of_Strings* implements *Bag_of_Strings*.

4.3 Soundness and Completeness Results

The soundness of the correspondence relation and of the pointwise correspondence relation is given by the following results. Proofs for the theorems are in Appendix B.

Theorem 4.1: Soundness of the correspondence relation.
If there is a correspondence relation from N to B
then N implements B. □

Theorem 4.2: Soundness of the pointwise correspondence relation.
If there is a pointwise correspondence relation from N to B
then N implements B. □

Completeness of the correspondence relation is with respect to two conditions. First, it may be necessary to introduce an adjunct history variable [1, 10] into the computations of the concrete component. This distinguishes states that were previously undistinguished but does not affect scenarios.

Definition: History adjunction. Component C is history adjoined to give C^a if its state space is adjoined with history s.t. every scenario σ of C^a computes (s, σ) where s is computable by σ in C. □

The second condition for completeness is that the abstract component must have finite invisible nondeterminism [1, 10].

Definition: Finite invisible nondeterminism. Component C has finite invisible nondeterminism (fin) if for every finite scenario σ of C, the set of states computable by σ is finite. □

Since constructing a history adjunction is always possible, the only real restriction is that the abstract must have fin. If this condition is met, we have the following result.

Theorem 4.3: Relative completeness of the correspondence relation.
If N implements B and B has fin
then there is a correspondence from a history adjunction of N to B.
□

5 Related Work and Conclusions

Related Work. In this discussion we made several simplifications of RESOLVE. RESOLVE components are parameterized and are instantiated as *facilities*. A *Bag* component, for instance, can be parameterized as for the kinds of items stored, so *Bag*(*String*) would be a facility. In this sense, our discussion here concerns facilities, not components. RESOLVE defines a *constraint* predicate for concepts; this is a programming convenience, and we assume that the constraint condition has been conjoined to the *requires* and *ensures* predicates of *init* and each operation. Existing extensions to RESOLVE components include the addition of a communal variable that can be shared by all instances of a component. This has been used to model the memory management of linked lists [6]. Additional proposals have been made to extend RESOLVE by permitting a component to export two or more types, by including global variables (accessible by any component), and multiple-instance access (letting all instances of a component be directly accessible by that component). The correspondence remains sound and relatively complete in these cases, although space does not permit a justification of this claim here. There also appears to be no problem extending the idea of component and component verification to include spontaneously initiated actions of the sort contained in Seuss [11, 12].

In the proof of completeness we rely on the completeness results of Lynch and Vaandrager [10]. The backward simulation condition of the correspondence is very similar to the backward simulation they present. However, their framework has no concept of enablement nor consequently of extensions, ideas that are critical to our formulation of service and behavior.

The goals of component implementation and behavioral subtyping [9, 7] are similar: to ensure that one component (type) implements another if upon substituting the first component for the second in any program, the program continues to work as expected. With the introduction of the idea of partial substitution of components, the definition of component implementation may be adapted to the setting of behavioral subtyping and used as a basis for showing soundness, though space prohibits a detailed development here. Further, the notion of an abstraction function may be extended to that of an abstraction relation, expanding the set of types that may be verified.

The work done here overlaps with that of de Roever and Engelhardt [4]. A feature common to their approach and ours is that their abstraction relation is not pointwise, but is over the entire variable space of the component, similar to the correspondence relation presented here. There are however several important differences. First, they deliberately limit the notion of data type to be parameterized over a fixed set of variables. This contrasts with the unbounded number of instances provided for in this presentation. There are examples in which one component appears to implement another for a fixed number of instances, but does not implement it for a larger number of instances. Second, they view a data type as monolithic, including not only the operations that manipulate representation (self) variables but also those that manipulate normal (indicator) variables. This severely restricts the ability to define and verify components in isolation, a feature that our set-up permits. Third, their data refinement relation permits implementation that ours forbids. Their definition of data type (component) implementation gives behavior guarantees but not service. Under this view, *Bounded_Queue_of_Strings* would implement *Bag_of_Strings* (Example A.5). We believe this should not be the case, and so forbid it in our definition. Fourth, their data refinement relation forbids implementations that ours accepts. The trace / scenario distinction in our view separates ill-formed from well-formed programs. In their view, there are no ill-formed programs, causing them to reject certain implementations that we accept. This is illustrated in Example A.9.

Conclusions. In this paper we have provided a missing foundation for component verification in RESOLVE. If the standard method using pointwise correspondences works, the developer is free to use it, confident in its soundness. If no such pointwise correspondence can be found, the more general correspondence can be used, subject only to the restriction that the abstract component must have fin.

Since the definition of component is semantically-based rather than depending on RESOLVE syntax, it can be applied to other kinds of component or data type systems, and the correspondence can be used for verification.

Acknowledgments. Bill Ogden [13] provided several important ideas for this paper. He gave the definition of the general correspondence, and, based on the idea of scenarios, proposed the behavior requirement as a condition for component implementation. We are grateful for his involvement in discussions and in reviewing drafts of this paper. We are also grateful Neelam Soundarajan for his help on semantic issues and to Bruce Weide and the rest of the RESOLVE team for their productive discussions and comments on this material.

References

1. Martin Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82:253–284, 1991.
2. Krzysztof R. Apt and Ernst-R diger Olderog. *Verification of sequential and concurrent programs.* Graduate texts in computer science. Springer-Verlag New York Inc., New York, 2nd edition, 1997.
3. Paolo Bucci, Joseph E. Hollingsworth, Joan Krone, and Bruce W. Weide. Implementing components in resolve. *Software Engineering Notes*, 19(4):40–51, 1994.
4. Willem-Paul de Roever and Kai Engelhardt. *Data refinement: Model-oriented proof methods and their comparison.* Cambridge University Press, Cambridge, 1998.
5. Stephen H. Edwards, Wayne D. Heym, Timothy J. Long, Murali Sitaraman, and Bruce W. Weide. Specifying components in resolve. *Software Engineering Notes*, 19(4):29–39, 1994.
6. George W. Ernst, Raymond J. Hookway, and William F. Ogden. Modular verification of data abstractions with shared realizations. *IEEE Transactions on Software Engineering*, 20(4):288–307, 1994.
7. Gary T. Leavens. Modular verification of object-oriented programs with subtypes, 1990.
8. Gary T. Leavens and Don Pigozzi. An exact algebraic characterization of behavioral subtyping, 1996.
9. Barbara H. Liskov and Jeanette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994.
10. Nancy Lynch and Frits Vaandrager. Forward and backward simulations, part i: Untimed systems. *Information and Computation*, 121(2):214–233, 1995.
11. Jayadev Misra. A discipline of multiprogramming. In Guy E. Blelloch, K. Mani Chandy, and Suresh Jagannathan, editors, *Specification of Parallel Algorithms: DIMACS workshop, May 9-11, 1994*, pages 357–382. American Mathematical Society, Providence, RI, 1994.
12. Jayadev Misra. A discipline of multiprogramming, 1999.
13. William Ogden, 1998.
14. William F. Ogden, Murali Sitaraman, Bruce W. Weide, and Stuart H. Zweben. The resolve framework and discipline–a research synopsis. *Software Engineering Notes*, 19(4):23–28, 1994.
15. Mulali Sitaraman, Bruce Weide, and Bill Ogden. On the practical need for abstraction relations to verify abstract data type representations. *IEEE Transactions on Software Engineering*, 23(3):157–170, 1997.
16. Clemens Szyperski. *Component software: Beyond object-oriented programming.* Addison-Wesley - ACM Press, New York, 1998.

A Appendix: Examples

Example A.1. No pointwise correspondence for Figure 2.
The realization convention, *conv*, is not given in Figure 2 and so is taken to be *true*.

Now we consider all possible cases for the pointwise correspondence relation, $A \subseteq \{On\_Edge, Matched\} \times \{Head, Tail, Edge\}$. Conditions cited are from the definition of pointwise correspondence.

Case 1. $A[\mathit{On\_Edge}] \neq \{\mathit{Edge}\}$. Consider *init*. $R_f = \{\mathit{On\_Edge}\}$. Since $A[\mathit{On\_Edge}] \neq \emptyset$, either *Head* or *Tail* must be in $A[\mathit{On\_Edge}]$. But condition 2b fails since $(C = \mathit{Edge})$ does not hold for *Head* or *Tail*.

Case 2. $A[\mathit{On\_Edge}] = \{\mathit{Edge}\}$ and $\mathit{Edge} \in A[\mathit{Matched}]$. Consider *Make_Match*. $R_i = \{(\mathit{On\_Edge}, \mathit{On\_Edge})\}$, $R_f = \{(\mathit{Matched}, \mathit{Matched})\}$, $T_i = \{(\mathit{Edge}, \mathit{Edge})\}$ and $T_f = A[R_f]$. The backward condition 3c fails since $(C, D) = (\mathit{Edge}, \mathit{Edge}) \in T_f$ but there is no $(@C, @D) \in T_i$ s.t. $(C = D \wedge C = \mathit{Edge})$ holds for $(@C, @D)$.

Case 3. $A[\mathit{On\_Edge}] = \{\mathit{Edge}\}$ and $A[\mathit{Matched}] = \{\mathit{Head}\}$.
Consider *Check_Were_Heads*.
$R_i = \{(\mathit{Matched}, \mathit{Matched}, \mathit{true}), (\mathit{Matched}, \mathit{Matched}, \mathit{false})\}$,
$R_f = \{(\mathit{On\_Edge}, \mathit{On\_Edge}, \mathit{true}), (\mathit{On\_Edge}, \mathit{On\_Edge}, \mathit{false})\}$,
$T_i = \{(\mathit{Head}, \mathit{Head}, \mathit{true}), (\mathit{Head}, \mathit{Head}, \mathit{false})\}$ and
$T_f = \{(\mathit{Edge}, \mathit{Edge}, \mathit{true}), (\mathit{Edge}, \mathit{Edge}, \mathit{false})\}$.
The backward condition 3c fails since $(C, D, \mathit{Ans}) = (\mathit{Edge}, \mathit{Edge}, \mathit{false}) \in T_f$ but there is no $(@C, @D, \mathit{false}) \in T_i$ s.t. $(\mathit{false} = (@C = \mathit{Head}) \wedge C = D = \mathit{Edge})$ holds for $(@C, @D, \mathit{Ans})$.

Case 4. $A[\mathit{On\_Edge}] = /\mathit{Edge}/$ and $A[\mathit{Matched}] = \{\mathit{Tail}\}$. This fails the backward condition 3c for *Check_Were_Heads* similarly to the preceding case.

Case 5. $A[\mathit{On\_Edge}] = \{\mathit{Edge}\}$ and $A[\mathit{Matched}] = \{\mathit{Head}, \mathit{Tail}\}$.
Consider *Make_Match*.
$R_i = \{(\mathit{On\_Edge}, \mathit{On\_Edge})\}$,
$R_f = \{(\mathit{Matched}, \mathit{Matched})\}$,
$T_i = \{(\mathit{Edge}, \mathit{Edge})\}$ and
$T_f = \{\mathit{Head}, \mathit{Tail}\} \times \{\mathit{Head}, \mathit{Tail}\}$.
The backward condition 3c fails since $(C, D) = (\mathit{Head}, \mathit{Tail}) \in T_f$ but there is no $(@C, @D) \in T_i$ s.t. $(C = D \wedge C = \mathit{Edge})$ holds for $(@C, @D)$.

Since all possible cases of A have failed, we conclude that no pointwise correspondence exists.

Example A.2. *Eager_Coin* / *Lazy_Coin* **Components.**

These match the components of Figure 2

Component *Eager_Coin* $\mathit{values}(\mathit{Eager\_Coin}) = \{\mathit{Head}, \mathit{Tail}, \mathit{Edge}\}$ $\mathit{init}(\mathit{Eager\_Coin}) = \{\mathit{Edge}\}$ Operation $\mathit{Make\_Match}(C : \mathit{self}, D : \mathit{self}) = \{((@C, @D), (C, D)) : @C = @D = \mathit{Edge} \wedge C = D \wedge C \neq \mathit{Edge}\}$ Operation $\mathit{Check\_Were\_Heads}(C : \mathit{self}, D : \mathit{self}, \mathit{Ans} : \mathit{Boolean}) = \{((@C, @D, @\mathit{Ans}), (C, D, \mathit{Ans})) : @C = @D \wedge @C \neq \mathit{Edge} \wedge \mathit{Ans} = (@C = \mathit{Head}) \wedge C = D = \mathit{Edge}\}$	Component *Lazy_Coin* $\mathit{values}(\mathit{Lazy\_Coin}) = \mathit{Match}, \mathit{Edge}$ $\mathit{init}(\mathit{Lazy\_Coin}) = \mathit{Edge}$ Operation $\mathit{Make\_Match}(C : \mathit{self}, D : \mathit{self}) = \{((@C, @D), (C, D)) : C = D = \mathit{Matched}\}$ Operation $\mathit{Check\_Were\_Heads}(C : \mathit{self}, D : \mathit{self}, \mathit{Ans} : \mathit{Boolean}) = \{((@C, @D, @\mathit{Ans}), (C, D, \mathit{Ans})) : C = D = \mathit{Edge}\}$

Example A.3. *Bag_of_Strings* / *Queue_of_Strings* **Components.**

These match the components of Figure 1. Note: *Concat*, *First* and *AllButFirst* have the expected meanings.

Component *Bag_of_Strings* (* Take {} here to refer to a bag, not a set and $\oplus$ as bag union *) $values(Bag\_of\_Strings) = \{y : y \in values(string)\}$ $init(Bag\_of\_Strings) = \emptyset$	Component *Queue_of_Strings* $values(Queue\_of\_Strings) = values(string)^*$ $init(Queue\_of\_Strings) = \langle\rangle$
Operation $Add(b : self, y : string) = \{((@b, @y), (b, y)) : b = @b \oplus \{@y\} \wedge @y = y\}$	Operation $Add(q : self, y : string) = \{((@q, @y), (q, y)) : q = Concat(@q, \langle @y \rangle) \wedge @y = y\}$
Operation $Remove(b : self, y : string) = \{((@b, @y), (b, y)) : @b \neq \emptyset \wedge @b = b \oplus \{y\}\}$	Operation $Remove(q : self, y : string) = \{((@q, @y), (q, y)) : (@q = \langle\rangle \Rightarrow @q = q) \wedge (@q \neq \langle\rangle \Rightarrow q = AllButFirst(@q) \text{and} y = First(@q)\}$

Example A.4. Verification of *Eager_Coin* / *Lazy_Coin*, Example A.2.

Define a correspondence γ over scenario states. For Example A.2, γ is defined in part as follows, where $\langle\rangle$ means a null state with no instances.

$\langle\rangle \mapsto \langle\rangle$	$\langle On\_Edge \rangle \mapsto \langle Edge \rangle$
$\langle On\_Edge,\ On\_Edge \rangle \mapsto \langle Edge,\ Edge \rangle$	$\langle Matched,\ Matched \rangle \mapsto \langle Tail,\ Tail \rangle$ $\mapsto \langle Head,\ Head \rangle$

The invariant realization convention for *Lazy_Coin* defines all reachable states: $rc = \{\langle a_1, a_2, \ldots, a_n \rangle : n \geq 0$ and the count of *Matched* in $\langle a_1, a_2, \ldots, a_n \rangle$ is even$\}$.

Then the proposed correspondence

$$\gamma \subseteq values(Lazy\_Coin)^* \times values(Eager\_Coin)^*$$

is given by $\gamma = \{(\langle a_1, a_2, \ldots, a_n \rangle, \langle b_1, b_2, \ldots, b_n \rangle) : n \geq 0$ and (if $a_i = Matched$ then $b_i = Head$ or $Tail$ else $b_i = Edge$) and the count of *Head*'s in $\langle b_1, b_2, \ldots, b_n \rangle$ is even$\}$.

To verify the components, we must check that the realization convention is correct, γ is total on the realization convention, γ is image-finite and that $\gamma[\langle\rangle] = \langle\rangle$. Then we must consider all realization convention states and all possible events to be sure that the two required simulations hold.

First we check the general conditions.

1. Checking the realization convention is an easy induction since the initial state is in the convention and the *Var*, *Make_Match* and *Check_Were_Heads* operations preserve the convention.
2. Next we must check the totality of γ over rc. Let $s \in rc$. Let u be s.t. $u^{(j)} = Heads$ if $s^{(j)} = Matched$ else $u^{(j)} = Edge$. Since the count of *Matched* in s is even, the count of *Heads* in u is even, so $(s, u) \in \gamma$.
3. For a given $s \in rc$ there are only a finite number of u s.t. $(s, u) \in \gamma$, so γ is image-finite.

4. If $s = \langle\rangle$ and $u \in \gamma[s]$ then $u = \langle\rangle$ and there is no other u in $\gamma[s]$, so $\gamma[\langle\rangle] = \langle\rangle$.

Next we need to check the operations. Assume in all cases that $@s$ is in rc and that the event e is enabled in *Eager_Coin* on all $@u \in \gamma[@s]$.

1. Operation *Var*. Event e is *Var*$(\#i)$. By the definition of *Var*, $@s$ must be of size i-1.
   - **Service Simulation.** Since *Lazy_Coin.Var*$(\#i)$ is enabled on all states it is enabled on $@s$.
   - **Backward Simulation.** Suppose $@s \xrightarrow{Var(\#1)}_{Lazy\_Coin} s$ and $u \in \gamma[s]$. We must show that there is $@u \in \gamma[@s]$ s.t. $@u \xrightarrow{Var(\#1)}_{Eager\_Coin} u$. Let $@u$ be of length i s.t. $@u^{(j)} = u^{(j)}$ for $j < i$. Since $@s$ and s differ only in that s is one longer than $@s$ and has *On_Edge* in the ith position, $@u$ is in $\gamma[@s]$. By the definition of γ, $@u$ has *Edge* in the ith position. Consequently, $@u \xrightarrow{Var(\#1)}_{Eager\_Coin} u$.
2. Operation *Make_Match*. Event e is *Make_Match*$(\#i, \#j)$.
   - **Service Simulation.** Since *Lazy_Coin.Make_Match* is total, it is enabled on $@s$.
   - **Backward Simulation.** Suppose $@s \xrightarrow{Make\_Match(\#i,\#j)}_{Lazy\_Coin} s$ and $u \in \gamma[s]$. We must show that there is $@u \in \gamma[@s]$ s.t.

     $$@u \xrightarrow{Make\_Match(\#i,\#j)}_{Eager\_Coin} u.$$

     Let $@u$ be the same as u except that $@u^{(i)} = @u^{(j)} = Edge$. Since *Eager_Coin.Make_Match*$(\#i, \#j)$ is enabled on all $@u$ in $\gamma[@s]$, $@s^{(i)} = @s^{(j)} = On\_Edge$, so $@u \in \gamma[@s]$. By the definition of *Lazy_Coin.Make_Match*, $s^{(i)} = s^{(j)} = Matched$, so $u^{(i)} = u^{(j)} = Head$ or $u^{(i)} = u^{(j)} = Tail$, and $@u \xrightarrow{Make\_Match(\#i,\#j)}_{Eager\_Coin} u$.
3. Operation *Check_Were_Heads*. Event e is *Check_Were_Heads*$(\#i, \#j, @b) \rightarrow b$.
   - **Service Simulation.** Since *Lazy_Coin.Check_Were_Heads* is total, it is enabled on $@s$.
   - **Backward Simulation.** Suppose

     $$@s \xrightarrow{Check\_Were\_Heads(\#i,\#j,@b)\rightarrow b}_{Lazy\_Coin} s$$

     and $u \in \gamma[s]$. We must show that there is $@u \in \gamma[@s]$ s.t.

     $$@u \xrightarrow{Check\_Were\_Heads(\#i,\#j,@b)\rightarrow b}_{Eager\_Coin} u.$$

     - Suppose b is *true*. Let $@u$ be the same as u except that $@u^{(i)} = @u^{(j)} = Head$. Since *Eager_Coin.Check_Were_Heads*$(\#i, \#j)$ is enabled on all $@u$ in $\gamma[@s]$, $@s^{(i)} = @s^{(j)} = Matched$, so $@u \in \gamma[@s]$. By the definition of *Lazy_Coin.Check_Were_Heads*, $s^{(i)} = s^{(j)} = On\_Edge$, so $u^{(i)} = u^{(j)} = Edge$. Since $@u^{(i)} = @u^{(j)} = Head$ and $b = true$,

       $$@u \xrightarrow{Check\_Were\_Heads(\#i,\#j,@b)\rightarrow b}_{Eager\_Coin} u.$$

- Suppose b is *false*. This check is similar to the case $b = true$.

From this we may conclude that γ is a correspondence from *Lazy_Coin* to *Eager_Coin* and so *Lazy_Coin* implements *Eager_Coin*.

Example A.5. Bounded queue.

Component *Bounded_Queue_of_Strings*
(* Values are null queue or queue with one string in it *)
$values(Bounded\_Queue\_of\_Strings) = values(string)^{\leq 1}$
$init(Bounded\_Queue\_of\_Strings) = \langle\rangle$

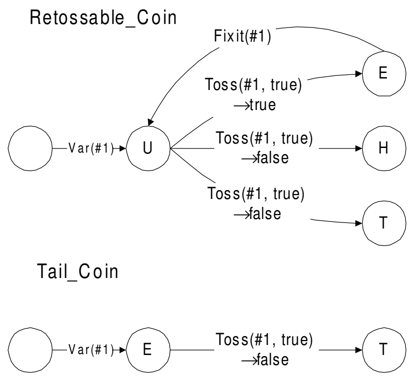
(* The queue only holds one string, so *Add* is defined on empty queues only *)
Operation $Add(q : self, y : string) = \{((@q, @y), (q, y)) :$
$@q = \langle\rangle \wedge$
$q = \langle @y \rangle \wedge @y = y\}$

Operation $Remove(q : (self), y : string) = \{((@q, @y), (q, y)) :$
$(@q = \langle\rangle \Rightarrow q = @q \wedge y = @y) \wedge$
$(@q \neq \langle\rangle \Rightarrow q = \langle\rangle \wedge y = First(@q)\}$

In this example we have the component *Bounded_Queue_of_Strings*, which is identical to *Queue_of_Strings* (Example A.3) except that a queue can only hold one item. Does this implement *Bag_of_Strings*? A program using *Bag_of_Strings* could create an instance and then add two strings. That would be impossible if we substituted *Bounded_Queue_of_Strings* for *Bag_of_Strings* in the program, and so we reject *Bounded_Queue_of_Strings* as an implementation of *Bag_of_Strings*.

Example A.6. Coin toss components.

Component *Retossable_Coin*	Component *Tail_Coin*
$values(Retossable\_Coin) =$ $\{Untossed, Edge, Head, Tail\}$ $init(Retossable\_Coin) = Untossed$	$values(Tail\_Coin) = NotTossed,\ OnTail$ $init(Tail\_Coin) = \{NotTossed\}$
(* Toss the coin and set C accordingly. Return $R{=}true$ if coin is on edge, else false *) Operation $Toss(C : self, R : Boolean) =$ $\{((@C, @R), (C, R)) :$ $\{(@C, C) : @C = Untossed \wedge ((C = Edge \wedge R = true) \wedge (C = Head \wedge R = false) \vee$ $(C = Tail \wedge R = false))\}$	(* Toss the coin and always land on tails *) Operation $Toss(C : self, R : Boolean) = \{(@C, C) :$ $@C = Untossed \wedge C = Tail \wedge R = false\}$
Operation $Fixit(C : self) = \{(@C, C) :$ $@C = Edge \wedge C = Untossed\}$	(* Operation *Fixit* is the null relation*) Operation $Fixit(C : self) = \emptyset$



Retossable_Coin has an operation that tosses a coin and includes an operation to permit it to be re-tossed if it lands on edge. *Tail_Coin*'s toss operation always gives tails. In this example, the *Fixit* operation for *Tail_Coin* is never enabled. This does not cause a problem in terms of whether *Tail_Coin* implements *Retossable_Coin*. To see this, consider any program designed for *Tail_Coin*. To call *Fixit*, the program would have had to take a branch based on the Boolean *true* result of a prior *Toss*. Since *Tail_Coin.Toss* always gives a result of *false*, that program branch will never be taken, as illustrated in the diagram.

Note that the service requirement for component implementation of *Retossable_Coin* by *Tail_Coin* is met.

Example A.7. Required extensions of *Retossable_Coin* with respect to *Tail_Coin* from Example A.6.

One extension is:

$$\ll Var(\#1), Toss(\#1, true) \gg$$

The following is not an extension:

$$\ll Var(\#1), Toss(\#1, true) \rightarrow true, Fixit(\#1) \gg$$

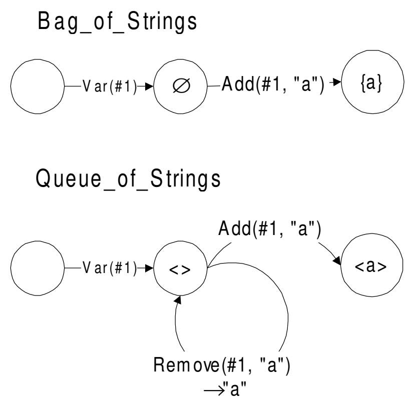
since

$$\ll Var(\#1), Toss(\#1, true) \rightarrow true \gg$$

is not a scenario of *Tail_Coin*.

Example A.8. Useless scenarios.

The figure shows selected scenarios of Example A.3.



The scenario $\ll Var(\#1), Remove(\#1, "a") \rightarrow "a" \gg$ is a useless scenario of *Queue_of_Strings* with respect to *Bag_of_Strings*.

Example A.9. Trace / scenario distinction vs. de Roever-Engelhardt's formulation.

We define scenarios as traces that, at every point, are enabled on all self values computed by the immediate predecessor. The formulation of de Roever and Engelhardt [4] makes no such distinction and so rejects component pairs that we include, as illustrated below.

The example given here is in the de Roever and Engelhardt formulation. Normal variables correspond to RESOLVE's indicator variables and representation variables correspond to RESOLVE's self variables. A program in their formulation is a composition of the C_i (respectively, A_i) operations, bracketed by CI (resp. AI) at the beginning to construct and initialize representation variables, and CF (resp. AF) at the end to destroy them. The operations are given as binary relations.

Define normal variable x over a, b, c and representation variable a over $\{1, 2\}$. Define data types $\mathcal{A}$ and $\mathcal{C}$ as follows.

$\mathcal{C}$	$\mathcal{A}$
$CI = \{a, (a, 1)), (a, (a, 2))\}$	$AI = \{(a, (a, 1)), (a, (a, 2))\}$
$C_1 = \{((a, 1), (b, 1))\}$	$A_1 = \emptyset$
$CF = \{((b, 1), b\}$	$AF = \{((b, 1), b)\}$

C_1 in $\mathcal{C}$ is defined on $(a, 1)$ but not on $(a, 2)$, while A_1 is null. So the concrete program $CI; C_1; C = \{a, b\}$ but the abstract program $AI; A_1; AF = \emptyset$. Since $CI; C_1; CF = \{a, b\} \not\subset AI; A_1; AF = \emptyset$, $\mathcal{A}$ does not refine $\mathcal{C}$.

Converting this to our approach, $\ll \mathit{Var}(\#1), C_1(\#1, a) \rightarrow b \gg$ is a trace but is not a scenario since C_1 is not enabled on all the states computable by $\ll \mathit{Var}(\#1) \gg$. So the scenarios of $\mathcal{C}$ are all of the form $\ll \mathit{Var}(\#1), \mathit{Var}(\#2), \ldots \gg$, which are identical with the scenarios of $\mathcal{A}$. This distinction reflects our view that since a program using C_1 will nondeterministically abort depending on the initialization of the self (representation) variable, the program should be rejected as ill-formed.

B Appendix: Theorem Proofs and Technical Material

Proposition B.1: Prefix-closure of Scenarios. If C is a component and σ is a scenario of C then any prefix σ' of σ is a scenario. Further, if N and B are components and σ is in $\mathit{scenarios}(N/B)$ then any prefix σ' of σ is in $\mathit{scenarios}(N/B)$.

Proof. This follows immediately from the definitions and a simple induction on the length of finite scenarios. □

Proposition B.2: Non-null computable states. If C is a component then the set of states computable by non-null finite trace σ of C is not null.

Proof. This follows by a simple induction on the length of σ since any non-null trace must start with a Var event whose values are not null. □

Lemma B.3: Computable Subset Under Correspondence. Let N and B be components and γ be a correspondence from N to B. If σ is a finite scenario of both N and B and s is computable by σ in N, then every u in $\gamma[s]$ is computable by σ in B.

Proof. By induction on length of σ. Base case: $\sigma = \lambda$. Let s be computable by λ in N. Since λ computes initial states in N, $s = \langle\rangle$. By the definition of correspondence, $\gamma[s] = \{\langle\rangle\}$. Since λ also computes initial states in B, $\langle\rangle$ is computable by λ in B. Consequently, every u in $\gamma[s]$ is computable by λ in B.

Inductive step: $\sigma = \sigma'; e$ where we assume that the lemma holds for σ'. By Proposition B.2, there is some state s' computable by σ' in N. Since scenarios are prefix-closed, σ' is a scenario of N and by the definition of realization convention, $s' \in \mathit{rc}$. We know that e is enabled on each $u' \in \gamma[s']$ in B since (a) by the inductive hypothesis, every $u' \in \gamma[s']$ is computable by σ' in B and (b) by the fact that since $\sigma'; e$ is a scenario of B, e is enabled on each state computable by σ' in B. The conditions for part (2) of the definition of correspondence are satisfied, so backward simulation (2b) holds: if there are s and u s.t. $s' \xrightarrow{e}_N s$ and $u \in \gamma[s]$ then there is $u' \in \gamma[s']$ such that $u' \xrightarrow{e}_B u$.

Since σ is a scenario of N, e is enabled on s', so there is an s s.t. $s' \xrightarrow{e}_N s$. Since s is computable by scenario σ in N, s is in rc. We know that $\gamma[s]$ is not null since γ is total on *rc*. Let u be any member of $\gamma[s]$. Then by the backward simulation condition of γ there is $u' \in \gamma[s']$ such that $u' \xrightarrow{e}_B u$. So every u in $\gamma[s]$ is computable by σ in B. □

Proof of Theorem 4.1: Soundness of the correspondence relation.

Behavior Requirement, finite case. Let N and B be components. Let γ be an image-finite correspondence relation from N to B. Let σ be a finite scenario in $\mathit{scenarios}(N/B)$ so that every extension prefix $\sigma''; v$ of σ is an extension of B. Show that $\sigma \in \mathit{scenarios}(B)$.

Proof. By induction on length of σ. Base case. $\sigma = \lambda$. Then $\lambda \in \mathit{scenarios}(B)$ by definition.

Inductive step. $\sigma = \sigma'; e$ where we assume as the inductive hypothesis that $\sigma' \in \mathit{scenarios}(B)$. Let $v = \mathit{inv}(e)$. Then since $\sigma \in \mathit{scenarios}(N/B)$ and $\sigma'; v$ is an extension prefix of σ, $\sigma'; v$ is an extension of B. By Proposition B.2, there is s' in *rc* computable by σ'. By Lemma B.3, every u' in $\gamma[s']$ is computable by σ' in B. Since $\sigma'; v$ is an extension of B, e is enabled on every state computable by σ' in B, so e is enabled on every u' in $\gamma[s']$. The condition for part (2) of the definition of correspondence are satisfied, so backward simulation (2b) holds: if there are s and u s.t. $s' \xrightarrow{e}_N s$ and $u \in \gamma[s]$ then there is $u' \in \gamma[s']$ such that $u' \xrightarrow{e}_B u$.

Let s' be some state computable by σ' in N. Since $\sigma = \sigma'; e$ is a scenario of N, there is s s.t. $s' \xrightarrow{e}_N s$. Since s is computable by scenario σ in N, s is in *rc*. Since s is in *rc* and γ is total on *rc*, there is $u \in \gamma[s]$. So by γ's backward simulation condition, there is $u' \in \gamma[s']$ such that $u' \xrightarrow{e}_B u$.

Since u' is computable by σ' in B and $u' \xrightarrow{e}_B u$, $\sigma'; e$ is a trace of B. Since $\sigma'; v$ is an extension of B and $v = \mathit{inv}(e)$, e is enabled on every state computable by σ' in B, so $\sigma'; e$ is a scenario of B. □

Behavior Requirement, infinite case. Let N and B be components. Let γ be an image-finite correspondence relation from N to B. Let σ be an infinite scenario in $\mathit{scenarios}(N/B)$ so that every prefix $\sigma''; v$ is an extension of B. Show that $\sigma \in \mathit{scenarios}(B)$.

Proof. The strategy is as follows. First we establish that every prefix of σ is in $\mathit{scenarios}(B)$. Then we show that σ is a trace of B. Then by the definition of scenario, since every prefix of σ is a scenario of B, σ is a scenario of B.

Let $\ll s_0 e_1 s_1 e_2 \ldots \gg$ be an execution of N whose trace is σ. For each n, $\ll e_1 e_2 \ldots e_{n-1} \gg \in \mathit{scenarios}(B)$ since (a) $\mathit{scenarios}(N/B)$ is prefix closed so for each n, $\ll e_1 e_2 \ldots e_{n-1} \gg \in \mathit{scenarios}(N/B)$ and (b) by the finite part of this proof, $\ll e_1 e_2 \ldots e_{n-1} \gg \in \mathit{scenarios}(B)$.

Now we show that σ is a trace of B. Let G be a digraph with nodes being all pairs (u, i) s.t. $(s_i, u) \in \gamma$, with an edge from (u', i') to (u, i) exactly when $i = i' + 1$ and $u' \xrightarrow{e}_B u$. We must show that G meets the conditions of König's lemma [10].

1. G is infinite since there are infinite s_i, each $s_i \in rc$ since each prefix of σ is a scenario of N, and for each s_i, $\gamma[s_i]$ is not null.

2. The roots of G are of the form $(u, 0)$ where $u \in \gamma[s_0]$. Since γ is image finite, $\{(u, 0) : u \in \gamma[s_0]\}$ is finite, so G has finite roots.
3. $\{(u, i) : u \in \gamma[s_i]\}$ is finite since γ is image-finite, so the edges from any (u', i') to $\{(u, i) : u \in \gamma[s_i]\}$ are finite. Consequently, G has finite out-degree.
4. Finally we show by induction on n that every node of G is reachable from a root. The base case is obvious. For the inductive step, assume all nodes $(u', n{-}1)$ are reachable from a root. Consider any node (u, n). Then $s_{n-1} \xrightarrow{e_n}_N s_n$. Since $\ll e_1 e_2 \dots e_n \gg$ is a scenario of B, e_n is enabled on all states computable by $\ll e_1 e_2 \dots e_{n-1} \gg$ in B and so, by Lemma B.3, e_n is enabled on all states in $\gamma[s_{n-1}]$. The conditions for part (2) of the definition of correspondence are satisfied, so backward simulation (2b) holds: if there are s and u s.t. $s' \xrightarrow{e}_N s$ and $u \in \gamma[s]$ then there is $u' \in \gamma[s']$ such that $u' \xrightarrow{e}_B u$.

Since the conditions of König's lemma are met, there is an infinite path a in G starting from some root. Now, every path of G is an execution of B of the form $\ll u_0 e_1 u_1 e_2 \dots \gg$, finite or infinite, because of the way G was constructed. So a is an infinite execution of B and $\mathit{trace}(a) = \ll e_1 e_2 \dots \gg = \sigma$. Since every prefix of σ is a scenario of B, σ is a scenario of B. $\square$

Service Conformity. Let N and B be components. Let γ be an image-finite correspondence relation from N to B. Let $d = \sigma'; v \in \mathit{extensions}(B/N)$. Show $d \in \mathit{extensions}(N)$.

Proof. Since σ' is a scenario of N, by Proposition B.2 there is s' computable by σ' in N. By Lemma B.3, each u' in $\gamma[s']$ is computable by σ' in B. Since $\sigma'; v \in \mathit{extensions}(B/N)$, there is an event b s.t. $\sigma'; b$ is a scenario of B where $v = \mathit{inv}(b)$ and σ' is a scenario of N. Since $\sigma'; b$ is a scenario of B, b is enabled on all states computable by σ' in B and so is enabled on each u' in $\gamma[s']$. The conditions for part (2) of the definition of correspondence are satisfied, so service simulation (2a) holds: b is enabled on s' in N. Since the choice of s' among the computable states of σ' in N was arbitrary, b is enabled on all states computable by σ' in N. By the definition of enablement, there is an event e s.t. $\sigma'; e$ is a scenario of N and $\mathit{inv}(e) = v$. Since $\mathit{inv}(e) = v = \mathit{inv}(b)$, $\sigma'; v$ is an extension of N. $\square$

Proposition B.4: N **has fin iff for all** $e \in \mathit{events}(N)$ **and for all** $@s \in \mathit{states}(N)$**,** $\{s : s' \xrightarrow{e}_N s\}$ **is finite..**

Proof. The proof is straightforward. $\square$

Definition: Matching component.

1. Let T be a concept. Then B is a matching component if the following hold.
   (a) $\mathit{values}(B) = \mathit{values}(T)$.
   (b) $\mathit{init}(B) = \{t : \exists @t : (\mathit{initialize.ensures}[@t/@y, t/y])\}$.
   (c) For each operation P, where we assume P has interface $(x_1 : \mathit{self}, x_2 : \mathit{self}, z : Q)$,

$$
\begin{aligned}
&B.P = \\
&\quad \{((@t_1, @t_2, @d), (t_1, t_2, d)) : \\
&\quad\quad T.P.requires[@t_1/y_1, @t_2/y_2, @d/z] \wedge \\
&\quad\quad T.P.ensures[@t_1/@y_1, @t_2/@y_2, @d/@z, t_1/y_1, t_2/y_2, d/z])\} \cup \\
&\quad \{((@t_1, @t_2, @d), \perp) : T.P.requires[@t_1/y_1, @t_2/y_2, @d/z]\}.
\end{aligned}
$$

2. Let R be a realization. Then N is a matching component if the following hold.
   (a) $values(N) = values(R)$.
   (b) $init(N) = R.init.R_f$.
   (c) For each operation $N.P$, where we assume $N.P$ has interface $(x_1 : self, x_2 : self, z : Q)$,

$$
\begin{aligned}
&N.P = \\
&\quad \{((@r_1, @r_2, @d), (r_1, r_2, d)) : \\
&\quad\quad (r_1, r_2, d) \in \mathcal{M}_{tot}[\![T.P]\!](@r_1, r_2, @d) \wedge \\
&\quad\quad abort \notin \mathcal{M}_{tot}[\![T.P]\!](@r_1, @r_2, @d)\} \cup \\
&\quad \{((@r_1, @r_2, @d), \perp) : \\
&\quad\quad \perp \in \mathcal{M}_{tot}[\![T.P]\!](@r_1, @r_2, @d) \wedge \\
&\quad\quad abort \notin \mathcal{M}_{tot}[\![T.P]\!](@r_1, @r_2, @d)\}.
\end{aligned}
$$

□

Proof of Theorem 4.2: Soundness of the pointwise correspondence relation.

Let T be a concept and R a realization. Suppose there is a pointwise correspondence A from R to T. If B matches T and N matches R, then N implements R.

Proof. We will show that there is a correspondence from N to B. Then by the soundness of the correspondence relation we may conclude that N implements B. Let A be the pointwise correspondence from R to T. Define realization convention rc as

$$rc = \{\langle r_1, r_2, \ldots, r_n\rangle : for 1 \leq i \leq N : (T.conv[r_i/x])\} \cup \perp.$$

Define $\gamma \subseteq states(N) \times states(B)$ as

$$
\begin{aligned}
&\gamma = \\
&\quad \{(\langle r_1, r_2, \ldots, r_n\rangle, \langle t_1, t_2, \ldots, t_n\rangle) : \\
&\quad\quad \langle r_1, r_2, \ldots, r_n\rangle \in rc \text{ and for } 1 \leq k \leq N : (t_k \in A[r_i])\} \cup \\
&\quad \{(\perp, \perp)\}.
\end{aligned}
$$

Note that since A is image-finite, γ is image-finite.

1. Show rc contains all the states computable in N by finite scenarios. Proof is by induction on the length of finite scenarios of N.

(a) Base case. Let $\sigma = \lambda$. Then σ computes $\{\langle\rangle\}$ in N, which is a subset of rc.

(b) Inductive step. Let $\sigma = \sigma'; e$ where we assume that all states computable by σ' in N are in rc. Since σ is a scenario of N, there are states $@s$ and s s.t. $@s \xrightarrow{e}_N s$. There are three cases.

Case i. $s = \bot$. Then $\bot$ is in rc by definition.

Case ii. $s \neq \bot$ and e is $Var(\#i)$.

$@s \in rc$ by assumption. By the definition of $@s \xrightarrow{Var(\#i)}_N s$, the size of $@s$ is $i-1$ and the size of s is i, for $1 \le k < i : (s^{(k)} = @s^{(k)}$, and $s^{(i)} \in init(N))$. Consequently, for $1 \le k < i : (T.conv[s^{(k)}/x])$ and it suffices to show that $T.conv[s^{(i)}/x]$. Since $s^{(i)} \in init(N)$, $s^{(i)} \in T.init.R_f$. So by condition (2a) of the definition of pointwise correspondence, $T.conv[s^{(i)}/x]$.

2. Show γ is total on rc.
Let $\langle r_1, r_2, \ldots, r_n\rangle \in rc$. Then for $1 \le k \le N : (T.conv[r_k/x])$ and so by condition (1) of the definition of pointwise correspondence, for $1 \le k \le N$: $(A[r_k] \neq \emptyset)$. So there are $t_1, t_2, \ldots, t_n$ s.t. for $1 \le k \le N : (t_k \in A[r_k])$ and $\langle t_1, t_2, \ldots, t_n\rangle \in \gamma[\langle r_1, r_2, \ldots, r_n\rangle]$.
3. Show that service simulation holds. Let $@s$ be a state of rc and let e be an event that is enabled on each $@u$ of $\gamma[@s]$ in B. Show that e is enabled on $@s$ in N. There are two cases.

Case i. e is $Var(\#i)$.

Let $@u \in \gamma[@s]$; such a value exists because $@s \in rc$ and γ is total on rc. Since $Var(\#i)$ is enabled on $@u$, there is u s.t. $@u \xrightarrow{Var(\#i)}_B u$ and the size of $@u$ is $i-1$. By the definition of γ, the size of $@u$ and of $@s$ are the same, so the size of $@s$ is $i-1$. Let s be s.t. for $1 \le k < i : (s^{(k)} = @s^{(k)})$ and $s^{(i)} \in init(N)$. Then $@s \xrightarrow{Var(\#i)}_N s$ and e is enabled on $@s$ in N.

Case ii. e is $P(\#i, \#j, @d) \to d$.

$$
\begin{aligned}
& \forall @u \in \gamma[@s] : (P(\#i, \#j, @d) \to d \text{ is enabled on } @u) \\
\Leftrightarrow\ & \text{(def of enablement)} \\
& \forall @u \in \gamma[@s] : \exists u \in states(B) : (@u \xrightarrow{P(\#i, Ij, @d)\to d}_B u) \\
\Leftrightarrow\ & \text{(def of transition)} \\
& \forall @u \in \gamma[@s] : \exists u \in states(B) : \\
& \quad ((@u^{(i)}, @u^{(j)}, @d), (u^{(i)}, u^{(j)}, d)) \in B.P \vee \\
& \quad (@u^{(i)}, @u^{(j)}, @d), \bot) \in B.P) \\
\Leftrightarrow\ & \text{(def of } \gamma) \\
& \forall @u^{(i)} \in A[@s^{(i)}] : \forall @u^{(j)} \in A[@s^{(j)}] : \exists u \in states(B) : \\
& \quad ((@u^{(i)}, @u^{(j)}, @d), (u^{(i)}, u^{(j)}, d)) \in B.P \vee \\
& \quad (@u^{(i)}, @u^{(j)}, @d), \bot) \in B.P)
\end{aligned}
$$

4. Show that backward simulation holds. Let $@s$ be a state of rc and let e be an event that is enabled on each $@u'$ of $\gamma[@s]$ in B. Suppose there is s and u s.t. $@s \xrightarrow{e}_N s$ and $u \in \gamma[s]$. Show that there is $@u \in \gamma[@s]$ s.t. $@u \xrightarrow{e}_B u$. We begin by establishing the proposition $r \in init(N) \Rightarrow A[r] \subseteq init(B)$.

$$
\begin{aligned}
& r \in \mathit{init}(N) \\
\Leftrightarrow\ & (\text{def of } \mathit{init}(N)) \\
& r \in T.\mathit{init}.R_f \\
\Rightarrow\ & (\text{condition (2b) of def of pointwise correspondence}) \\
& \forall t \in A[r] : \exists @t \in \mathit{values}(T) : (T.\mathit{init}.\mathit{ensures}[@t/@x, t/x]) \\
\Rightarrow\ & (\text{ def of } \mathit{init}(B)) \\
& \forall t \in A[r] : t \in \mathit{init}(B) \\
\Rightarrow\ & (\text{def of } \subseteq) \\
& A[r] \subseteq \mathit{init}(B)
\end{aligned}
$$

There two cases to consider.

Case i. e is $\mathit{Var}(\#i)$. Let $@u = \langle u^{(1)}, u^{(2)}, \ldots, u^{(i-1)} \rangle$.

By the definition of $@s \xrightarrow{Var(\#i)}_N s$, $@s$ and s differ only in that s is longer by one than $@s$ and $s^{(i)} \in \mathit{init}(N)$. So by the definition of γ, $@u \in \gamma[@s]$. Since $s^{(i)} \in \mathit{init}(N)$, by the proposition, $A[s^{(i)}] \subseteq \mathit{init}(B)$, and so by the definition of γ, $u^{(i)} \in \mathit{init}(B)$. Since u and $@u$ differ only in that u is one longer than $@u$ and $u^{(i)} \in \mathit{init}(B)$, then by the definition of transition, $@u \xrightarrow{Var(\#i)}_B u$.

Case ii. e is $P(\#i, \#j, @d) \to d$. This is further divided into two cases.

Case ii-a. $s = \bot$.

Since $u \in \gamma[s]$, by the definition of γ, $u = \bot$. Let $@u \in \gamma[@s]$. Since e is enabled on each $@u' \in \gamma[@s]$ in B, e is enabled on $@u$ in B. By the construction of $B.P$, if $(@t_1, @t_2, @d) \in \mathrm{dom} B.P$ then $\bot \in B.P[(@t_1, @t_2, @d)]$. So since $(@u^{(i)}, @u^{(j)}, @d) \in \mathrm{dom} B.P$, by the definition of transition, $@u \xrightarrow{e}_B \bot$.

Case ii-b. $s \neq \bot$.

Since $P(\#i, \#j, @d) \to d$ is enabled on each $@u' \in \gamma[@s]$, $(@u'^{(i)}, @u'^{(j)}, d) \in \mathrm{dom} B.P$ for each $@u' \in \gamma[@s]$ and so, by the definition of γ and $B.P$, $T.P.\mathit{requires}[@u'^{(i)}/x_1, @u'^{(j)}/x_2, @d/z]$ for each $@u'^{(i)} \in A[@s^{(i)}]$ and $@u'^{(j)} \in A[@s^{(j)}]$. Since $@s \xrightarrow{e}_N s$, by the definition of transition and of $N.P$, $(s^{(i)}, s^{(j)}, d) \in \mathcal{M}_{tot}[\![T.P]\!](@s^{(i)}, @s^{(j)}, @d)$ and $\mathit{abort} \notin \mathcal{M}_{tot}[\![T.P]\!](@s^{(i)}, @s^{(j)}, @d)$. Since $u^{(i)} \in A[s^{(i)}]$ and $u^{(j)} \in A[s^{(j)}]$, the premise of condition (3c) of the definition of pointwise correspondence is met and we have that

$$
\begin{aligned}
& \exists (@t_1, @t_2, @d) \in T_i : \\
& \quad (@t_1 \in A[@s^{(i)}] \wedge @t_2 \in A[@s^{(j)}] \wedge \\
& \quad P.\mathit{ensures}[@t_1/@x_1, @t_2/@x_2, @d/@z, u^{(i)}/x_1, u^{(i)}/x_2, d/z])
\end{aligned}
$$

Let $@t_1$ and $@t_1$ be values that meet this condition. Define $@u$ s.t. for $1 \leq k \leq N \wedge k \neq i \wedge k \neq j$, where N is the size of $@u$. $@u^{(k)} = u^{(k)}$ and $@u^{(i)} = @t_1$ and $@u^{(j)} = @t_1$. We must show that $@u \in \gamma[@s]$ and that $@u \xrightarrow{P(\#i,\#j,@d)\rightarrow d}_B u$.

By the definition of transition, $@s \xrightarrow{P(\#i,\#j,@d)\rightarrow d}_N s$ implies that for $1 \leq k \leq N \wedge k \neq i \wedge k \neq j : (@s^{(k)} = s^{(k)})$. So since for $1 \leq k \leq N : (u^{(k)} \in s^{(k)})$, for $1 \leq k \leq N \wedge k \neq i \wedge k \neq j : (@u^{(k)} \in @s^{(k)})$. Since $@t_1 \in A[@s^{(i)}] \wedge @t_2 \in A[@s^{(j)}]$, we have that $@u^{(i)} \in A[@s^{(i)}] \wedge @u^{(j)} \in A[@s^{(j)}]$. So by the definition of γ, $@u \in \gamma[@s]$.

Since $T.P.\mathit{requires}[@u'^{(i)}/x_1, @u'^{(j)}/x_2, @d/z]$ for each $@u'^{(i)} \in A[@s^{(i)}]$ and $@u'^{(j)} \in A[@s^{(j)}]$, we have that $T.P.\mathit{requires}[@u^{(i)}/x_1, @u^{(j)}/x_2, @d/z]$. By condition (3c) of the definition of pointwise correspondence we have that

$$P.\mathit{ensures}[@u^{(i)}/@x_1, @u^{(j)}/@x_2, @d/@z, u^{(i)}/x_1, u^{(i)}/x_2, d/z].$$

Consequently, by the definition of $B.P$,

$$((@u^{(i)}, @u^{(j)}, @d), (u^{(i)}, u^{(j)}, d)) \in B.P$$

and by the definitions of event and enablement,

$$@u \xrightarrow{P(\#i,\#j,@d)\rightarrow d}_B u.$$

□

Technical material for proof of Theorem 4.3, completeness of the correspondence relation. We recall various definitions from [10] and then prove a lemma that establishes an equivalence between history followed by backward simulations on the one hand, and forward followed by backward simulations on the other. In what follows, we will assume that A, B and C are automata in the sense of [10].

- A has trace-inclusion with B iff $\mathit{traces}(A) \subseteq \mathit{traces}(B)$. In this case we write $A \leq_T B$.
- A refinement from A to B is a function r from states of A to states of B that satisfies (1) if $s \in \mathit{start}(A)$ then $r(s) \in \mathit{start}(B)$ and (2) if $s' \xrightarrow{a}_A s$ then $r(s') \xrightarrow{a}_B r(s)$. In this case we write $A \leq_R B$. This definition and the others have been simplified somewhat since we do not have hidden actions.
- A forward simulation from A to B is a relation f over states of A and states of B that satisfies (1) if $s \in \mathit{start}(A)$ then $f[s] \cap \mathit{start}(B) \neq \emptyset$ and (2) if $s' \xrightarrow{a}_A s$ and $u' \in f[s']$ then there exists a state $u \in f[s]$ s.t. $u' \xrightarrow{a}_B u$. In this case we write $A \leq_F B$.
- A backward simulation from A to B is a relation b over states of A and states of B that satisfies (1) b is total; (2) if $s \in \mathit{start}(A)$ then $b[s] \subseteq \mathit{start}(B)$ and

(3) if $s' \xrightarrow{a}_A s$ and $u \in b[s]$ then there exists a state $u' \in b[s']$ such that $u' \xrightarrow{a}_B u$. In this case we write $A \leq_B B$. If B also has fin then we write $A \leq_{iB} B$, where the backward simulation is image-finite.
- A history simulation from A to B is a relation h over states of A and states of B that satisfies (1) h is a forward simulation from A to B and (2) h^{-1} is a refinement from B to A. In this case we write $A \leq_H B$. A history simulation is conservative in that the traces of A and B are identical.

If B has fin, Lynch and Vaandrager have shown that $A \leq_T B \Leftrightarrow \exists C s.t. A \leq_F C \leq_{iB} B$; that is, there is a sound and complete verification method based on forward and image-finite backward simulations. We show in the following lemma that this equivalence still holds if we substitute a history simulation for the forward simulation. References to LV in the proofs refer to results in Lynch and Vaandrager [10].

Proposition B.5: Equivalence of iB and RiB simulations.. Proposition B.5: If A and B are automata and B has fin then $A \leq_{iB} B \Leftrightarrow \exists C s.t. A \leq_R C \leq_{iB} B$.

Proof. $\Rightarrow$. Let $C = A$. From LV we know that $A \leq_R A$ and we are given that $A \leq_{iB} B$.

$\Leftarrow$. From LV we know that $A \leq_R C \Rightarrow A \leq_{iB} C$ and that $A \leq_{iB} C$ and $C \leq_{iB} B \Rightarrow A \leq_{iB} B$. □

Lemma B.6: Equivalence of T and HiB simulations.. If A and B are automata and B has fin then $A \leq_T B \Leftrightarrow \exists C s.t. A \leq_H C \leq_{iB} B$.

Proof.

$$\begin{aligned}
& A \leq_T B \\
\Leftrightarrow\ & \text{(by LV)} \\
& \exists D : A \leq_F D \leq_{iB} B \\
\Leftrightarrow\ & \text{(by LV since } A \leq_F D \Leftrightarrow \exists C : A \leq_H C \leq_R D) \\
& \exists C : \exists D : A \leq_H C \leq_R D \leq_{iB} B \\
\Leftrightarrow\ & \text{(by Proposition B.5)} \\
& \exists C : A \leq_H C \leq_{iB} B \square
\end{aligned}$$

Definition B.7: Scenario Automaton.. The scenario automaton $A_{N/B}$ of N with respect to B is:

- $states(A_{N/B}) = \{(s, \sigma) : s \in states(N), \sigma \in scenarios(N/B)$ and s is computable by σ in $N\}$
- $init(A_{N/B}) = \{(\langle\rangle, \lambda)\}$
- $acts(A_{N/B}) = events(N)$

- $step(A_{N/B}) = \{(s', \sigma') \xrightarrow{e}_N (s, \sigma) : (s', \sigma') and (s, \sigma) \in states(N),\ s' \xrightarrow{e}_N s \in step(N)$, and $\sigma = \sigma'; e\}$

The scenario automaton A_B of B is:

- $states(A_B) = \{(s, \sigma) : s \in states(B),\ \sigma \in scenarios(B)$ and s is computable by σ in $B\}$
- $init(A_B) = \{(\langle\rangle, \lambda)\}$
- $acts(A_B) = events(B)$
- $step(A_B) = \{(u', \sigma') \xrightarrow{e}_B (u, \sigma) : (u', \sigma')$ and $(u, \sigma) \in states(A_B),\ u' \xrightarrow{e}_B u \in step(B)$, and $\sigma = \sigma'; e\}$ □

Proposition B.8: Equivalence of scenario automata equivalent with matching component. The traces of $A_{N/B}$ and A_B, respectively, are identical with $scenarios(N/B)$ and $scenarios(B)$, respectively.

Proof. This is immediate from the construction of $A_{N/B}$ and A_B, respectively. □

Proof of Theorem 4.3, completeness of the correspondence relation. Define $\gamma \subseteq states(N^a) \times states(B)$ as follows:

$\gamma[(s, \sigma)] = \{u : u$ is computable by σ in $B\}$, if σ is a scenario of B.
$\gamma[(s, \sigma)] = \perp$, otherwise.

Totality, Initial State Subsetting and Service Simulation. Let N and B be components and let B have fin. Let γ be as given above, N^a be the adjunction of N and rc^a be the states computable by scenarios of N^a. Assume N implements B so that $scenarios(N/B) \subseteq scenarios(B)$ and $extensions(B/N) \subseteq extensions(N)$. Show that the following hold:

1. $rc^a \subseteq \mathrm{dom}\gamma$.
2. $\gamma[(\langle\rangle, \lambda)] \subseteq \langle\rangle$.
3. For each state (s', σ') in rc^a and for each event e that is enabled on each u' of $\gamma[s']$, e is enabled on (s', σ') in N^a.

Proof. For (1), the conclusion is immediate from the definition of γ.

For (2), $\gamma[(\langle\rangle, \lambda)] = \{u : u$ is computable by λ in $B\} = \langle\rangle$.

For (3), assume that $(s', \sigma') \in rc^a$ and some event e is enabled on all $u' \in \gamma[(s', \sigma')]$ in B. We must show that e is enabled on (s', σ') in N^a.

Let $u' \in \gamma[(s', \sigma')]$. $u' \neq \perp$ since e is enabled on u' in B. So by the definition of γ, σ' is a scenario of B. We are given that e is enabled on all $u' \in \gamma[(s', \sigma')]$ in B and that $\gamma[(s', \sigma')] = \{u : u$ is computable by σ in $B\}$, so e is enabled on all u' computable by σ' in B, and consequently $\sigma'; v$ is an extension of B where $v = inv(e)$.

σ' is a scenario of N^a since (s', σ') is in rc^a. σ' is a scenario of N since every scenario of N^a is a scenario of N. Since σ' is a scenario of N and $\sigma'; v$ is an extension of B, $\sigma'; v$ is in $extensions(B/N)$. Since by assumption $extensions(B/N) \subseteq$

extensions(N), $\sigma'; v$ is in *extensions*(N). So for some event b, $\sigma'; b$ is a scenario of N where $v = inv(b)$.

Since (s', σ') is a state of N^a and for some s, $s' \xrightarrow{b}_N s$, there is a transition of N^a, so b is enabled on (s', σ') in N^a. Since $inv(b) = v = inv(e)$, e is enabled on (s', σ') in N^a. □

Commutativity. We begin with a discussion of the proof of commutativity. Our strategy is to construct automata whose traces are exactly the scenarios of the concrete and abstract components, respectively. Then we employ the results of Lynch and Vaandrager [10] to construct a backward relation between a history-enhanced version of the concrete and abstract automata. The correspondence relation constructed above is similar to but stronger than the backward relation. The properties of the backward relation are used to prove the commutativity result for the correspondence.

Let N and B be components and let B have fin. Let γ be the proposed correspondence given above, N^a be the adjunction of N and rc^a be the states computable by scenarios of N^a. Let $A_{N/B}$ and A_B be the scenario automata given above. Assume N implements B so that *scenarios*$(N/B) \subseteq$ *scenarios*(B) and *extensions*$(B/N) \subseteq$ *extensions*(N). Show that for each state (s', σ') in rc^a and for each event e that is enabled on each u' of $\gamma[(s', \sigma')]$, if there are (s, σ) and u s.t. $(s', \sigma') \xrightarrow{e}_{N^a} (s, \sigma)$ and $u \in \gamma[(s, \sigma)]$ then there is $u' \in \gamma[(s', \sigma')]$ such that $u' \xrightarrow{e}_B u$.

Proof. Since B has fin, Proposition B.4 gives us that A_B must also have fin else for some event e of B and some state u', there would be an infinite number of states u s.t. $u' \xrightarrow{e}_B u$.

Next we establish that $A_{N/B} \leq_T A_B$. This is equivalent to showing that *traces*$(A_{N/B}) \subseteq$ *traces*(A_B).

$\sigma \in traces(A_{N/B})$
$\Leftrightarrow$ (by Proposition B.8)
$\sigma \in scenarios(B/N)$
$\Rightarrow$ (assumption, *scenarios*$(N/B) \subseteq$ *scenarios*(B))
$\sigma \in scenarios(B)$
$\Leftrightarrow$ (by Proposition B.8)
$\sigma \in traces(A_B)$

Since $A_{N/B} \leq_T A_B$ and A_B has fin, by AL there is $A^h_{N/B}$ s.t. $A_{N/B} \leq_H A^h_{N/B} \leq_{iB} A_B$. By inspection of the proof constructions in AL, $A^h_{N/B}$ is given by:

- $states(A^h_{N/B}) = \{(s, \sigma, \sigma) : (s, \sigma) \in states(A_{N/B})\}$
- $init(A^h_{N/B}) = \{(s, \lambda, \lambda) \in states(A^h_{N/B})\}$
- $acts(A^h_{N/B}) = acts(A_{N/B})$
- $step(A^h_{N/B}) = \{(s', \sigma', \sigma') \xrightarrow{a}_{A^h_{N/B}} (s, \sigma, \sigma) : (s', \sigma') \xrightarrow{a}_{A_N} (s, \sigma) \in step(A_{N/B})\}$

The proof constructions in AL also show that the relation $back \subseteq states(A^h_{N/B}) \times states(A_B)$ is defined as $back[(s, \sigma, \sigma)] = \{(u, \sigma) : (u, \sigma)$ is computable by σ in A_B and is an image-finite backward simulation from $A^h_{N/B}$ to A_B.

By assumption, if (s', σ') in rc^a and event e is enabled on each u' of $\gamma[(s', \sigma')]$ then if there are (s, σ) and u s.t. $(s', \sigma') \xrightarrow{e}_{N^a} (s, \sigma)$ and $u \in \gamma[(s, \sigma)]$ then there is $u' \in \gamma[(s', \sigma')]$ such that $u' \xrightarrow{e}_B u$. Assume (s', σ') in rc^a and e is enabled on each u' of $\gamma[(s', \sigma')]$ in B. Assume further that there is (s, σ) s.t. $(s', \sigma') \xrightarrow{e}_{N^a} (s, \sigma)$ and that $u \in \gamma[(s, \sigma)]$. We must show that there is $u' \in \gamma[(s', \sigma')]$ such that $u' \xrightarrow{e}_B u$.

Since e is enabled on each u' of $\gamma[(s', \sigma')]$ in B, σ' must be a scenario of B; if not, $\gamma[(s', \sigma')] = \{\perp\}$ but no action is enabled on $\perp$. So $\sigma'; v$ is an extension of B, where $v = inv(e)$. Since $(s', \sigma') \xrightarrow{e}_{N^a} (s, \sigma)$, σ is a scenario of N. Since σ is a scenario of N and $\sigma'; v$ is an extension of B, $\sigma \in scenarios(N/B)$. Consequently (s', σ', σ') and (s, σ, σ) are states of $A^h_{N/B}$, and $(s', \sigma', \sigma') \xrightarrow{e}_{A^h_{N/B}} (s, \sigma, \sigma)$. There is a u s.t. $(u, \sigma) \in back[(s, \sigma, \sigma)]$ since $back$ is total on states of $A^h_{N/B}$. So since $back$ is a backwards relation from $A^h_{N/B}$ to A_B, there is $(u', \sigma') \in back[(s', \sigma', \sigma')]$ s.t. $(u', \sigma') \xrightarrow{e}_B (u, \sigma)$. Then

$(u', \sigma') \in back[(s', \sigma', \sigma')]$
$\Leftrightarrow$ (by definition of $back$)
(u', σ') is computable by $\sigma' in A_B$
$\Leftrightarrow$ (by definition of A_B)
u' is computable by σ' in B
$\Leftrightarrow$ (by definition of γ)
$u' \in \gamma[(s', \sigma')]$.

Further,

$(u, sigma') \xrightarrow{e}_{A_B} (u, \sigma)$
$\Leftrightarrow$ (by construction of A_B
$u' \xrightarrow{e}_B u$
$\Rightarrow$ (since $u' \in \gamma[(s', \sigma')]$
there is $u' \in \gamma[(s', \sigma')]$ s.t. $u' \xrightarrow{e}_B u$.

$\square$