Specification Matching of Software Components

AMY MOORMANN ZAREMSKI
Xerox Corporation
and
JEANNETTE M. WING
Carnegie Mellon University

Specification matching is a way to compare two software components, based on descriptions of the component's behaviors. In the context of software reuse and library retrieval, it can help determine whether one component can be substituted for another or how one can be modified to fit the requirements of the other. In the context of object-oriented programming, it can help determine when one type is a behavioral subtype of another. We use formal specifications to describe the behavior of software components and, hence, to determine whether two components match. We give precise definitions of not just exact match, but, more relevantly, various flavors of relaxed match. These definitions capture the notions of generalization, specialization, and substitutability of software components. Since our formal specifications are pre- and postconditions written as predicates in first-order logic, we rely on theorem proving to determine match and mismatch. We give examples from our implementation of specification matching using the Larch Prover.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.2 [**Software Engineering**]: Tools and Techniques—*software libraries*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*modules, packages; procedures, functions, and subroutines*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and

This research was sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant number F33615-93-1-1330. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Wright Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. This manuscript is submitted for publication with the understanding that the U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes. This article is a revised and extended version of a paper presented at the 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering, October 1995, and is based on work done while the first author was at Carnegie Mellon University.
Authors' addresses: A. M. Zaremski, The Wilson Center for Research and Technology, Xerox Corporation, 800 Phillips Road, 128-51E, Webster, NY 14580; email: zaremski@wrc.xerox.com; J. M. Wing, School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213; email: wing@cs.cmu.edu.

Verifying and Reasoning about Programs—*pre- and post-conditions; specification techniques*; H.3.3 [**Information Storage and Retrieval**]: Information Storage and Retrieval—*retrieval models*; *selection process*

General Terms: Documentation, Standardization, Verification

Additional Key Words and Phrases: Component retrieval, pre-/postconditions, signature matching, software components, software reuse, specification, specification matching, subtyping, theorem proving

1. MOTIVATION AND INTRODUCTION

Specification matching is a process of determining if two software components are related. It underlies understanding this seemingly diverse set of questions:

—*Retrieval*: How can I retrieve a component from a software library based on its semantics, rather than on its syntactic structure?
—*Reuse*: How might I adapt a component from a software library to fit the needs of a given subsystem?
—*Substitution*: When can I replace one software component with another without affecting the observable behavior of the entire system?
—*Subtype*: When is an object of one type a subtype of another?

In retrieval, we search for all library components that satisfy a given query. In reuse, we adapt a component to fit its environmental constraints, based on how well the component meets our requirements. In substitution, we expect the behavior of one component to be observably equivalent to the other's; a special case of substitution is when a subtype object is the component substituting for the supertype object. Common to answering these questions is deciding when one component *matches* another, where *matches* generically stands for "satisfies," "meets," or "is equivalent to." Common to these kinds of matches is the need to characterize the dynamic behavior, i.e., semantics, of each software component.

It is rarely the case that we would want one component to match the other "exactly." In retrieval, we want a close match; as in other information retrieval contexts [Mauldin and Leavitt 1994; Olsen et al. 1993; Salton and McGill 1983], we might be willing to sacrifice precision for recall. That is, we would be willing to get some false positives as long as we do not miss any (or too many) true positives. In determining substitutability, we do not need the substituting component to have the exact same behavior as the substituted, only the same behavior relative to the environment that contains it.

In this article, we lay down a foundation for different kinds of semantic matches. We explore not just *exact match* between components, but many flavors of *relaxed match*. To be concrete and to narrow the focus of what *match* could mean, we make the following assumptions:

—The software components in which we are interested are *functions* (e.g., C routines, Ada procedures, ML functions) and *modules* (roughly speaking, sets of functions) written in some programming language. These components might typically be stored in a program library, a shared directory of files, or a software repository.

—Associated with each component, C, is a signature, C_{sig}, and a specification of its behavior, C_{spec}.

Whereas signatures describe a component's type information (which is usually statically checkable), specifications describe the component's dynamic behavior. Specifications more precisely characterize the semantics of a component, rather than just its signature. In this article, our specifications are formal, i.e., written in a formally defined assertion language.

Given two components, $C = \langle C_{sig}, C_{spec} \rangle$ and $C' = \langle C'_{sig}, C'_{spec} \rangle$, we define a generic component match predicate, *Match*:

Definition 1 (*Component Match*).

$$\mathit{Match}\text{: Component, Component} \rightarrow \text{Bool.}$$

$$\mathit{Match}(C, C') = \mathit{match}_{sig}(C_{sig}, C'_{sig}) \wedge \mathit{match}_{spec}(C_{spec}, C'_{spec})$$

Two components C and C' *match* if (1) their signatures match, given some definition of signature matching, and (2) their specifications match, given some definition of specification match. Although we define match as a conjunction, we can think of signature match as a "filter" that eliminates the obvious nonmatches before trying the more expensive specification match.

There are many possible definitions for the signature match predicate, match_{sig}, which we thoroughly analyzed in a previous article [Zaremski and Wing 1995]. For simplicity, we use the most basic signature match definitions in the remainder of this article. For match_{sig} of functions, we use type equivalence modulo variable renaming ("exact match" in Zaremski and Wing [1995]), and for match_{sig} of modules, we use a partial mapping of functions in the modules with exact signature match on the functions ("generalized module match" in Zaremski and Wing [1995]).

In this article, we focus on the specification match predicate, match_{spec}. We write pre-/postcondition specifications for each function, where assertions are expressed in a first-order predicate logic. We determine a match between two functions by some logical relationship, e.g., implication, between the two pre-/postcondition specifications. We modularly define match between two modules in terms of some kind of match between corresponding functions in the modules. Given our choice of formal specifications, we exploit state-of-the-art theorem-proving technology as a way to implement a specification match engine. All of the example matches in this article have been proven using the Larch Prover (LP) [Garland and Guttag 1991].

Specification match goes a step beyond signature match. For functions, signature match is based entirely on the functions' types, e.g., $\mathit{int} * \mathit{int} \rightarrow$

int, and not at all on their behavior. For example, integer addition and subtraction both have the same signature, but completely opposite behavior; the C library routines *strcpy* and *strcat* have the same signature, but users would be unhappy if one were substituted for the other. Given a large software library or a large software system, many functions will have identical signatures, but very different behavior. For example, in the C math library nearly two-thirds of the functions (31 out of 47) have the signature *double* $\rightarrow$ *double*. Based on signature match alone, we cannot know which of a large number of retrieved functions does what we want. Since specification match takes into consideration more knowledge about the components, it allows us to increase the precision with which we determine when two components match.

For each kind of match we define, there is both a match name and a match predicate symbol. For example, the strongest function specification match is named *exact pre/post match* and has the predicate symbol $match_{E\text{-}pre/post}$. For each match named M with the predicate symbol $match_M$ and components S and Q, if $match_M$ (S, Q) holds, we say equivalently that

—M match of S with Q,

—S matches with Q (under M), and

—Q is matched by S (under M).

It is important to distinguish between "matches with" and "is matched by," because not all matches are symmetric: $match_M$ (S, Q) does not necessarily imply that $match_M$ (Q, S). For the matches that are symmetric, we also say that "S and Q satisfy the match."

In what follows, we first briefly describe the language with which we write our formal specifications. We define exact and relaxed match for functions (Section 3) and then for modules (Section 4). We discuss our implementation of a specification matcher using LP in Section 5 and two applications of specification match in the software engineering context in Section 6. We close with related work and a summary.

2. LARCH/ML SPECIFICATIONS

We use Larch/ML [Wing et al. 1993], a Larch interface language for the ML programming language, to specify ML functions and ML modules. Larch provides a "two-tiered" approach to specification [Guttag and Horning 1993]. In one tier, the specifier writes *traits* in the Larch Shared Language (LSL) to assert state-independent properties. Each trait introduces *sorts* and *operators* and defines equality between terms composed of the operators (and variables of the appropriate sorts). Appendix A shows the *OrderedContainer* trait. Ordered containers are multisets that maintain an ordering on elements based on time of insertion (i.e., there is a notion of a first and last element). Elements are also ordered by a total order, $>$, on their values, e.g., integral values. Counter to the Larch style of using different traits for different theories, we chose to use the single trait

```
signature Stack = sig
  (*+ using OrderedContainer +*)
  type α t (*+ based on
    OrderedContainer.E OrderedContainer.C +*)

  val create : unit → α t
  (*+ create ( ) = s
    ensures s = empty +*)

  val push : α t ∗ α → α t
  (*+ push (s,e) = s2
    ensures s2 = insert(e, s) +*)

  val pop : α t → α t
  (*+ pop s = s2
    requires not(isEmpty(s))
    ensures s2 = butLast(s) +*)

  val top : α t → α
  (*+ top s = e
    requires not(isEmpty(s))
    ensures e = last(s) +*)
end
```

```
signature Queue = sig
  (*+ using OrderedContainer +*)
  type α t (*+ based on
    OrderedContainer.E OrderedContainer.C +*)

  val create : unit → α t
  (*+ create ( ) = q
    ensures q = empty +*)

  val enq : α t ∗ α → α t
  (*+ enq (q,e) = q2
    ensures q2 = insert(e,q) +*)

  val rest : α t → α t
  (*+ rest q = q2
    requires not(isEmpty(q))
    ensures q2 = butFirst(q) +*)

  val deq : α t → α
  (*+ deq q = e
    requires not(isEmpty(q))
    ensures e = first(q) +*)
end
```

Fig. 1. Two Larch/ML specifications.

OrderedContainer in multiple ways in order to simplify the explanations of our examples. The trait defines operators to generate containers (*empty* and *insert*), to return the container resulting from deleting a particular element (*delete*), to return the element or container resulting from deleting the first, last, or maximum element of a container according to the total ordering on elements (*first*, *last*, *max*, *butFirst*, *butLast*, and *butMax*), and to return information about a container (*size*, *isEmpty*) or information about a particular element (*isIn*, *count*).

In the second tier, the specifier writes *interfaces* in a Larch interface language to describe state-dependent effects of a program (see Figure 1). The Larch/ML interface language extends ML by adding specification information in special comments delimited by $(* + \cdots + *)$. The **using** and **based on** clauses link interfaces to LSL traits by specifying a correspondence between (programming-language-specific) types and LSL sorts. For polymorphic sorts, there must be an associated sort for both the polymorphic variable (e.g., α) and the type constructor (e.g., T) in the **based on** clause. The specification for each function begins with a *call pattern* consisting of the function name followed by a pattern for each parameter, optionally followed by an equal sign (=) and a pattern for the result. In ML, patterns are used in binding constructs to associate names to parts of values (e.g., (x, y) names x as the first of a pair and y as the second of a pair). The **requires** clause specifies the function's precondition as a predicate in terms of trait operators and names introduced by the call pattern. Similarly, the **ensures** clause specifies the function's postcondition. If a function does not have an explicit **requires** clause, the default is **requires** *true*. A function specification may also include a **modifies** clause, which lists those objects whose values may change as a result of executing the function. Larch/ML also includes rudimentary support for specifying higher-order functions.

Though the Larch/ML interface specifications of Figure 1 are simplistic, for exposition purposes, we will use them as the "library" for our examples of specification matching. It contains two module specifications: one for *Stack* with the functions *create*, *push*, *pop*, and *top*; and one for *Queue*, with the functions *create*, *enq*, *rest*, and *deq*. We specify each function's pre- and postconditions in terms of operators from the *OrderedContainer* trait (shown in Appendix A).

3. FUNCTION MATCHING

For a function specification, S, we denote the pre- and postconditions as S_{pre} and S_{post}, respectively. S_{pred} defines the interpretation of the function's specification as an implication between the two: $S_{pred} = S_{pre} \Rightarrow S_{post}$. This interpretation means that if S_{pre} holds when the function specified by S is called, then S_{post} will hold after the function has executed (assuming the function terminates). If S_{pre} does not hold, there are no guarantees about the behavior of the function. This interpretation of a pre- and postcondition specification is the most common and natural for functions in a standard programming model. For example, for the Stack *top* function in Figure 1

—the precondition top_{pre} is $not(isEmpty(s))$,

—the postcondition top_{post} is $e = last(s)$, and

—the specification predicate top_{pred} is $(not(isEmpty(s))) \Rightarrow (e = last(s))$.

To be consistent in terminology with our signature matching work, we present function specification matching in the context of a retrieval application. Example matches are between a library specification S and a query specification Q. We assume that variables in S and Q have been renamed consistently.[1] For example, if we compare the Stack *pop* function with the Queue *rest* function, we must rename q to s and $q2$ to $s2$. The examples presented in this section are intended primarily as illustrations of the various match definitions. Additional examples of more practical applications appear in Section 6. In this section, we examine several definitions of the specification match predicate ($match_{spec}(S, Q)$). We characterize definitions as either grouping preconditions S_{pre} and Q_{pre} together and postconditions S_{post} and Q_{post} together, or relating predicates S_{pred} and Q_{pred}. Both of these kinds of matches have a general form.

Definition 2 (*Generic Pre/Post Match*).

$$match_{pre/post}(S, Q) = (Q_{pre} \mathcal{R}_1 S_{pre}) \wedge (\hat{S} \mathcal{R}_2 Q_{post})$$

Pre/post matches relate the preconditions of each component and the postconditions of each component. Postconditions of related functions are often similar, so we want to compare them directly to each other. For example, postconditions may specify related properties of the return val-

[1]This renaming is easily provided by signature matching; we are assuming that the signatures of S and Q match.

Table I. Instantiations of Generic Pre/Post Match $((Q_{pre}\ \mathcal{R}_1\ S_{pre}) \wedge (\hat{S}\ \mathcal{R}_2\ Q_{post}))$

Match	Predicate Symbol	$\mathcal{R}_1$	$\mathcal{R}_2$	$\hat{S}$
Exact pre/post	$match_{E\text{-}pre/post}$	$\Leftrightarrow$	$\Leftrightarrow$	S_{post}
Plug-in	$match_{plug\text{-}in}$	$\Rightarrow$	$\Rightarrow$	S_{post}
Plug-in post	$match_{plug\text{-}in\text{-}post}$	$*$	$\Rightarrow$	S_{post}
Guarded plug-in	$match_{guarded\text{-}plug\text{-}in}$	$\Rightarrow$	$\Rightarrow$	$S_{pre} \wedge S_{post}$
Guarded post	$match_{guarded\text{-}post}$	$*$	$\Rightarrow$	$S_{pre} \wedge S_{post}$
		$*$: dropped		

ues. Similarly, preconditions of related functions may specify related bounds conditions of input values. In some cases, we may want to include some information about the precondition in the postcondition clause. In order to allow this flexibility, we let $\hat{S}$ be either S_{post} or $S_{pre} \wedge S_{post}$ in the generic pre/post match definition. The relations $\mathcal{R}_1$ and $\mathcal{R}_2$ relate preconditions and postconditions, respectively, and they are either equivalence ($\Leftrightarrow$) or implication ($\Rightarrow$), but need not be the same. The matches may vary from this form by dropping some of the terms. Table I summarizes how $\mathcal{R}_1$, $\mathcal{R}_2$, and $\hat{S}$ are instantiated for each of the pre/post matches in Section 3.1. For example, for plug-in match, $\mathcal{R}_1$ and $\mathcal{R}_2$ are both $\Rightarrow$, and $\hat{S}$ is S_{post}, so $match_{plug\text{-}in}$ is $(Q_{pre} \Rightarrow S_{pre}) \wedge (S_{post} \Rightarrow Q_{post})$. For $match_{plug\text{-}in\text{-}post}$ and $match_{guarded\text{-}post}$, $\mathcal{R}_1$ is not instantiated because its arguments are dropped. For $match_{guarded\text{-}plug\text{-}in}$ and $match_{guarded\text{-}post}$, $\hat{S}$ is $S_{pre} \wedge S_{post}$. We do not consider the case where $\mathcal{R}_1$ and $\mathcal{R}_2$ are $\Leftrightarrow$ and $\hat{S}$ is $S_{pre} \wedge S_{post}$, since including S_{pre} in $\hat{S}$ is only useful if $\mathcal{R}_2$ is $\Rightarrow$.

Definition 3 (*Generic Predicate Match*).

$$match_{pred}(S, Q) = S_{pred}\ \mathcal{R}\ Q_{pred}$$

Predicate matches relate the specification predicates, S_{pred} and Q_{pred}, in their entirety. Predicate matches are useful in cases where we need to consider the relationship of the specifications as a whole rather than the relationships of the parts, for example, when we need to assume something from the precondition in order to reason about postconditions. Additionally, these definitions apply for specifications of other forms (e.g., for specifications that do not have separate pre- and postconditions). The relation $\mathcal{R}$ between the specification predicates is equivalence ($\Leftrightarrow$) for the strictest match, but may be relaxed to either implication ($\Rightarrow$) or reverse implication ($\Leftarrow$). Table II summarizes how $\mathcal{R}$ is instantiated for each of the predicate matches in Section 3.2.

It is important to look at both pre/post matches and predicate matches. Which kind of match is appropriate may depend on the context in which the match is being used or on the specifications being compared. We present the pre/post matches in Section 3.1 and the predicate matches in Section 3.2. For each, we present a notion of exact match as well as relaxed matches.

Table II. Instantiations of Generic Predicate Match ($S_{pred}\ \mathcal{R}\ Q_{pred}$)

Match	Predicate Symbol	$\mathcal{R}$
Exact predicate	$match_{E\text{-}pred}$	$\Leftrightarrow$
Generalized	$match_{gen\text{-}pred}$	$\Rightarrow$
Specialized	$match_{spcl\text{-}pred}$	$\Leftarrow$

3.1 Pre/Post Matches

Pre/post matches on specifications S and Q relate S_{pre} to Q_{pre} and S_{post} to Q_{post}. Each match is an instantiation of the generic pre/post match (Definition 2). We consider five kinds of pre/post matches, beginning with the strongest match and weakening the match by relaxing the relations $\mathcal{R}_1$ and $\mathcal{R}_2$ from $\Leftrightarrow$ to $\Rightarrow$, by adding S_{pre} to $\hat{S}$, or by dropping the precondition term. In each case, relaxing the match allows us to make comparisons between less closely related components, but weakens the guarantees about the relationship between the two components. For example, dropping the precondition term would allow us to relate components that have the same behavior for the subset of inputs that they handle, but that make different assumptions about which inputs are valid (e.g., routines on arrays with different bounds). However, since we are not comparing the preconditions at all, we cannot guarantee that the components are behaviorally equivalent for all inputs.

3.1.1 *Exact Pre/Post Match.* If exact pre/post match holds for two specifications, the components are essentially equivalent and thus completely interchangeable. Anywhere that one component is used, it could be replaced by the other with no change in observable behavior. Exact pre/post match instantiates both $\mathcal{R}_1$ and $\mathcal{R}_2$ to $\Leftrightarrow$ and $\hat{S}$ to S_{post} in the generic pre/post match of Definition 2; two function specifications satisfy the *exact pre/post match* if their preconditions are equivalent and their postconditions are equivalent.

Definition 4 (*Exact Pre/Post Match*).

$$match_{E\text{-}pre/post}(S, Q) = (Q_{pre} \Leftrightarrow S_{pre}) \wedge (S_{post} \Leftrightarrow Q_{post})$$

Exact pre/post match is a strict relation, yet two different-looking specifications can still satisfy the match. Consider, for example, the following query $Q1$, based on the *OrderedContainer* trait. $Q1$ specifies a function that returns an ordered container whose size is zero, one way of specifying a function to create a new ordered container:

signature $Q1$ = **sig** ($Q1$)
  (∗+ **using** OrderedContainer +∗)
  type $\alpha\ t$ (∗+ **based on** OrderedContainer.E OrderedContainer.C +∗)
  val $qCreate : unit \rightarrow \alpha\ t$
  (∗+ $qCreate$ ( ) = c

```
        ensures $size(c) = 0$ +*)
    end
```

Under exact pre/post match, $Q1$ is matched by both the Stack and Queue *create* functions of Figure 1. (The specifications of Stack and Queue *create* are identical except for the name of the return value.)

Let us look in more detail at how the Stack *create* specification matches with $Q1$. Let S be the specification for Stack *create*, and let $Q1$ be the query specification with c renamed to s. $S_{pre} = true$, and $S_{post} = (s = empty)$. $Q1_{pre} = true$, and $Q1_{post} = (size(s) = 0)$. Since both S_{pre} and $Q1_{pre}$ are *true*, showing $match_{E\text{-}pre/post}(S, Q1)$ reduces to proving $S_{post} \Leftrightarrow Q1_{post}$, or $(s = empty) \Leftrightarrow (size(s) = 0)$. The "if" case $((s = empty) \Rightarrow (size(s) = 0))$ follows immediately from the axioms in the *OrderedContainer* trait about *size*. Proving the "only-if" case $((size(s) = 0) \Rightarrow (s = empty))$ requires only basic knowledge about integers and the fact that for any ordered container, s, $size(s) \geq 0$, which is provable from the *OrderedContainer* trait.

3.1.2 *Plug-In Match.* Equivalence is a strong requirement. Sometimes a weaker match is "good enough." For *plug-in match*, we relax both $\mathcal{R}_1$ and $\mathcal{R}_2$ from $\Leftrightarrow$ to $\Rightarrow$ in the generic pre/post match and keep $\hat{S} = S_{post}$. Under plug-in match, Q is matched by any specification S whose precondition is weaker (to allow at least all of the conditions that Q allows) and whose postcondition is stronger (to provide a guarantee at least as strong as Q).

Definition 5 (*Plug-In Match*).

$$match_{plug\text{-}in}(S, Q) = (Q_{pre} \Rightarrow S_{pre}) \wedge (S_{post} \Rightarrow Q_{post})$$

Plug-in match[2] captures the notion of being able to "plug-in" S for Q, as illustrated in Figure 2. A specifier writes a query Q saying essentially,

I need a function such that if Q_{pre} holds before the function executes then Q_{post} holds after it executes (assuming the function terminates).

With plug-in match, if Q_{pre} holds (the assumption made by the specifier) then S_{pre} holds (because of the first conjunct of plug-in match). Since we interpret S to guarantee that $S_{pre} \Rightarrow S_{post}$, we can assume that S_{post} will hold after executing the plugged-in S. Finally, since $S_{post} \Rightarrow Q_{post}$ from the second conjunct of plug-in match, Q_{post} must hold, as the specifier desired. We say that S is behaviorally equivalent to Q, since we can plug in S for Q and have the same observable behavior, but this is not a true equivalence because it is not symmetric: we cannot necessarily plug in Q for S and get the same guarantees.

Consider the following query. $Q2$ is a fairly weak specification of an *add* function. It requires that an input container has less than 50 elements, and guarantees that the resulting container is one element larger than the input container:

[2]This is similar to the notion of behavioral subtyping in object-oriented types; we discuss this similarity in detail in Section 6.2.

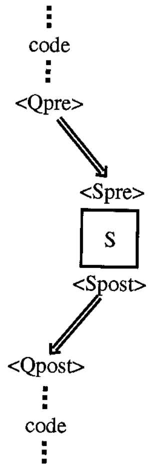

Fig. 2. Idea behind plug-in match.

```
signature Q2 = sig                                                     (Q2)
  (*+ using OrderedContainer +*)
  type α t (*+ based on OrderedContainer.E OrderedContainer.C +*)
  val add : α t * α → α t
  (*+ add (q1, e) = q2
    requires size(q1) < 50
    ensures size(q2) = (size(q1) + 1) +*)
end
```

Under exact pre/post match, $Q2$ is not matched by any function in the library, since no function in the library has a precondition equivalent to $Q2$'s. Under plug-in match, however, $Q2$ is matched by both the Stack *push* and the Queue *enq* functions. Since *push* and *enq* are identical except for their names and the names of the variables, the proof of the match is the same for both.

The precondition requirement, $Q_{pre} \Rightarrow S_{pre}$, holds, since $S_{pre} = true$. To show that $S_{post} \Rightarrow Q_{post}$, we assume S_{post} $(q2 = insert(e, q))$ and try to show Q_{post} $(size(q2) = size(q) + 1)$. Substituting for $q2$ in Q_{post}, we have $size(insert(e, q)) = size(q) + 1$, which follows immediately from the equations for *size*.

3.1.3 *Plug-In Postmatch.* Often, we are concerned with only the effects of functions; thus, a useful relaxation of the plug-in match is to consider only the postcondition part of the conjunction. Most preconditions could be satisfied by adding an additional check before calling the function. *Plug-in postmatch* is also an instance of generic pre/post match of Definition 2, with $\mathcal{R}_2$ instantiated to $\Rightarrow$ and $\hat{S}$ instantiated to S_{post} but dropping Q_{pre} and S_{pre}.

Definition 6 (*Plug-In Postmatch*).

$$match_{plug\text{-}in\text{-}post}(S, Q) = (S_{post} \Rightarrow Q_{post})$$

Consider the following query. $Q3$ is identical to Stack *top* except that $Q3$ has no **requires** clause:

```
signature Q3 = sig                                                    (Q3)
  (*+ using OrderedContainer +*)
  type α t (*+ based on OrderedContainer.E OrderedContainer.C +*)
  val delete : α t → α
  (*+ delete c = e
    ensures e = last(c) +*)
end
```

Stack *top* does not match with $Q3$ under either exact pre/post or plug-in match, because $Q3$'s precondition is weaker than Stack *top*'s precondition. Since the postconditions are equivalent, Stack *top* does match with $Q3$ under plug-in postmatch.

3.1.4 *Guarded Plug-In Match.* In some cases, the postcondition relation, $S_{post} \Rightarrow Q_{post}$, only holds for values of the input allowed by the precondition. For example, the *butLast* clause mentioned in the postcondition of Stack *pop* is not defined for the empty stack. The guarded plug-in match adds S_{pre} as an assumption (or "guard") to the postcondition relation, to exclude such cases. We instantiate $\mathcal{R}_1$ and $\mathcal{R}_2$ to $\Rightarrow$ in the generic pre/post match, as with plug-in match, but we use $\hat{S} = S_{pre} \wedge S_{post}$ rather than $\hat{S} = S_{post}$. We use S_{pre} and not Q_{pre}, since S_{pre} is likely to be necessary to limit the conditions under which we try to prove $S_{post} \Rightarrow Q_{post}$.

Definition 7 (*Guarded Plug-In Match*).

$$match_{guarded\text{-}plug\text{-}in}(S, Q) = (Q_{pre} \Rightarrow S_{pre}) \wedge ((S_{pre} \wedge S_{post}) \Rightarrow Q_{post})$$

For example, suppose we wish to find a function to delete from an ordered container using the following query $Q4$:

```
signature Q4 = sig                                                    (Q4)
  (*+ using OrderedContainer +*)
  type α t (*+ based on OrderedContainer.E OrderedContainer.C +*)
  val remainder : α t → α t
  (*+ remainder c = c2
    requires not(isEmpty(c))
    ensures size(c2) = (size(c) − 1) +*)
end
```

$Q4$ describes a function that requires a nonempty container and returns a container whose size is one less than the size of the input container. This is a fairly weak way of describing deletion, since it does not specify which element is removed. Even this weak specification match still gives us a big gain in precision over signature matching, however. $Q4$ would not be matched by other functions with the signature $\alpha\ t \rightarrow \alpha\ t$, for example, a

Assume $not(isEmpty(s))$	Assume S_{pre}	(1)
Assume $s2 = butLast(s)$	Assume S_{post}	(2)
$size(s2) = size(s) - 1$	Attempt to prove Q_{post}	(3)
$size(butLast(s)) = size(s) - 1$	Apply (2) to (3)	(4)
Let $s = insert(ec, sc)$	Since s is not empty (1), and s generated by empty and insert	(5)
$size(butLast(insert(ec, sc))) = size(insert(ec, sc)) - 1$	Substitute (5) for s in (4)	(6)
$size(sc) = size(insert(ec, sc)) - 1$	Axioms for $butLast$	(7)
$size(sc) = (size(sc) + 1) - 1$	Axioms for $size$	(8)
$size(sc) = size(sc)$	Axioms for $+, -$	(9)

Fig. 3. Proof sketch of second conjunct of $match_{guarded\text{-}plug\text{-}in}$ $(pop, Q4)$.

function that reverses or sorts the elements in the container or removes duplicates.

While, intuitively, $Q4$ would seem related to Stack *pop* and Queue *rest*, neither *pop* nor *rest* match with $Q4$ under either plug-in or plug-in postmatch. Consider Stack *pop* (the reasoning is similar for Queue *rest*). We cannot prove $S_{post} \Rightarrow Q_{post}$ (i.e., $(s2 = butLast(s)) \Rightarrow (size(s2) = size(s) - 1)$) for the case where $s = empty$. However, by adding the assumption S_{pre} $(not(isEmpty(s)))$, we are able to show that Stack *pop* matches with $Q4$ under guarded plug-in match. The first conjunct $(Q_{pre} \Rightarrow S_{pre})$ is trivial, since the preconditions of $Q4$ and Stack *pop* are the same. Figure 3 sketches the proof of the second conjunct $((S_{pre} \wedge S_{post}) \Rightarrow Q_{post})$.

3.1.5 *Guarded Postmatch.* As with plug-in match, we define a more relaxed guarded match by dropping the precondition relation term. Because we do not have the precondition term, there is no guarantee that S_{pre} actually holds, so we may have to provide an additional "wrapper" in our code to establish S_{pre} before we call the function specified by S.

Definition 8 (*Guarded Postmatch*).

$$match_{guarded\text{-}post}(S, Q) = (S_{pre} \wedge S_{post}) \Rightarrow Q_{post}$$

For example, consider the following query, which is the same as $Q4$ but without a **requires** clause:

```
signature Q5 = sig                                                  (Q5)
  (*+ using OrderedContainer +*)
  type α t (*+ based on OrderedContainer.E OrderedContainer.C +*)
  val remainder : α t → α t
  (*+ remainder c = c2
  ensures size(c2) = (size(c) − 1) +*)
end
```

Because this query has a stronger precondition, it is not matched by any functions in the library under either plug-in or guarded plug-in match. Plug-in postmatch does not work either because we need to assume S_{pre} $(not(isEmpty(s)))$ to show $S_{post} \Rightarrow Q_{post}$. However, under guarded postmatch, $Q5$ is matched by both Stack *pop* and Queue *rest*. The proofs are very similar to that for $Q4$ in the guarded plug-in match (Figure 3).

3.2 Predicate Matches

Recall the generic predicate match (Definition 3)

$$match_{pred}(S, Q) = S_{pred} \mathcal{R} Q_{pred},$$

where the relation $\mathcal{R}$ is either equivalence ($\Leftrightarrow$), implication ($\Rightarrow$), or reverse implication ($\Leftarrow$). Note that this general form allows alternative definitions of the specification predicates. One alternative is $S_{pred} = S_{pre} \wedge S_{post}$, which is stronger than $S_{pred} = S_{pre} \Rightarrow S_{post}$. This interpretation is reasonable in the context of state machines, where the precondition serves as a guard so that a state transition occurs only if the precondition holds. For our examples, we use $S_{pred} = S_{pre} \Rightarrow S_{post}$.

As we did with the generic pre/post match, we consider instantiations of the generic predicate match including an exact match and various relaxations.

3.2.1 *Exact Predicate Match.* We begin with *exact predicate match*. Two function specifications match exactly if their predicates are logically equivalent (i.e., $\mathcal{R}$ is instantiated to $\Leftrightarrow$). This is less strict than exact pre/post match (Definition 4), since there can be some interaction between the pre- and postconditions (i.e., $match_{E\text{-}pre/post} \Rightarrow match_{E\text{-}pred}$). In fact, in cases where $S_{pre} = Q_{pre} = true$, exact pre/post and exact predicate matches are equivalent.

Definition 9 (*Exact Predicate Match*).

$$match_{E\text{-}pred}(S, Q) = S_{pred} \Leftrightarrow Q_{pred}$$

Our example $Q1$ is still matched by Stack and Queue *create* under exact predicate match, since

$$S_{pred} \Leftrightarrow Q_{pred} = (true \Rightarrow (s = empty)) \Leftrightarrow (true \Rightarrow (size(s) = 0))$$

$$= (s = empty) \Leftrightarrow (size(s) = 0),$$

which is exactly what we proved to show that $Q1$ is matched by Stack and Queue *create* under exact pre/post match.

3.2.2 *Generalized Match.* Generalized match is an intuitive match in the context of queries and libraries: specifications of library functions will be detailed, describing the behavior of the functions completely, but queries can be simple. The query can focus on just the aspect of the behavior that we are most interested in or that we think is most likely to differentiate among functions in the library. Generalized match allows the library specification to be stronger (more general) than the query and, hence, allows for simple queries; $\mathcal{R}$ in the generic predicate match is instantiated to $\Rightarrow$. Generalized match is a weaker match than plug-in match (i.e., $match_{plug\text{-}in} \Rightarrow match_{gen\text{-}pred}$).

Definition 10 (*Generalized Match*).

$$match_{gen\text{-}pred}(S, Q) = S_{pred} \Rightarrow Q_{pred}$$

For example, again consider $Q4$. Using the exact predicate match, neither the Stack *pop* nor the Queue *rest* specifications match with this query. However, under generalized match, $Q4$ is matched by both of these. The proofs are very similar to that for $Q4$ in the guarded plug-in match (Figure 3).

Consider another example specifying a function that removes the most recently inserted element of an ordered container. This query does not require that the specifier knows the axiomatization of ordered containers, since the query uses only the container constructor, *insert*. The postcondition specifies that the input container, c, is the result of inserting the returned element, e, into another container $c2$. The existential quantifier (**there exists**) is a way of being able to name $c2$.

```
signature Q6 = sig                                                    (Q6)
  (*+ using OrderedContainer +*)
  type α t (*+ based on OrderedContainer.E OrderedContainer.C +*)
  val delete : α t → α
  (*+ delete c = e
    requires not(isEmpty(c))
    ensures there exists c2:OrderedContainer.C
      (c = insert(e, c2)) +*)
end
```

Under generalized match, the query is matched by the Stack *top* function, but not by Queue *deq*, since the query specifies that the most recently inserted element is returned. To show $match_{gen}(Stack.top, Q6)$, we consider two cases: $c = empty$ and $c = insert(ec, cc)$. In the first case, the preconditions for both *top* and *delete* are false, and thus, the match predicate is vacuously true. In the second case, the preconditions are both true, so we need to prove that $S_{post} \Rightarrow Q_{post}$. If we instantiate $c2$ to cc, the proof goes through.

$Q6$ retrieves Stack *top* not only under generalized match, but also under guarded plug-in and guarded postmatches using similar reasoning. $Q6$ does not retrieve anything under any of the other matches, because the other matches do not exclude the case of an empty stack by using the preconditions.

3.2.3 *Specialized Match.* Specialized match is the converse of generalized match: $match_{spcl\text{-}pred}(S, Q) = match_{gen\text{-}pred}(Q, S)$. A function whose specification is weaker than the query might still be of interest as a base

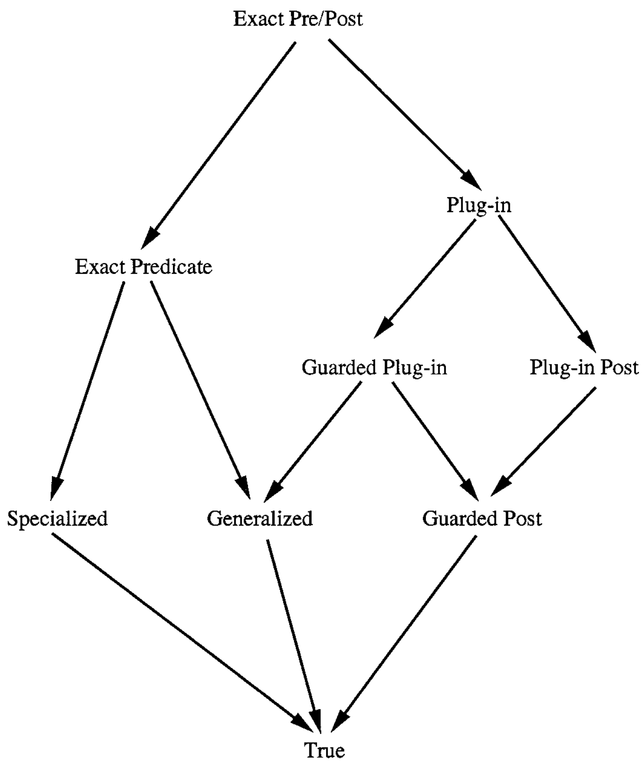

Fig. 4. Lattice of function specification matches.

from which to implement the desired function. Specialized match allows the library specification to be weaker than the query; we instantiate $\mathcal{R}$ in the generic predicate match to $\Leftarrow$.

Definition 11 (*Specialized Match*).

$$match_{spcl\text{-}pred}(S, Q) = Q_{pred} \Rightarrow S_{pred}$$

Consider again the query $Q3$, which is the same as Stack *top* but without the precondition. Stack *top* is thus weaker than $Q3$, but we can show that $Q3$ implies Stack *top* and, hence, that $Q3$ is matched by Stack *top* under specialized match.

3.3 Relating the Function Matches

We relate all of our function specification match definitions in a lattice (Figure 4). An arrow from a match $M1$ to another match $M2$ indicates that $M1$ is stronger than $M2$ (i.e., $M1(S, Q) \Rightarrow M2(S, Q)$ for all S, Q). We also say that $M2$ is more relaxed than $M1$.

Table III summarizes which of the library functions match each of the six example queries under each of the eight matches we have defined. For example, under generalized match, $Q4$ is matched by both *Queue.rest* and

Table III. Summary of which Functions Match which Queries (where Q = *Queue* module and S = *Stack* module)

Query	Exact Pre/Post	Exact Predicate	Plug-In	Guarded Plug-In	Plug-In Post	Specialized	Generalized	Guarded Post
$Q1$	*Q.create*	(*Q.create*)	(*Q.create*)	(*Q.create*)	(*Q.create*)	(*Q.create*)	(*Q.create*)	(*Q.create*)
	S.create	(*S.create*)	(*S.create*)	(*S.create*)	(*S.create*)	(*S.create*)	(*S.create*)	(*S.create*)
$Q2$	—	—	*Q.enq*	(*Q.enq*)	(*Q.enq*)	—	(*Q.enq*)	(*Q.enq*)
	—	—	*S.push*	(*S.push*)	—	(*S.push*)	(*S.push*)	
$Q3$	—	—	—	—	*S.top*	*S.top*	—	(*S.top*)
$Q4$	—	—	—	*Q.rest*	—	—	(*Q.rest*)	(*Q.rest*)
	—	—	—	*S.pop*	—	—	(*S.pop*)	(*S.pop*)
$Q5$	—	—	—	—	—	—	—	*Q.rest*
	—	—	—	—	—	—	—	*S.pop*
$Q6$	—	—	—	*S.top*	—	—	(*S.top*)	(*S.top*)

Stack.pop, but under plug-in postmatch, $Q4$ is not matched by any functions in the library. Parentheses around a function indicate that the match is implied by a stronger match (e.g., $match_{plug\text{-}in}(Q2,\ Queue.enq) \Rightarrow match_{guarded\text{-}plug\text{-}in}(Q2,\ Queue.enq)$).

We define a variety of matches. Which match is most appropriate to use will depend on the particular situation. First, the choice of match depends on the context in which the match is used: how strong of a guarantee is needed about the relation between the two specifications? If we want to know that we can substitute one function for the other and still have the same behavior, we would use plug-in match or an exact match. In contrast, if we are only interested in whether the functions have the same effects and are willing to check preconditions separately, we can use guarded postmatch. Which match is most appropriate also depends on the actual form of the predicates. In some cases, pre/post matches will be easier to prove with a theorem prover, since the pre/post matches relate preconditions to preconditions and postconditions to postconditions, and since for two specifications, S and Q, it is likely that S_{pre} and Q_{pre} are related; hence, we can reason about the relation (and similarly for S_{post} and Q_{post}). In other cases, however, it is necessary to make some assumptions about the precondition in order to prove a relation between the postconditions. In these cases, the predicate matches are easier to prove.

4. MODULE MATCHING

Function matching addresses the problem of matching individual functions. However, a programmer may need to compare collections of functions, for example, ones that provide a set of operations on an abstract data type. Modules, such as Ada packages or C++ classes, are a common language feature of most modern programming languages and are typically used to support explicitly the definition of abstract data types. Modules are also often used just to group a set of related functions, like I/O routines. This section addresses the problem of matching module specifications.

A *module specification interface* is a pair, $\Sigma = \langle \Sigma_T, \Sigma_F \rangle$, where

—Σ_T is a set of user-defined types and
—Σ_F is a set of function abstracts.

Σ_T introduces the names of user-defined type constructors that may appear in Σ_F. A function abstract is the function name together with the function specification. We include the function name both as useful feedback to the user and to distinguish between abstracts that would otherwise be the same (thus, Σ_F is a set rather than a multiset). For example, the *Queue* interface in Figure 1 has one user-defined type ($\Sigma_T = \{\alpha\ t\}$) and four function abstracts in Σ_F.

For a library interface, $\Sigma_L = \langle \Sigma_{LT}, \Sigma_{LF} \rangle$, to match a query interface, $\Sigma_Q = \langle \Sigma_{QT}, \Sigma_{QF} \rangle$, there must be correspondences both between Σ_{LT} and Σ_{QT} and between Σ_{LF} and Σ_{QF}.

In the module match definition we use here, the user-defined types and function abstracts in the query interface are a subset of those in the library interface. We consider other module match definitions elsewhere [Zaremski 1996]. We allow the query interface to be a subset of the library interface so that the querier may specify exactly the functions of interest and match a module that is more general in the sense that its set of functions may properly contain the query's set.

Definition 12 (*Module Match*).

$$M\text{-}match(\Sigma_L, \Sigma_Q, match_{fn}) = \exists \text{ total functions,}$$

$$U_{TC} : UserOp(\Sigma_{QT}) \rightarrow UserOp(\Sigma_{LT})$$

(with corresponding renaming TC), and

$$U_F : \Sigma_{QF} \rightarrow \Sigma_{LF}$$

such that

(1) U_{TC} and U_F are one-to-one;
(2) $\forall \tau \in \Sigma_{QT}$, $match_E(\tau, TC\ \tau)$; and
(3) $\forall Q \in \Sigma_{QF}$, $match_{fn}(U_F(Q), TC\ Q)$.

U_{TC} and TC ensure that user-defined types are named consistently in the two interfaces. For a set of user-defined types Σ_T, $UserOp(\Sigma_T)$ extracts the set of type constructor variables in Σ_T (e.g., for $\Sigma_T = \{\alpha\ T, int\ X\}$, $UserOp(\Sigma_T) = \{T, X\}$). The domain of function U_{TC} is a set of type constructor variables; from it we construct the type constructor renaming sequence TC, which is applied to the signature of each function specification in Σ_{QF}. For each $u_q \in UserOp(\Sigma_T)$, the renaming $[U_{TC}(u_q)/u_q]$ appears in TC. To avoid potential naming conflicts, we assume that $UserOp(\Sigma_{QT})$ and $UserOp(\Sigma_{LT})$ are disjoint (if they are not, we can easily make them so).

U_F maps each query function abstract Q to a corresponding library function abstract, $U_F(Q)$. Since any user-defined types in $U_F(Q)$ come from Σ_{LT}, we apply TC to Q to ensure consistent naming of type constructors. The correspondence between each $TC\ Q$ and $U_F(Q)$ is that they satisfy the function match, $match_{fn}$. The library module may contain more functions than the query module (i.e., $|\Sigma_{LF}| \geq |\Sigma_{QF}|$, and $\Sigma_{LF} \supseteq TC\ \Sigma_{QF}$, where $TC\ \Sigma_{QF}$ is a shorthand for applying TC to each element of Σ_{QF}). Section 6.2 contains an example of a module match, including a proof of the match relation with LP. Currently, the user must supply the mappings U_{TC} and U_F by hand.

Our definition of module match is highly parameterized and extensible. The function match relation between the pairs of functions is completely orthogonal to the module match definitions; we can instantiate $match_{fn}$ with any of the function specification matches defined in Section 3. In fact, the module match definitions are completely independent of the fact that we are matching specifications at the function level. If we use the same definitions of module matching, but instantiate $match_{fn}$ with a function signature match, we have module signature matching [Zaremski and Wing 1995].

Most generally, a module interface consists of some global information (Σ_T) and a set of functions (Σ_F). This framework allows the potential to extend the module interface to contain even more information. For example, we could extend module specification interfaces to include information about shared types or global invariants in Σ_T. A new module match definition including global invariants would be similar to Definition 12; however, U_{TC} would change, and point (2) of the definition would require some kind of consistency between invariants.

5. IMPLEMENTATION

We use LP, the Larch Prover [Garland and Guttag 1991], to attempt to prove that a match holds between two specifications. LP is a theorem prover for a subset of multisorted first-order logic. We implemented tools to translate Larch/ML specifications and to match predicates into LP input. Each of the specification match examples given in Section 3 (i.e., all entries in Table III) and in Section 6 have been specified in Larch/ML, translated automatically to LP input, and proven using LP.

For each specification file (e.g., Stack.sig), we check the syntax of the specification and then translate it into a form acceptable to LP. Namely, we generate a corresponding .lp file (e.g., Stack.lp), which includes the axioms from the appropriate LSL trait and contains the appropriate declarations of variables, operators, and assertions (axioms) for the pre- and postconditions of each function specified. Each function *foo* generates two operators, *fooPre* and *fooPost*; the axioms for *fooPre* and *fooPost* are the bodies of the **requires** and **ensures** clauses of *foo*. Figure 5 shows Stack.lp and Q2.lp, the result of translating the Stack specification from Figure 1 and the query $Q2$ into LP format. The **thaw** *OrderedContainer_Axioms* command

```
% Stack.lp
%% Using OrderedContainer
thaw OrderedContainer_Axioms
%% signature Stack
set name Stack
declare var
  e: E
  s: C
  s2: C
  ..

declare op
  createPre: ->Bool
  createPost: C ->Bool
  pushPre: ->Bool
  pushPost: C, E, C ->Bool
  popPre: C, C ->Bool
  popPost: C, C ->Bool
  topPre: C, E ->Bool
  topPost: C, E ->Bool
  ..

assert
  createPre = true;
  createPost(s) = (s = empty);
  pushPre = true;
  pushPost(s, e, s2) = (s2 = insert(e,s));
  popPre(s, s2) = (~(isEmpty(s)));
  popPost(s, s2) = (s2 = butLast(s));
  topPre(s, e) = (~(isEmpty(s)));
  topPost(s, e) = (e = last(s))
```

```
% Q2.lp
%% Using OrderedContainer
%%% thaw OrderedContainer_Axioms
%% signature Q2
set name Q2
declare var
  e: E
  q1: C
  q2: C
  ..

declare op
  addPre: C, E, C ->Bool
  addPost: C, E, C ->Bool
  ..

assert
  addPre(q1, e, q2) = (size(q1) < 50);
  addPost(q1, e, q2) =
    (size(q2) = size(q1) + 1)
  ..
```

Fig. 5. LP input for *Stack* and $Q2$.

loads the state resulting from executing the commands in OrderedContainer_Axioms.lp. We use the lsl tool to generate the file OrderedContainer_Axioms.lp from the LSL trait OrderedContainer.lsl. We comment out the **thaw** command in Q2.lp, since we assume that the query ($Q2$) uses the same trait as the library specification (*Stack*). The command **set name** $Q2$ tells LP to use $Q2$ as the prefix for names of facts and conjectures. Commands **declare var** and **declare op** declare variables and operators that will be used in the axioms. In particular, Q2.lp declares the element variable e, container variables $q1$ and $q2$, and operators *addPre* and *addPost*. The input sorts (C, E, C) correspond to the types of the input arguments and the result of *add* ($\alpha\ t * \alpha \rightarrow \alpha\ t$). The **assert** clause adds axioms to the logical system for *addPre* and *addPost*, corresponding to the **requires** and **ensures** clauses of *add*, respectively.

Given the names of two function specifications, their corresponding specification files, and which match definition to use, we also automatically generate the appropriate LP input to initiate an attempt to determine the match between those two functions. For example, Figure 6 shows the LP input to prove the plug-in match of Stack *push* with $Q2$. The input to LP for the proof consists simply of commands to load the theories for the library and query (**execute** *Stack* and **execute** $Q2$) and the proof statement (**prove** . . .).

```
% PlugIn-Q2-Stack.lp
%% Load library and query specs
execute Stack
execute Q2

%% Plug-in Match: (Qpre => Spre) /\ (Spost => Qpost)
prove (addPre(s, e, s2) => pushPre) /\ (pushPost(s, e, s2) => addPost(s, e, s2))
```

Fig. 6. LP input for plug-in match of *Stack.push* with *Q*2.

We could alternatively have chosen to generate the LP axioms on a per-query basis rather than generating axioms for each .sig file (i.e., given a particular pair of functions, generate only the necessary axioms for that particular pair). However, we assume that generating an .lp file from a .sig file will happen only once and that there may be several queries on a library specification or several match definitions for a particular query. This approach enables us to consider module-level matches as well.

Although we have defined specification matches that are either true or false, our implementation does not answer quite so succinctly. Since LP is designed as a proof assistant, rather than as an automatic theorem prover, some of the proofs require user assistance. Thus, a proof assertion submitted to LP yields the result that either a proof was found (i.e., the match is true) or a proof was not found (i.e., the match may or may not be true). In the second case, user assistance is required either to complete the proof or to determine that it cannot be completed. An extension to our implementation would be to also submit to the prover the negation of the match to directly prove that a match is false.

Each of the 40 entries in Table III corresponds to a match that we have used LP to prove. In characterizing how much assistance the proofs require, we consider only the *primary matches* (the 11 entries in the table that are not in parentheses), since proofs for all others follow automatically from an entry to the left in the same row. Table IV summarizes the level of user assistance required for the primary matches. "None" means the proof went through with no user assistance; "guidance" means that the proof required user input to apply the appropriate proof strategies; and "lemma" means that the user had to prove additional lemmas to complete the proof.

Four of the proofs needed no assistance from the user: plug-in match of *Stack.push* and *Queue.enq* with *Q*2 and plug-in postmatch and specialized match of *Stack.top* with *Q*3. Plug-in match of *Stack.push* with *Q*2 is the example shown in Figure 6; executing the statements in Figure 6 results in the response from LP that the match conjecture was proved using the default proof methods; no user assistance was required.

Guarded plug-in match of *Stack.top* with *Q*6 is an example of a match that requires some user assistance to LP. Figure 7 shows an LP-annotated script for this proof. The lines with boldface are user input; ⟨ ⟩ and [ ] are proof notes from LP; and % is the comment character. The additional user input **resume by induction** tells the prover to use the induction proof strategy. This generates a base case (the container, *c*, is empty) and an induction hypothesis (assumes the assertion is true for a container, *cc*, and

Table IV. Level of User Assistance Required for LP Proofs of Queries

Query	Library	Match	User Assistance
$Q1$	*Queue.create*	Exact pre/post	Lemma
$Q1$	*Stack.create*	Exact pre/post	Lemma
$Q2$	*Queue.enq*	Plug-in	None
$Q2$	*Stack.push*	Plug-in	None
$Q3$	*Stack.top*	Specialized	None
$Q3$	*Stack.top*	Plug-in post	None
$Q4$	*Queue.rest*	Guarded plug-in	Lemma
$Q4$	*Stack.pop*	Guarded plug-in	Guidance
$Q5$	*Queue.rest*	Guarded post	Lemma
$Q5$	*Stack.pop*	Guarded post	Guidance
$Q6$	*Stack.top*	Guarded plug-in	Guidance

```
% exec M-Guard-Q6-Stack
%% Load library and query specs
execute Stack
execute Q6

%% Guarded Plug-in Match: (Qpre => Spre) /\ ((Spre /\ Spost) => Qpost)
prove (deletePre(c, e) => topPre(c, e)) /\ ((topPre(c, e) /\ topPost(c, e)) => deletePost(c, e))
  % Additional user input
  resume by induction
    <> basis subgoal
    [ ] basis subgoal
    <> induction subgoal
    resume by specializing c2 to cc
      <> specialization subgoal
      [ ] specialization subgoal
    [ ] induction subgoal
  [ ] conjecture
  %% End of input from file 'Guard-Q6-Stack.lp'.
```

Fig. 7. Proof script of guarded plug-in match of *Stack.top* with $Q6$.

attempts to prove the assertion for the container *insert*(e, cc)). The prover is able to prove the basis subgoal automatically, but needs further assistance with the induction subgoal. The prover is able to simplify the induction subgoal to $\exists\ c2 : insert(e, cc) = insert(e, c2)$. The user input **resume by specializing** $c2$ **to** cc tells the prover to use cc as the value for the existential variable $c2$, and the proof then goes through. The line [ ] conjecture indicates that LP completed the proof. We classify the user assistance for this proof as simply *guidance*, telling LP what proof strategy to use next in cases where the default strategies do not complete the proof. A total of three proofs require guidance: guarded plug-in match of *Stack.top* with $Q6$, guarded plug-in match of *Stack.pop* with $Q4$, and guarded postmatch of *Stack.pop* with $Q5$.

The remainder of the proofs (exact pre/post match of *Queue.create* and *Stack.create* with $Q1$, guarded plug-in match of *Queue.rest* with $Q4$, and guarded postmatch of *Queue.rest* with $Q5$) required not only guidance but also additional lemmas in order to prove the match. In all four cases, one of the additional lemmas is $\sim(insert(e, q) = empty)$ (something that might

```
% exec M-GuardPost-Q5-Queue
%% Load library and query specs
execute Queue
execute Q5

set name Lemma
prove ~(insert(e,q) = empty) by contradiction
      <> contradiction subgoal
      critical-pair *Hyp with OrderedContainer
      [ ] contradiction subgoal
   [ ] conjecture

prove size(butFirst(insert(e,q))) = size(q) by induction on q
      <> basis subgoal
      [ ] basis subgoal
      <> induction subgoal
      [ ] induction subgoal
   [ ] conjecture

set name Query
prove restPre(q, q2) /\ restPost(q, q2) => remainderPost(q, q2)
   resume by induction on q
      <> basis subgoal
      [ ] basis subgoal
      <> induction subgoal
      [ ] induction subgoal
   [ ] conjecture
   %% End of input from file 'M-GuardPost-Q5-Queue.lp'.
```

Fig. 8. Proof script of guarded postmatch of *Queue.rest* with $Q5$.

reasonably be included in a more complete theory of containers). The proofs for *Queue.rest* with $Q4$ and $Q5$ additionally need the lemma $size(butFirst(insert(e, q))) = size(q)$, which falls out directly from the axioms for *Stack* but not *Queue*. The proofs for $Q1$ need additional lemmas about the sizes of containers. Figure 8 shows an LP-annotated script for the proof of guarded postmatch of *Queue.rest* with $Q5$.

6. APPLICATIONS

As mentioned in Section 1, any problem that involves comparing the behavior of two software components is a potential candidate for specification matching. In particular, we focus on problems that center around substituting one component for another. In this section, we examine two such problems: retrieval for reuse and subtyping of object-oriented types.

6.1 Retrieval for Reuse

If we have a library of components with specifications, we can use specification matching to retrieve components from the library. Formally, we define the retrieval problem as follows:

Definition 13 (*Retrieval*).

Retrieve: Query Specification, Match Predicate, Component Library $\rightarrow$ Set of Components

$$Retrieve(Q,\ match_{spec},\ L) = \{C \in L : match_{spec}(C,\ Q)\}$$

Given a query specification Q, a specification match predicate $match_{spec}$, and a library of component specifications L, *Retrieve* returns the set of components in L that match with Q under the match predicate $match_{spec}$. Note that the components can be either functions or modules, provided that $match_{spec}$ is instantiated with the appropriate match. Parameterizing the definition by $match_{spec}$ also gives the user the flexibility to choose the degree of relaxation in the specification match. In practice, it is inefficient to perform the specification match on every component in a library. A practical implementation would use a more efficient match (e.g., signature match) as a filter, perhaps even allowing the user to specify on a case-by-case basis whether to attempt the specification match.

Using specification match as part of the retrieval process (or separately on a given pair of components) gives us assurances about how appropriate a component is for reuse. At the function level especially, the various specification matches give us various assurances about the behavior of a component we would like to use. We treat Q as the "standard" we expect a component to meet, and S as the library component we would like to reuse. If the exact pre/post match holds on S and Q, we know that S and Q are behaviorally equivalent under all conditions; using S for Q should be transparent. If the plug-in or guarded plug-in match holds, we know that S can be substituted for Q, and the behavior specified by Q will still hold, although we are not guaranteed the same behavior when Q_{pre} is false. If the guarded postmatch holds, we know that the specified behavior holds when S_{pre} is satisfied. Depending on the context, we may be able to ensure that S_{pre} holds and, hence, guarantee the behavior specified by Q.

For example, suppose that we are implementing a file cache manager. Among many other things, we will need a function to replace a file in the cache with a newly fetched file when the cache is full. We want to know whether there are functions in the library to do this. Given that library functions have specifications associated with them, we can use specification matching to retrieve the functions we want. If we use a match definition like guarded plug-in match, we can use a fairly weak specification like $Q7$ as our query:

```
signature Query = sig                                              (Q7)
  (*+ using OrderedContainer +*)
  type α fscache (*+ based on OrderedContainer.E OrderedCon-
  tainer.C +*)
  val qReplace : α fscache * α → unit
  (*+ qReplace (cache, file)
    requires size (cache) = 50
    modifies cache
    ensures isIn(file, cache') and (size(cache') = size(cache)) +*)
end
```

```
signature Component1 = sig
    (*+ using OrderedContainer +*)
    type α fscache (*+ based on OrderedContainer.E OrderedContainer.C +*)
    val replaceFirst : α fscache * α → unit
    (*+ replaceFirst (cache, file )
      requires not(isEmpty(cache))
      modifies cache
      ensures cache′ = insert(file, butFirst(cache)) +*)
end

signature Component2 = sig
    (*+ using OrderedContainer +*)
    type α fscache (*+ based on OrderedContainer.E OrderedContainer.C +*)
    val replaceMax : α fscache * α → unit
    (*+ replaceMax (cache, file )
      requires not(isEmpty(cache))
      modifies cache
      ensures cache′ = insert(file, delete(max(cache),cache)) +*)
end
```

Fig. 9. Two library file replacement functions.

$Q7$ specifies a property that would hold for a destructive replacement function, namely, that the size of the cache remains the same and that the new file is in the cache in the final state. The query function takes as input a file system cache (of type α *fscache*) and a file (of type α). The **requires** clause indicates that the cache must be a particular size (i.e., we are assuming that we are operating on a full cache). The **modifies** clause indicates that the value of *cache* may be changed by the function. In the **ensures** clauses, we use *cache'* to stand for the value of the cache in the final state and the unprimed *cache* to refer to the value in the initial state.

Suppose that the two functions listed in Figure 9 are in the library. Both require that the cache be nonempty, and replace a current element of the cache with the new file. The *replaceFirst* function in *Component1* uses a FIFO replacement strategy: the first file inserted is the one replaced (i.e., the file that has been in the cache the longest). The *replaceMax* function in *Component2* uses a priority-based replacement strategy: it replaces the maximum element in the cache, for some (unspecified) total ordering on the elements of the cache. This ordering could be based on the time since the file was last referenced (i.e., an LRU replacement strategy) or on the priority of the elements in the cache (e.g., hoard priorities).

Using guarded plug-in match, retrieval using the query $Q7$ returns both of the library functions in Figure 9, since both replacement strategies guarantee the properties specified in $Q7$'s postcondition. Proofs of guarded plug-in match of both *replaceFirst* with $Q7$ and *replaceMax* with $Q7$ are shown in Appendix B.

Thus, we could use both of these functions to experiment with the effects of a particular replacement strategy on the performance of our cache manager. We could also use a more specific query (e.g., the same as one of the library components) to distinguish between the two library components.

This example also illustrates the importance of the precondition guard in guarded plug-in match. If we used plug-in match rather than guarded plug-in match, we would not retrieve either function, since it is necessary to exclude the case of an empty cache when trying to prove that the sizes of *cache* and *cache′* are equal.

6.2 Subtyping

A second application of specification matching is determining when one object is a subtype of another. In object-oriented programming languages, an *object type*[3] defines a collection of *objects*, which consist of *data* (state) and *methods* that act on the data [America 1991; Cardelli 1989; Meyer 1988]. Intuitively, a type σ is a subtype of another type τ if an object of type σ can be substituted for an object of type τ. Precise definitions of subtyping vary in the strictness of this notion of substitutability from simply requiring the methods' signatures to match (*signature subtyping*) to requiring a correspondence between the methods' dynamic semantics (*behavioral subtyping*).

In order to relate subtyping to signature and specification matching, we must first convert object types to our context. We base our definition of an object type on that of Liskov and Wing [1994], but differ from their definition in that we do not include invariants or constraints. We restrict our focus here to relating methods, which is only one aspect of their subtyping relation. We model an object type as a module interface, with a type declaration for the object type (a description of the object type's value space), a global variable of the object type to hold the current state of the object (an element of the value space), and a function signature (and specification) for each method.

Let T represent the module interface of the supertype, and let S represent the module interface of the subtype. Subtyping requires a correspondence between each method in T and a method in S, but allows additional methods in S. The correspondence between methods varies among the subtype definitions but is always a function match definition. There is also a correspondence between type declarations. These are exactly the correspondences captured by the module match definition (Definition 12). Thus, we define subtyping in terms of module match using the following general form:

Definition 14 (*Generic Subtype*).

$$Subtype(S, T) = M\text{-}match(S, T, match_{method})$$

S is a subtype of T if their modules match. The particular notion of subtyping depends on $match_{method}$, the match used at the method (function) level. We discuss other possible instantiations of $match_{method}$ and the more general relation between both signature and behavioral subtyping to signature and specification matching in more detail elsewhere [Zaremski 1996].

[3]These are usually simply called "types," but we need to distinguish types of objects from types in signatures.

```
signature BagObj = sig
    (*+ using OrderedContainer +*)
    type α t (*+ based on
      OrderedContainer.E OrderedContainer.C +*)

    val b : α t

    val put : α → unit
    (*+ put (e)
      modifies b
      ensures b' = insert(e, b) +*)

    val get : unit → α
    (*+ get ( ) = e
      requires not(isEmpty(b))
      modifies b
      ensures (b' = delete(e, b)) and
              (isIn(e, b)) +*)

    val card : unit → int
    (*+ card ( ) = n
      ensures n = size(b) +*)
  end
```

```
signature StackObj = sig
    (*+ using OrderedContainer +*)
    type α t (*+ based on
      OrderedContainer.E OrderedContainer.C +*)

    val s : α t

    val push : α → unit
    (*+ push (e)
      modifies s
      ensures s' = insert(e, s) +*)

    val pop_top : unit → α
    (*+ pop_top ( ) = e
      requires not(isEmpty(s))
      modifies s
      ensures (s' = butLast(s)) and
              (e = last(s)) +*)

    val swap_top : α → unit
    (*+ swap_top (e)
      requires not(isEmpty(s))
      modifies s
      ensures s' = insert(e, butLast(s)) +*)

    val height : unit → int
    (*+ height ( ) = i
      ensures i = size(s) +*)
  end
```

Fig. 10. Larch/ML specifications of bag and stack object types.

In the remainder of this section, we relate behavioral subtyping to specification matching and illustrate how to use specification matching to show that one object is a behavioral subtype of another with an example.

Figure 10 shows the module specifications for two objects (an example similar to that in Liskov and Wing [1994]). The first is *BagObj*, a mutable bag object with a global variable *b* and methods *put*, *get*, and *card*. The second specification is of a stack object. *StackObj* is based on the same trait as bag, but has a stricter specification for the method that removes an object (*pop_top*) and has an additional method, *swap_top*. In keeping with the Liskov and Wing approach, we assume that create methods are defined elsewhere. Appendix A lists the *OrderedContainer* trait on which both specifications are based.

The *StackObj* specification differs in several ways from the *Stack* specification in Figure 1. First, in *StackObj*, stacks are mutable, whereas in *Stack*, they are not. Because the *Stack* specification in Figure 1 specifies the behavior of a typical implementation in a functional language, its stacks are immutable. Here, however, we wish to model the specification of a stack in the object-oriented paradigm, and hence, these stacks are mutable. Second, *Stack* has separate functions for *pop* and *top*, whereas *StackObj* combines these in *pop_top*. Again, this is mainly a by-product of the difference between a functional implementation and an object-oriented one. Third, each specification has additional functions that the other does not.

We now consider how to define the behavioral subtype relation between two objects (modules). Behavioral subtyping attempts to capture the notion that anywhere in a program that an object of type T is used we should be able to substitute an object of type S, where S is a subtype of T, and still have the same observable behavior of the program.

There are a number of definitions of behavioral subtyping that attempt to capture this substitutability property [America 1991; Dhara and Leavens 1992; 1996; Leavens 1989; Leavens and Weihl 1990; Liskov and Wing 1994; Meyer 1988]. There are subtle differences between all of these subtype definitions, but common to all is the use of pre/postcondition specifications both to describe the behavior of types and to determine whether one type is a subtype of another. Let m_T be a method of supertype T, and let m_S be the corresponding method of subtype S.

Behavioral subtyping requires that each method in the supertype T have a corresponding method in the subtype S, but there may be additional methods in S. We use the following rules for behavioral subtyping:

—*Precondition Rule*: $m_T.pre \Rightarrow m_S.pre$.
—*Postcondition Rule*: $(m_S.pre \wedge m_S.post) \Rightarrow m_T.post$.

This is the same as our guarded plug-in match and is used for the same reason: to show substitutability, making assumptions about the precondition when necessary. Thus, we define behavioral subtyping by instantiating $match_{method}$ in the generic subtype definition (Definition 14) with guarded plug-in match (Definition 7). We assume that the signatures match.

Definition 15 (*Behavioral Subtype*).

$$Subtype_{behav}(S, T) = M\text{-}match(S_{spec}, T_{spec}, match_{guarded\text{-}plug\text{-}in})$$

We can model other versions of behavioral subtyping by substituting other function specification definitions for $match_{method}$. For example, substituting plug-in match for $match_{method}$ yields America's [1991] subtype definition, which is also the methods rule in Liskov and Wing's [1994] subtype definition. Substituting a conjunction of generalized match with the precondition rule from plug-in match (i.e., $match_{method} = (m_T.pre \Rightarrow m_S.pre) \wedge (m_S.pred \Rightarrow m_T.pred)$) yields Dhara and Leaven's [1996] method rule.

Consider the *StackObj* and *BagObj* specifications in Figure 10. If we expect a bag object, we will not be surprised by the behavior of a stack object (i.e., we should be able to substitute a stack for a bag). Stack *push* adds an element to a container, just as bag *put* does, and stack *height* returns the size of a container, just as bag *card* does. Bag *get* is nondeterministic: it deletes and returns an element in a container. Stack *pop_top* is just more restrictive about which element it deletes. In contrast, if we expect a stack object, we may be surprised by a bag object when we remove an element, since the bag *get* method may remove an element other than the top element. Thus, intuitively we would expect stack to be a subtype of bag, but not vice versa. We would like to show that *StackObj* is a behavioral subtype of *BagObj* according to Definition 15. As the objects are specified, we would not be able to show the subtype relation if we used plug-in match as the method match, because we cannot prove $match_{plug\text{-}in}(pop\_top, get)$ (since we cannot reason about the case where the stack or bag is empty). However, we can show that *StackObj* is a behavioral subtype of *BagObj*, since our behavior subtype definition uses guarded plug-in match, which specifically allows us to exclude the case where the stack or bag is empty.

To show $Subtype_{behav}(StackObj, BagObj)$ (or, equivalently, $M\text{-}match(StackObj_{spec}, BagObj_{spec}, match_{guarded\text{-}plug\text{-}in})$), we must define the mappings U_F and U_{TC} to satisfy the three requirements of module match in Definition 12. There is only one user-defined type in both *StackObj* and *BagObj*, and it is the same (i.e., $UserOp(\Sigma_{BagT}) = UserOp(\Sigma_{StackT}) = t$). So U_{TC} is the identity function ($U_{TC}(t) = t$). We define U_F as follows: $U_F(put) = push$, $U_F(get) = pop\_top$, and $U_F(card) = height$. U_{TC} and U_F satisfy the three requirements of module match:

(1) U_{TC} and U_F are both one-to-one total functions (U_F is not onto, but does not need to be)

(2) $match_E(\alpha\ t,\ \alpha\ t)$

(3) $match_{guarded\text{-}plug\text{-}in}(push, put)$
$match_{guarded\text{-}plug\text{-}in}(pop\_top, get)$
$match_{guarded\text{-}plug\text{-}in}(height, card)$.

We translated our specifications of *StackObj* and *BagObj* into LP input and were able to prove the guarded plug-in matches with very little user guidance. Appendix C shows the LP proof script of guarded plug-in match between each pair of methods. The proofs for $match_{guarded\text{-}plug\text{-}in}(push, put)$ and $match_{guarded\text{-}plug\text{-}in}(height, card)$ are trivial, since the specifications are identical modulo variable names. The proof for $match_{guarded\text{-}plug\text{-}in}(pop\_top, get)$ requires an additional lemma and some guidance.

Thus, not only have we shown how subtyping fits into our framework of specification matching, but we can also use our specification matching tools to automate checking our subtype relation. Other subtype definitions (e.g., Liskov and Wing [1994]) include additional global information, such as invariants and constraints, which we do not model. It should be possible, however, to add this in our framework by extending Σ_T to include constraint specifications in addition to user-defined type declarations.

7. RELATED WORK

Other work on specification matching has focused on using one or two particular match definitions for retrieval of software components (usually functions). Rollins and Wing [1991] proposed the idea of function specification matching and implemented a prototype system in λProlog using plug-in match. λProlog does not use equational reasoning, and so the search may miss some functions that match a query but that require the use of equational reasoning to determine that they match. The VCR retrieval system [Fischer et al. 1994] uses plug-in match with VDM as the specification language. The focus of this work was on the efficiency of proving match; the tool performs a series of filtering steps before doing all-out match. Penix and Alexander [1995] used theorem proving to translate specifications automatically into domain-specific feature sets (sets of attribute-value pairs), which they then used to do a more efficient retrieval. Such an approach depends on formulating the feature sets for each domain, however. Perry's [1989] Inscape system is a specification-based software development environment. Its Inquire tool [Perry and Popovich 1993] provides predicate-based retrieval in Inscape. Match can be either exact pre/post match or a form of generalized match. The prototype system has a simplified and, hence, fairly limited inference mechanism. In Inscape, the user must provide specifications for each component anyway, so the query for a retrieval will already be written. Jeng and Cheng [1992] used order-sorted predicate logic specifications. They defined two matches, both of which are instances of our generalized function match, but with the additional property that they generate a series of substitutions to apply to the library component to reuse in the desired context. Mili et al. [1994] defined a specification as a binary relation. A specification S *refines* another specification Q if S has information about more inputs and assigns fewer images to each argument. This is like plug-in match except that the match is in terms of relations rather than predicates. Goguen et al. [1996] described retrieval of modules using a series of filtering matches that rank the closeness of each candidate component to the query. One of their matches is a form of module specification match. The module match measures the intersection of matching query and candidate functions (as opposed to requiring that the query functions be a subset of the candidate functions, as we do). The function match is a weak form of generalized match, checking only that ground equations in the query are satisfied by the candidate. An advantage of the match is that the user can specify the query by giving results of sample executions rather than by writing a formal specification.

The PARIS system [Katz et al. 1987] maintains a library of partially interpreted *schemata*. Each schema includes a specification of assertions about the input and results of the schema and about how the abstract parts of the schema can be instantiated. Matching corresponds to determining whether a partial library schema could be instantiated to satisfy a query. The system does some reasoning about the schemata but with a limited logic. Katoh et al. [1986] used "ordered linear resolution" to match English-like specifications that have been translated into first-order predicate logic formulas. They allow some relaxations, but check only for equivalence and do not verify that the subroutines match.

To summarize, our work on specification matching is more general than the above in three ways: (1) we handle not just function match, but also module match; (2) we have a framework, which is extremely modular (e.g., function match is a parameter to module match; specification match is one conjunct of component match), within which we can express each of the specific matches "hardwired" in the definitions used by others; and (3) we have a flexible prototype tool that lets us easily experiment with all of the different matches. Finally, we are not wedded to just the software retrieval application; we also apply specification match to other application areas.

Signature matching is a very restricted form of specification matching. Most work in this area has focused on using the expressiveness and theoretical properties of type systems to define various forms of relaxed matches [Di Cosmo 1992; Rittri 1990; Runciman and Toyn 1989; Stringer-Calvert 1994; Zaremski and Wing 1995]. Chen et al. [1993] described a framework for both signature and specification matching, but only implemented signature matching. Wileden et al. [1991] surveyed *specification-level interoperability*. Most work thus far has focused on signature-based interoperability and on how to convert types in a heterogeneous environment [Konstantas 1993; Thatté 1994; Yellin and Strom 1994].

Less closely related work, but relevant to our context of software library retrieval, is divided into three categories. Text-based information retrieval [Arnold and Stepoway 1987; Frakes and Nejmeh 1987; Maarek et al. 1991; Prieto-Díaz 1989] and AI-based semantic net classifications [Fischer et al. 1991; Ostertag et al. 1992] have the advantage that many efficient tools are available to do the search and match in these structures. The disadvantage is that a component's behavior is described informally. A third class of retrievals [Consens et al. 1992; Paul and Prakash 1994] allows queries over a representation of the component's actual code, for example, abstract syntax trees. Such queries are useful mainly for determining structural characteristics of a component, for example, nested loops or circular dependencies.

8. SUMMARY AND FUTURE WORK

The work described in this article makes three specific contributions with respect to specification matching: foundational definitions, a prototype tool, and descriptions of applications. By providing precise definitions, we lay the groundwork for understanding when two different software components are related, in particular, when their specifications match. Although we consider in detail functions and modules, exact and relaxed match, and formal pre/postcondition specifications, the general idea behind specification matching is to exploit as much information associated with the description of software components as possible. By building a working specification match engine, we have demonstrated the feasibility of our ideas. With this tool, we can explore the pragmatic implications of our definitions and apply specification matching to various applications. Although our notion of specification match was originally motivated by the software library retrieval application, it is more generally applicable to other areas of software engineering, e.g., determining subtyping in designing class hierarchies or showing that one component may be substituted for another when upgrading a system.

Specification matching is “expensive” in two respects: (1) the nontrivial overhead of writing the specifications and (2) the necessity of doing a proof to show a match. However, many of our match definitions (plug-in matches and generalized match) do not require a full specification of the query component, which somewhat mitigates the first expense. For the second expense, we can use theorem-proving technology to automate the match process rather than prove the match by hand. Additional advances in the field of theorem proving would improve this further. The overall expense is further mitigated by doing the match selectively. For example, in the retrieval domain, we use a less-expensive match, such as signature matching, to initially prune the search space, and then select one or two potential matches that we verify using specification matching.

Regardless of the cost, there are applications where other matches (e.g., text or signatures) are not descriptive enough and where we are willing to expend a little extra effort to specify and prove that a match holds. For example, if we want to replace a component in a safety-critical system with an updated, verified library component, we would want to prove that we could substitute the library component for the existing component.

Specification matching also has the potential to be extended to apply to other problems such as interoperability. The heart of an interoperability problem is that the interfaces of two or more systems do not match [Vernon et al. 1994]. Thus, our work makes a step in the direction of detecting an interoperability problem based on a system’s interface that specifies its input-output (black-box) functional behavior. However, even if two components’ specifications match according to our notion of interface specification, they may still fail to interoperate. One reason is that they may differ in the way they choose to communicate with their environment. One way to extend our work is to add more information to interface specifications to enable detection of other ways components interact with each other. Toward this goal, Allen and Garlan [1994] used a subset of CSP to specify “protocols” as a means to capture the way a component communicates with its environment and to determine when components interoperate smoothly with each other based on these protocol specifications. Hence, a more complete interface would include protocol specifications as well as our kind of functional specification; our notion of specification match could similarly be extended to include a notion of protocol match. We deliberately set up our framework to allow different notions of specification and different notions of specification match, depending on one’s personal definition of specification.

Finally, we can invert the notion of specification match: determining that two components do not match is determining that they mismatch. Garlan et al. [1995] took a step toward understanding this notion of mismatch at a system’s architectural level. Hence, a promising direction of future work is to extend our formal framework from the module level to the architectural level by modeling the various kinds of *architectural mismatch* they describe informally.

APPENDIX

A. THE *OrderedContainer* TRAIT

```
OrderedContainer (E, C) : trait
  includes Integer, TotalOrder(E)
  introduces

    empty: → C                butFirst: C → C
    insert: E, C → C          butLast: C → C
    delete: E, C → C          butMax: C → C
    first: C → E              isEmpty: C → Bool
    last: C → E               isIn: E, C → Bool
    max: C → E                size: C → Int
                              count: E, C → Int

  asserts
    C generated by empty, insert
    C partitioned by count
    ∀ e, e1 : E, c : C
      last(insert(e, c)) ==e
      butLast(insert(e, c)) == c
      first(insert(e, c)) == if c = empty then e else first(c)
      butFirst(insert(e, c)) == if c = empty then empty
                                 else insert(e, butFirst(c))
      max(insert(e, c)) == if c = empty then e
                           else if e > max(c) then e else max(c)
      butMax(insert(e, c)) == delete(max(c), c)
      isEmpty(empty)
      ¬isEmpty(insert(e, c))
      ¬isIn(e, empty)
      isIn(e, insert(e1, c)) == (e = e1) ∨ (isIn(e, c))
      size(empty) == 0
      size(insert(e, c)) == size(c) + 1
      size(delete(e, c)) == if isIn(e, c) then size(c) − 1 else size(c)
      count(e, empty) == 0
      count(e, insert(e1, c)) == count(e, c) + (if e = e1 then 1 else 0)
      count(e, delete(e1, c)) == if e = e1 then max(0, count(e, c) − 1)
                                 else count(e, c)
```

B. RETRIEVAL EXAMPLE PROOFS

```
%% Guarded plug-in match of replaceFirst with Q7

execute replace-fifo.lp
execute query7.lp

%% Guarded Plug-in match—precondition
prove (qReplacePre(cache, newobj) => replaceFirstPre(cache, newobj))
  resume by induction on cache
```

```
%% Additional Lemmas
set name Lemma
prove ~(insert(newobj,cache) = empty) by contradiction
  critical-pair *Hyp with Container

prove size(butFirst(insert(e,cache))) = size(cache) by induction on cache

%% Guarded Plug-in match—postcondition
set name Query
prove (replaceFirstPre(cache, newobj) /\ replaceFirstPost(cache, cache′,
        newobj)) =>
  qReplacePost(cache, cache′, newobj)
    resume by induction on cache

%% Full Guarded Plug-in match
prove (qReplacePre(cache, newobj) => replaceFirstPre(cache, newobj)) /\
        ((replaceFirstPre(cache, newobj) /\ replaceFirstPost(cache,
        cache′, newobj)) =>
        ((qReplacePost(cache, cache′, newobj))
qed

%% Guarded plug-in match of replaceMax with Q7

execute replace-priority.lp
execute query7.lp

%% Guarded Plug-in match—precondition
prove (qReplacePre(cache, newobj) => replaceMaxPre(cache, newobj))
  resume by induction on cache

%% Additional Lemmas
set name Lemma

prove ~(c=empty) => ~isEmpty(c)
  resume by induction on c

prove ~(isEmpty(c)) => isIn(max(c),c)
  resume by induction on c
    resume by case cc = empty
      resume by case max(cc) < e
        instantiate c by cc in Lemma.1

prove ~(isEmpty(c)) => size(insert(e, delete(max(c), c))) = size(c)
  resume by =>
    instantiate c by cc in Lemma.2

%% Guarded Plug-in match—postcondition
set name Query
prove (replaceMaxPre(cache, newobj) /\ replaceMaxPost(cache, cache′, ne-
        wobj)) =>
  qReplacePost(cache, cache′, newobj)

%% Full Guarded Plug-in match
prove (qReplacePre(cache, newobj) => replaceMaxPre(cache, newobj)) /\
        (replaceMaxPre(cache, newobj) ∧ replaceMaxPost(cache, cache′,
        newobj)) =>
        qReplacePost(cache, cache′, newobj)
qed
```

C. SCRIPT OF PROOF THAT STACK IS A BEHAVIORAL SUBTYPE OF BAG

```
execute bagobj.lp
execute stackobj.lp

% guarded-plug-in(push, put)
prove (putPre => pushPre) ∧ ((pushPre ∧ pushPost(b, b′, e)) =>
       putPost(b, b′, e))
  [ ] conjecture

% guarded-plug-in(height, card)
prove (cardPre => heightPre) ∧ ((heightPre ∧ heightPost(b, i)) =>
       cardPost(b, i))
  [ ] conjecture

% Additional lemma
assert 0 <= count(e,s)
prove delete(e,insert(e,s)) = s
  apply OrderedContainer.2 to conjecture
  [ ] conjecture

% guarded-plug-in(pop_top, get)
prove
  (getPre(b, e) => pop_topPre(b, e)) ∧
  ((pop_topPre(b,e) ∧ pop_topPost(b, b′, e)) => getPost(b, b′,e))
  . .
  resume by induction on b
    ⟨ ⟩ basis subgoal
    [ ] basis subgoal
    ⟨ ⟩ induction subgoal
    [ ] induction subgoal
  [ ] conjecture
qed
```

ACKNOWLEDGMENTS

We thank David Garlan and Stephen Garland for comments on earlier versions of this work, and we also thank Stephen Garland for his assistance in the details of using LP. The cache replacement strategy example was initially suggested by Maria Ebling. We also thank the anonymous referees for their valuable comments on the article.

REFERENCES

ALLEN, R. AND GARLAN, D. 1994. Formalizing architectural connection. In *Proceedings of the 16th International Conference on Software Engineering* (Sorrento, Italy, May). 71–80.

AMERICA, P. 1991. Designing an object-oriented programming language with behavioural subtyping. In *Foundations of Object-Oriented Languages, REX School/Workshop* (June 1990), J. W. de Bakker, W. P. de Roever, and G. Rozenberg, Eds. Lecture Notes in Computer Science, vol. 489. Springer-Verlag, New York, 60–90.

ARNOLD, S. P. AND STEPOWAY, S. L. 1987. The REUSE system: Cataloging and retrieval of reusable software. In *COMPCON Spring 87, 32nd IEEE Computer Society International Conference* (Feb.). IEEE Computer Society, Washington, D.C., 376–379.

CARDELLI, L. 1989. Typeful programming. Rep. 45, DEC Systems Research Center, Palo Alto, Calif., May.

CHEN, P. S., HENNICKER, R., AND JARKE, M. 1993. On the retrieval of reusable software components. In *Proceedings of the 2nd International Workshop on Software Reusability* (Mar.). IEEE Computer Society Press, Los Alamitos, Calif., 99–108.

CONSENS, M., MENDELZON, A., AND RYMAN, A. 1992. Visualizing and querying software structures. In *Proceedings of the 14th International Conference on Software Engineering* (May). 138–156.

DHARA, K. K. AND LEAVENS, G. T. 1992. Subtyping for mutable types in object-oriented programming languages. Tech. Rep. 92-36, Dept. of Computer Science, Iowa State Univ. of Science and Technology, Ames, Iowa, Nov.

DHARA, K. K. AND LEAVENS, G. T. 1996. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering* (Mar.).

DI COSMO, R. 1992. Type isomorphisms in a type-assignment framework. In *Proceedings of the 19th Annual POPL* (Jan.). ACM, New York, 200–210.

FISCHER, B., KIEVERNAGEL, M., AND STRUCKMANN, W. 1994. VCR: A VDM-based software component retrieval tool. Tech. Rep. 94-08, Technical Univ. of Braunschweig, Braunschweig, Germany, Nov.

FISCHER, G., HENNINGER, S., AND REDMILES, D. 1991. Cognitive tools for locating and comprehending software objects for reuse. In *Proceedings of the 13th International Conference on Software Engineering* (May). 318–328.

FRAKES, W. B. AND NEJMEH, B. A. 1987. Software reuse through information retrieval. In *The 20th Annual HICSS*. Vol. 2, *Software*, B. D. Shriver, Ed. Western Periodicals Company, 530–535.

GARLAN, D., ALLEN, R., AND OCKERBLOOM, J. 1995. Architectural mismatch: Why reuse is so hard. *IEEE Softw. 12,* 6 (Nov.), 17–26.

GARLAND, S. J. AND GUTTAG, J. V. 1991. A guide to LP, the Larch Prover. Rep. 82, DEC Systems Research Center, Palo Alto, Calif., Dec.

GUTTAG, J. V. AND HORNING, J. J., Eds. 1993. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, New York. (With contributions by S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing.)

GOGUEN, J., NGUYEN, D., MESEGUER, J., DU ZHANG, L., AND BERZINS, V. 1996. Software component search. *J. Syst. Integration 6,* 93–134.

JENG, J.-J. AND CHENG, B. H. C. 1992. Formal methods applied to reuse. In *Proceedings of the 5th Workshop in Software Reuse*.

KATOH, H., YOSHIDA, H., AND SUGIMOTO, M. 1986. Logic-based retrieval and reuse of software modules. In the *5th Annual International Phoenix Conference on Computers and Communications* (Mar.). 445–449.

KATZ, S., RICHTER, C. A., AND THE, K.-S. 1987. PARIS: A system for reusing partially interpreted schemas. In *Proceedings of the 9th International Conference on Software Engineering*. (Mar.). 377–385.

KONSTANTAS, D. 1993. Object-oriented interoperability. In *ECOOP 93—7th European Conference on Object-Oriented Programming,* O. M. Nierstrasz, Ed. Lecture Notes in Computer Science, vol. 707. Springer-Verlag, New York, 80–102.

LEAVENS, G. 1989. Verifying object-oriented programs that use subtypes. Ph.D. thesis and Tech. Rep. 439, Laboratory for Computer Science, MIT, Cambridge, Mass., Feb.

LEAVENS, G. T. AND WEIHL, W. E. 1990. Reasoning about object-oriented programs that use subtypes. In *ECOOP/OOPSLA 90 Proceedings.* ACM, New York.

LISKOV, B. H. AND WING, J. M. 1994. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst. 16,* 4 (Nov.), 1811–1841.

MAAREK, Y. S., BERRY, D. M., AND KAISER, G. E. 1991. An information retrieval approach for automatically constructing software libraries. *IEEE Trans. Softw. Eng. 8,* 17 (Aug.), 800–813.

MAULDIN, M. AND LEAVITT, J. 1994. Web-agent related research at the CMT. In *ACM Special Interest Group on Networked Information Discovery and Retrieval (SIGNIDR-94)* (Aug.). ACM, New York.

MEYER, B. 1988. *Object-Oriented Software Construction*. Prentice-Hall, New York.

MILI, A., MILI, R., AND MITTERMEIR, R. 1994. Storing and retrieving software components: A refinement-based approach. In *Proceedings of the 16th International Conference on Software Engineering* (May). 91–100.

OLSEN, K. A., KORFHAGE, R. R., SOCHATS, K. M., SPRING, M. B., AND WILLIAMS, J. G. 1993. Visualization of a document collection: The VIBE system. *Inf. Process. Manage. 29,* 1, 69–81.

OSTERTAG, E., HENDLER, J., PRIETO-DÍAZ, R., AND BRAUN, C. 1992. Computing similarity in a reuse library system: An AI-based approach. *ACM Trans. Softw. Eng. Methodol. 1,* 3 (July), 205–228.

PAUL, S. AND PRAKASH, A. 1994. A framework for source code search using program patterns. *IEEE Trans. Softw. Eng. 6,* 20 (June), 463–475.

PENIX, J. AND ALEXANDER, P. 1995. Design representation for automating software component reuse. In *Proceedings of the 1st International Workshop on Knowledge-Based Systems for the (Re)use of Program Libraries* (June).

PERRY, D. E. 1989. The Inscape environment. In *Proceedings of the 11th International Conference on Software Engineering.* 2–12.

PERRY, D. E. AND POPOVICH, S. S. 1993. Inquire: Predicate-based use and reuse. In *Proceedings of the 8th Knowledge-Based Software Engineering Conference* (Sept.). 144–151.

PRIETO-DÍAZ, R. 1989. Classification of reusable modules. In *Software Reusability.* Vol. 1, *Concepts and Models*, T. J. Biggerstaff and A. J. Perlis, Eds. ACM Press, New York, 99–123.

RITTRI, M. 1990. Retrieving library identifiers via equational matching of types. Tech. Rep. 65, Programming Methodology Group, Dept. of Computer Sciences, Chalmers Univ. of Technology and Univ. of Göteborg, Göteborg, Sweden, Jan. (Reprinted with corrections May 1992.)

ROLLINS, E. J. AND WING, J. M. 1991. Specifications as search keys for software libraries. In *Proceedings of the 8th International Conference on Logic Programming* (June).

RUNCIMAN, C. AND TOYN, I. 1989. Retrieving re-usable software components by polymorphic type. In *Conference on Functional Programming Languages and Computer Architectures* (Sept.). 166–173.

SALTON, G. AND MCGILL, M. J. 1983. *Introduction to Modern Information Retrieval*. McGraw-Hill, New York.

STRINGER-CALVERT, D. W. J. 1994. Signature matching for Ada software reuse. Master's thesis, Univ. of York, York, England, U.K.

THATTE, S. R. 1994. Automated synthesis of interface adapters for reusable classes. In *Proceedings of the 21st Annual Symposium on Principles of Programming Languages* (Jan.). ACM, New York, 174–187.

VERNON, M., LAZOWSKA, E., AND PERSONICK, S., Eds. 1994. *R&D for the NII: Technical Challenges.* Interuniversity Communications Council (EDUCOM).

WILEDEN, J. C., WOLF, A. L., ROSENBLATT, W. R., AND TARR, P. L. 1991. Specification-level interoperability. *Commun. ACM 34,* 5 (May), 72–87.

WING, J. M., ROLLINS, E., AND ZAREMSKI, A. M. 1993. Thoughts on a Larch/ML and a new application for LP. In the *1st International Workshop on Larch,* U. Martin and J. M. Wing, Eds. Springer-Verlag, New York.

YELLIN, D. M. AND STROM, R. E. 1994. Interfaces, protocols, and the semi-automatic construction of software adaptors. In OOPSLA Conference Proceedings. *SIGPLAN Not. 29,* 10 (Oct.), 176–190.

ZAREMSKI, A. M. 1996. Signature and specification matching. Ph.D. thesis and Tech. Rep. CMU-CS-96-103, School of Computer Science, Carnegie Mellon Univ., Pittsburgh, Pa., Jan.
ZAREMSKI, A. M. AND WING, J. M. 1995. Signature matching: A tool for using software libraries. *ACM Trans. Softw. Eng. Methodol. 4,* 2 (Apr.), 146–170.

Received June 1996; revised January 1997; accepted May 1997