in Proc. of 9th IEEE Intl. Conference on Tools with AI, page 94-101, Newport Beach, CA, Nov. 3-8, 1997

# Formalizing and Automating Component Reuse*

Yonghao Chen and Betty H. C. Cheng†
Department of Computer Science
Michigan State University
East Lansing, MI 48824
email: {chenyong, chengb}@cps.msu.edu

## Abstract

*Using existing components to construct software systems has significant potential to improving software productivity and quality. A key problem in software component reuse is the selection of appropriate components for satisfying a given requirement. In this paper, we define a component interface generality relation that provides a foundation for component selection. This generality relation, represented in terms of formal specifications, precisely captures the semantic obligations for an existing component to satisfy the requirements of a target system. The formal specifications facilitate the (semi-) automatic determination of the generality relation. We show how this generality relation has been used to determine the reusability of software components in a software architecture-based reuse and integration environment.*

## 1. Introduction

The use of existing software components to construct new systems is recognized as having significant potential to improving software development productivity and software quality [2, 6, 11]. Recently, a great deal of effort has been directed towards an architecture-driven, component-based software development paradigm. A software architecture is a collection of components, connectors, and a configuration of how the components should be connected via connectors. By specifying the interfaces of constituent components and connectors of a target system, software architectures serve as frameworks for assembling existing components. Given an architecture specification, the assembly process of software components consists of two phases: First, the existing components are retrieved, evaluated, and matched to the interfaces in the architecture specification. Then a packaging process is invoked to integrate these components, generating code as necessary for adapting components and/or implementing connections.

A key step in the assembly process is the component selection. Given an interface specification, the objective of component selection is to locate the most appropriate components that satisfy the interface. The criteria necessary for a component to satisfy an interface is usually implicit and not precisely captured. For example, a simple, but widely used, criterion is name (keyword) matching. However, keywords cannot convey significantly useful information, unless they are widely accepted terminology, such as mathematical functions, for example, *sin*, *cos*, etc.. A more informative criterion may be based on signatures (syntax and type information). Although signatures encapsulate type information, they still fail to capture the behavior of a component precisely. Natural language descriptions may document semantic information of components. However, the inherent ambiguity of natural languages (together with possible inconsistencies in the documentation) may make it difficult to locate the "right" component.

Recent work in formal methods has produced rich formalisms for use in software development. Among others is the Larch family of specification languages [7] used to specify programs written in C, C++, Modula-3, Smalltalk, Ada, and CLU. This paper describes an approach to software reuse that takes advantage of the logical reasoning and automated processing enabled by formal specifications of software components to determine which existing components can be used to satisfy the requirements of a new (target) system.

The remainder of this paper is organized as follows. The next section discusses the specification of software components. A Larch-styled formalism is presented to specify component interfaces. Based on the formal specifications, we define a component interface generality relation in Section 3. In Section 4, we describe the application of the interface generality relation in a software architecture-based reuse and integration environment. We discuss the use of the Larch theorem prover LP to determine the interface generality relation. Related work is described in Section 5. We

*This work is supported in part by National Science Foundation grants CCR-9633391, CCR-9407318 and CCR-9209873.

†Please contact this author for all correspondences.

conclude this paper with a summary and a brief description of future investigations.

## 2. Component Specification

In simplistic terms, a component is a computational unit. As a part of a system, each component provides certain services to its environment, as well as requires some services from its environment. The services provided by a component represent the *capabilities* of the component, and the services required by a component constitute the *assumptions* that the component requires of its environment. A service may be in a variety of forms, for example, an event in event-based systems, or a data stream in a pipeline system. In this paper, we consider components that have either data or functions (procedures) as their services. These components are also termed modules, since they have the same properties as those described in a conventional module interconnection language (MIL) [12].

The specification of the capabilities and assumptions of a module is encapsulated in the interface of the module. A module may or may not have an implementation. A component without an implementation is also termed an abstract component or simply an interface. Each existing component must have an implementation. The implementation of a module should conform to its interface.

**Definition 1** (*Conformance of modules to interfaces*) A module $M$ conforms to an interface $I$ (denoted as $M \models I$) iff (1) $M$ implements all of the capabilities of $I$, and (2) in order to implement the capabilities of $I$, $M$ must use and only use all of the assumptions stated in $I$. $M$ is called a conforming module of $I$, and $I$ is a conformed interface of $M$.[1]

The conformance definition makes it possible to systematically check the reusability of a component solely based on its conformed interface. It should be noted that an existing module may conform to multiple interfaces, each of which describes a collection of possible behaviors (providing services and/or requiring services) that the module may exhibit in constructing a system.

**Example 1** Consider a simple data access system. The system has three components that are organized into layers, where a given layer uses services provided by the layer below it. At the top is the user interface that accepts user data access requests, and handles them by calling data access functions provided in the middle layer. The middle layer component, in turn, uses an abstract data type (ADT) object defined in the bottom layer of the system. Figure 1 describes the interface of the middle layer component, where functions *read* and *write* are used by the top layer component, and the data object *x* of type *dtModel* is defined by the bottom layer component.

This example will be referenced and refined in the following discussions.

```
comp dataAccess is
    assumptions
        dtModel x;
    capabilities
        bool read(... ...);
        bool write(... ...);
end dataAccess
```

**Figure 1. An example component interface specification**

[1] In this context, module refers to the implementation of the module.

### 2.1. Refining Interface Specifications

In this section, we refine the definition of interfaces. In addition to the capabilities and assumptions of a component, we also include dependence information among capabilities and assumptions in an interface specification. We use Larch-styled formal specifications [7] to specify interfaces. Formal specifications of interfaces precisely capture the functionalities and provide a rigorous foundation to reason about the relations between interfaces. We choose Larch as the target specification language for several reasons. Larch has a simple syntax and provides support for commonly used programming languages, including C, C++, Smalltalk, Modula-3, and ML [7]. Second, Larch has support for libraries of specifications, thus promoting the reuse of specifications. Finally, there is good tool support for developing and analyzing Larch specifications, including a graphically-based browser and editor [9], syntax checker [7], and theorem prover [7].

#### 2.1.1. Specifying Domain Theories

Larch specifications consist of two tiers: one tier specifies domain theories, and the other tier specifies module interfaces in terms of the problem domain. A domain theory is an algebraic model of a problem domain. It defines a set of abstract types (sorts) and operations over those sorts. The Larch Shared Language (LSL) is used to specify a domain theory termed a *trait*. An LSL specification for a table is shown in Figure 2.

The specification starts with the inclusion of another trait, $Integer$. The $Integer$ trait defines the theory of integer, including constants $0$, $1$, and operator $+$, and so on. The $Integer$ trait is specified in the LSL handbook [7], which is a collection (library) of many useful LSL specifications. The **introduces** clause declares a set of operators, each with

```
// LSL specification for Table
Table: trait
 includes Integer
 Entry tuple of index: Ind, value: Val
 introduces
   ϕ: → Tab
   add: Tab, Entry → Tab
   lookup: Tab, Ind → Val
   __ ∈ __:  Ind, Tab → bool
   size:   Tab  → Int
 asserts
   Tab  generated by ϕ, add
   Tab  partitioned by ∈, lookup
   ∀i, i1: Ind, v, v1: Val, t: Tab, e: Entry
   ~(i ∈ ϕ);
   i ∈ add(t, e) == i = e.index ∨  i ∈ t;
   lookup(add(t, e), i1) == if e.index = i1
           then e.value else lookup(t, i1);
   size(ϕ) = 0;
   size(add(t, e)) == if e.index ∈ t
           then size(t) else size(t)+1;
```

**Figure 2. LSL specification for a table**

its own signature. The body of a trait contains, following the keyword **asserts**, equations between terms containing operators and variables. The theory of a trait is the set of all logical consequences of its assertions. The **generated by** clause asserts that each value of the sort $Tab$ is generated by applying $\phi$ and $add$ a finite number of times. And the **partitioned by** clause asserts that all distinct values of $Tab$ can be distinguished by using only the $lookup$ and $\in$ operators.

### 2.1.2. Interface Specifications

In Larch, an interface specification defines an interface between program components. Interface specifications are written in Larch Interface Languages (LILs), which are programming language dependent. For instance, LCL [7] is designed to specify C programs, whereas LM3 [7] is a Larch interface language for Modula-3. The components specified by these LILs are usually programming units, such as procedures, classes, and packages. In the following discussion, we present a Larch style interface specification for architectural components that not only provides services, but also requires services. The interface specification includes two sections, one for specifying the capabilities, the other for specifying the assumptions. In addition, we also specify the dependencies among capabilities and assumptions.

A template for an interface specification is given in Figure 3. The **uses** clause integrates the domain theory specifications and the interface specifications. Interface specifications are written using sorts and values defined in LSL traits. The **assumptions** clause declares the services required by the conforming module, whereas the **capabilities** clause specifies the services provided by the conforming

```
interface  interfaceName  is
   uses LSL traits
   assumptions
      {dataSpec; | functionSpec;}*
   capabilities
      {dataSpec; | functionSpec;}*
end  interfaceName
```

**Figure 3. Template for interface specification**

module. A service can be prefixed with the keyword **virtual**, indicating that the service is not an actual capability or assumption, but it is needed for the purposes of specifying other services. A service is either a piece of data or a function.

A function is formally specified by giving its precondition and postcondition, as depicted in Figure 4. A precon-

```
functionSpec = function typeName  functionName (argList)
                 [ depends on  dependencyList] {
        [ requires  logicalFormula ]
        [ modifies  dataNames ]
         ensures   logicalFormula
     }
```

**Figure 4. Function specification**

dition (specified in the **requires** clause) specifies when the function is applicable, whereas a postcondition (specified in the **ensures** clause) states what should be established by the execution of the function. In addition, the **modifies** clause specifies what data will be modified by the function. The dependencyList in the **depends on** section is used to list the specific assumptions for a given capability. An assumption specific for a capability is either an assumption of the interface or another capability that is required by the interface in order to provide the capability. By default (if not explicitly specified), the assumptions specific for a capability are those assumptions specified in the **assumptions** clause of the interface. By introducing the dependency list of a capability, we have a finer-grained description of the relations between capabilities and assumptions, rather than just a description of the assumptions of an interface as a single entity.

Data is specified by simply declaring it as shown in Figure 5. The behavioral specification for a complex data type (such as an ADT) is attached using the **with behavior** keywords. A behavioral specification consists of a set of function (method) specifications that each specify externally observable behavior of an object of the type.

Figure 6 is the refined interface specification for the component *dataAccess* from the previously presented example,

```
dataSpec = data typeName dataName
             [ depends on dependencyList]
           with behavior behavioralSpec
```

**Figure 5. Data specification**

where within a function specification, $result$ represents the return value of the function, $x$ˆ and $x$' represent the values of data (variable) $x$ before and after the function is executed, respectively. The specification states that the component *dataAccess* has two capabilities, *read* and *write*, that depend on a data assumption, $x$. $x$ is a table with behavior specified for checking the existence of a record in the table, inserting a record into the table, querying and updating the table.

```
comp dataAccess is
  uses Table(dtModel for Tab, int for Ind, int for Val)
  assumptions
    data dtModel x with behavior {
       bool exists(int recNo) {
          ensures result = recNo ∈ x;
        }
       void insert(int recNo, int val) {
          requires ˜(recNo ∈ xˆ);
          modifies x;
          ensures  recNo ∈ x′ ∧ lookup(x′, recNo) = val;
        }
       int query(int recNo) {
          requires recNo ∈ x;
          ensures  result = lookup(x, recNo);
        }
       void update(int recNo, int newVal) {
          requires recNo ∈ xˆ;
          modifies x;
          ensures  lookup(x′, recNo) = newVal;
        }
     }
  capabilities
    function bool read(int recNo, int recVal) depends on x {
       modifies recVal;
       ensures result = recNo ∈ x
           ∧ (if result then recVal′ = lookup(x,recNo));
     }
    function void write(int recNo,int recVal) depends on x {
       modifies x;
       ensures  recNo ∈ x′ ∧ recVal = lookup(x′, recNo)
     }
end dataAccess
```

**Figure 6. dataAccess interface specification**

## 3. Interface Generality Relation

In this section, based on the formal specifications of component interfaces, we define a logic-based generality relation between interfaces. This generality relation provides a rigorous basis, amenable to automation, to determine the reusability of an existing component for satisfying a given requirement.

### 3.1. Generality Relation of Function Specifications

Since the provided and required services of an interface are a set of functions or data items, we first define the generality relation of two function specifications.

**Definition 2** (*Generality relation of function specifications*) Given two function specifications, $g$ and $h$, $g$ is more general than $h$, denoted as $h \preceq_f g$, if the following rules hold:

- **Signature matching**
  - *Arguments rule.* $g$ and $h$ have the same number of arguments. Let the list of argument types of $g$ be ($T_g^1$, $T_g^2$, …, $T_g^n$), and that of $h$ be ($T_h^1$, $T_h^2$, …, $T_h^n$), then there exists a permutation of the list ($T_h^1$, $T_h^2$, …, $T_h^n$), denoted as ($T_h^{1'}$, $T_h^{2'}$, …, $T_h^{n'}$), such that for all $i$, $1 \leq i \leq n$, $T_h^{i'}$ is the same as $T_g^i$.
  - *Result rule.* Either both $g$ and $h$ have a result or neither has one. If there is a result, then $h$'s result type is the same as $g$'s result type.
- **Specification rule** [2]
  Let $pre(f)$ and $post(f)$ be the precondition and postcondition of function specification $f$, respectively.
  - *Precondition rule.* $pre(g) \rightarrow pre(h)$, the precondition of $g$ implies the precondition of $h$.
  - *Postcondition rule.* $post(h) \rightarrow post(g)$, the postcondition of $h$ implies the postcondition of $g$.

The signature matching requires that the two functions' range types match and their domain types match after permutation. The specification rule requires that a more general function have a stronger precondition and a weaker postcondition.

Intuitively, the generality relation between function specifications captures the following implementation property: *let $H$ be a function implementing function specification $h$, then $H$ also implements the function specification $g$ that is more general than $h$.* In other words, whenever a function implementing $g$ is needed, the function $H$ implementing $h$ can be used.

### 3.2. Generality Relation of Data Specifications

The generality relation of data items is determined based on their behavioral specifications.

[2]Parameter renaming has been conducted based on signature matching so that the specifications of function $g$ and $h$ are consistent.

**Definition 3** (*Generality relation of data specifications*) Given two data specifications $d_1$ and $d_2$, let $SPEC_i$ be the set of function specifications of $d_i$'s behavioral specification (where $i$ is 1 or 2), $d_2$ is more general than $d_1$, denoted as $d_1 \preceq_d d_2$, if there exists a map $\pi$ from $SPEC_2$ to $SPEC_1$, such that for any function (method) specification $m \in SPEC_2$, there exists a function specification $\pi(m)$ in $SPEC_1$, and $m$ is more general than $\pi(m)$, i.e., $\pi(m) \preceq_f m$.[3]

The above definition captures the behavioral property that a data specification should provide all the behavior provided by a more general data specification. In terms of implementation, the generality relation between data specifications implies that *a data object that implements a data specification also implements any more general data specifications*.

## 3.3. Generality Relation of Interfaces

An interface specification contains a set of provided services (capabilities) and a set of required services (assumptions) that are needed for providing all of the capabilities. The assumptions for a specific capability are specified in the dependency list of the capability specification. An assumption for a specific capability is either an assumption of the interface or another capability of the interface. The relations among the services (both provided and required) of an interface can be depicted using a *directed acyclic graph* (DAG), called a dependence DAG. For instance, Figure 7 contains the dependence DAG of the interface *dataAccess* depicted in Figure 6.

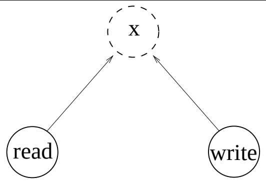


**Figure 7. Dependence DAG of dataAccess interface**

In Figure 7, there are three nodes, each of which corresponds to a service. The dashed circle (node) represents an assumption, whereas the solid circle (node) represents a capability. The directed edge $\overrightarrow{\langle u, v \rangle}$ means that $u$ depends on $v$ ($v$ is an assumption needed for providing $u$).

[3]There should be an abstraction function that maps the value space (described in terms of the Larch Shared Language) of $d_1$ to that of $d_2$. We omit it here for simplicity. For details please see [1, 10].

**Definition 4** (*Dependence DAG of interfaces*) Given an interface $I$, the dependence DAG of $I$, denoted as $G_I = \langle V, E \rangle$, is derived from the interface specification for $I$. Specifically, each service of $I$ defined both in terms of the capabilities and the assumptions corresponds to a node in $V$ (we name the node using its service name). If a service $u$ depends on another service $v$, then directed edge $\overrightarrow{\langle u, v \rangle} \in E$.

In the following discussion, $C_I$ and $A_I$ represent the set of capabilities and the set of assumptions specified in interface $I$, respectively.

**Definition 5** (*Assumptions specific to a single capability*) In an interface $I$, let $A_I^c$ denote the set of assumptions upon which a capability $c$ depends, then

$$A_I^c = \{t \mid t \in A_I \text{ and } t \text{ is reachable from } c \text{ in } G_I\}.$$

where $G_I$ is the dependence DAG of interface $I$. A node $u$ is reachable from a node $v$ in a directed acyclic graph (DAG), if there exists a directed path from $v$ to $u$.

For example, in the *dataAccess* interface, $A_I^{read} = \{x\}$ and $A_I^{write} = \{x\}$.

**Definition 6** (*Assumptions specific to a set of capabilities*) For a set of capabilities, $K$, of interface $I$, let $A_I^K$ be the set of assumptions upon which capabilities of $K$ depend, then

$$A_I^K = \bigcup_{c \in K} A_I^c$$

According to the above definition, it is true that $A_I^{C_I} \subseteq A_I$ for any interface $I$. However, a correctly specified interface should not have the case where $A_I^{C_I} \subset A_I$, because this would mean that stronger assumptions than needed have been imposed over the interface.

**Definition 7** (*Interface generality relation*) Given two interfaces $I_1$ and $I_2$, $I_2$ is more general than $I_1$, i.e., $I_1 \preceq I_2$, if there exists a map $\pi_{\mathbf{c}} : C_{I_2} \rightarrow C_{I_1}$, such that the following rules hold:

- **Capabilities rule.**
  $\forall s_2 \in C_{I_2}$, there exists a $s_1$, $s_1 \in C_{I_1}$, such that $s_1 = \pi_{\mathbf{c}}(s_2)$, and $s_1$ and $s_2$ are both data specifications or function specifications, and $s_2$ is more general than $s_1$, that is,
  - $s_1 \preceq_d s_2$, if $s_1$ and $s_2$ are both data specifications
  - $s_1 \preceq_f s_2$, if $s_1$ and $s_2$ are both function specifications

- **Assumptions rule.**
  There exists an onto map $\pi_{\mathbf{a}} : A_{I_1}^{\pi_{\mathbf{c}}(C_{I_2})} \rightarrow A_{I_2}$, such that, $\forall s_1 \in A_{I_1}^{\pi_{\mathbf{c}}(C_{I_2})}$, let $s_2 = \pi_{\mathbf{a}}(s_1)$, $s_1$ and $s_2$ are both data specifications or function specifications, and $s_1$ is more general than $s_2$, that is,
  - $s_2 \preceq_d s_1$, if $s_1$ and $s_2$ are both data specifications
  - $s_2 \preceq_f s_1$, if $s_1$ and $s_2$ are both function specifications

The capabilities rule states that a more specific interface $I_1$ should provide a corresponding capability for each capability specified in a more general interface $I_2$, and the corresponding capability in $I_1$ should be more specific than that in $I_2$. The assumptions rule states that every assumption required by the more specific interface $I_1$ needed to provide the specified capabilities should have a corresponding assumption in $I_2$ that is more specific. Moreover, the corresponding relation between the assumptions should be onto.

We claim that the interface generality relation defined in Definition 7 has the following property: (An informal justification for this claim is given in [4].)

**Property 1** Given two interfaces $I_1$ and $I_2$, if $I_1 \preceq I_2$, i.e., $I_2$ is more general than $I_1$, then any module $M$ conforming to $I_1$ also conforms to $I_2$.

Based on the generality relation, we can define the reusability of an existing module for fulfilling a given requirement, either for the construction of new systems or in the maintenance of existing systems.

**Definition 8** (*Reusability*) Given a requirement, represented as an abstract component (interface) $I$, an existing component $M$ (with conformed interface $I_m$) is reusable for fulfilling $I$ if $I_m \preceq I$.

According to Property 1, it is easy to see $M \models I$, that is, $M$ conforms to $I$. Therefore, we can use $M$ to implement $I$. More specifically, we can map every capability $c$ in $I$ to $\pi_{\mathbf{c}}(c)$ of $I_m$, and every assumption $a$ in $A_{I_m}^{\pi_{\mathbf{c}}(C_I)}$ to $\pi_{\mathbf{a}}(a)$ of $I$.

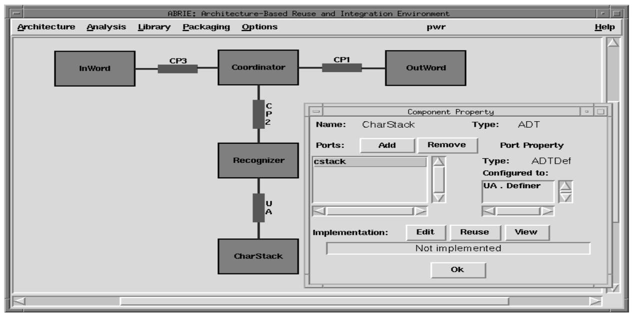
## 4. ABRIE: A Software Architecture-Based Reuse and Integration Environment

ABRIE is an experimental system designed to explore the use of software architectures as a framework for assembling software components. The design objective is to provide an integrated environment to address various reuse issues: composition specification, component management and evaluation, and component integration. Three characteristics are particularly emphasized in the ABRIE design: visualization, multilevel abstractions, and automation. In addition to the textual form, visual representation and manipulation of components, connections, and architectures are supported. ABRIE uses three levels of abstraction in determining the reusability of existing components: types, signatures, and formal (logic-based) specifications. Automation is one main potential benefit of architecture-based reuse. In ABRIE, the component integration (packaging) process is fully automated. Based on the formal (logic-based) specifications, (semi-)automatic support in component evaluation is provided by integrating the Larch Theorem Prover (LP) to assist in the determination of the interface generality relation between components.

In ABRIE, a component has a set of ports, each of which describes a service that the component provides or requires. Connectors are encapsulations for interactions among components. A connector consists of a collection of roles for the participants of the interaction specified by the connector. Components and connectors are put together to describe an architecture by configuring the ports of components to roles of connectors. Components, connectors, and ports are typed. The type of a component is intended to capture the architectural properties, that is, the way the component may be used. Connector types capture recurring component interaction styles. Finally, port types encapsulate information about services provided or required by a component.

In the following discussion, we focus on the integration and use of LP in ABRIE. We illustrate how LP can facilitate the determination of interface generality relation, and thus automate the component evaluation process. More detailed discussions on ABRIE regarding its features and development can be found in [3].

The main working area of ABRIE is a canvas that provides a graphical representation and manipulation for architectural elements. Figure 8 shows a main program/subroutine style architecture for an example system *pwr* for recognizing palindrome words.

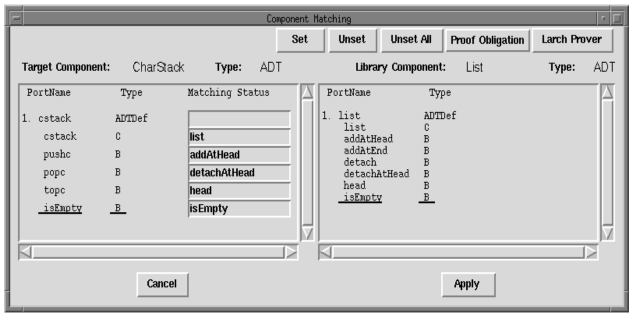


Name: CharStack
Type: ADT
Ports:
cstack
Port Property
Type: ADTDef
Configured to:
UA . Definer
Implementation:
Not implemented

Figure 8. ABRIE Architecture Design

In *pwr*, *Coordinator* is the main control component that calls procedures defined in components *InWord*, *OutWord*, and *Recognizer*. *InWord* inputs words from an *InFile* port that is configured to the standard input. *OutWord* writes words to a port that is configured to the standard output. *Recognizer* defines and exports a procedure *isPalindrome* that checks if a word is a palindrome word. *Recognizer* imports and uses a character stack through a connector UA. Component *CharStack* defines and exports an ADT *cstack*.

In order to illustrate the process of evaluating and reusing existing components in ABRIE, let us consider the implementation of component *CharStack*. The "Component Property" window in Figure 8 shows the properties of component *CharStack*. By clicking the "Reuse" button in the "Component Property" window, the user instructs ABRIE to search its component library and return all the candidate components that are of the same type as *CharStack*. Based on their specifications, the user selects one candidate for further evaluation. Figure 9 shows the scenario of matching a library component *List* for satisfying *CharStack*, where *B* and *C* represent behavior and constructor, respectively.



Target Component: CharStack Type: ADT

| PortName | Type | Matching Status |
|---|---|---|
| 1. cstack | ADTDef | |
| cstack | C | list |
| pushc | B | addAtHead |
| popc | B | detachAtHead |
| topc | B | head |
| isEmpty | B | isEmpty |

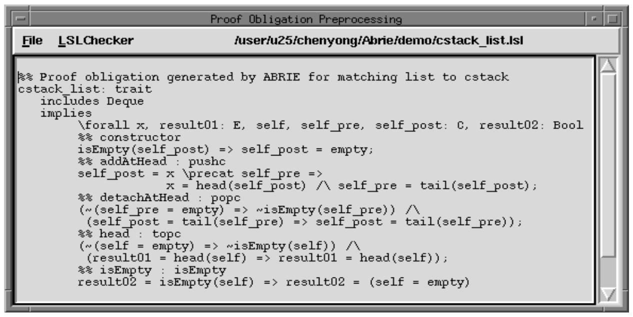
Library Component: List Type: ADT

| PortName | Type |
|---|---|
| 1. list | ADTDef |
| list | C |
| addAtHead | B |
| addAtEnd | B |
| detach | B |
| detachAtHead | B |
| head | B |
| isEmpty | B |

**Figure 9. Component Matching**

Both *List* and *CharStack* have only a provided service: a data item specified with a set of operations and their respective behavior. In order to determine the reusability of component *List*, we need to establish whether the ADT *cstack* defined in component *CharStack* is more general than *list* in component *List*. We proceed by assigning a mapping between the operators of the two ADTs. As shown in Figure 9, we assume that the constructor *cstack* is more general than constructor *list*, *pushc* than *addAtHead*, *popc* than *detachAtHead*, and etc.. Given the mapping, ABRIE will automatically generate the proof obligations for justifying the mapping. Figure 10 depicts the proof obligations generated for the mapping shown in Figure 9.

The proof obligations are generated based on the Larch specifications of the two components. In order to facilitate the application of the Larch Prover (LP) for analyzing them, the proof obligations are represented in terms of LSL specifications. After preprocessing the proof obligations, we can invoke the Larch Prover (LP) from the "Component Matching" window to assist in proving these obligations. If all these obligations are successfully resolved, then the user-assigned mapping is justified, and thereby establishing the


Proof Obligation Preprocessing
File
LSLChecker
/user/u25/chenyong/Abrie/demo/cstack_list.lsl

```
%% Proof obligation generated by ABRIE for matching list to cstack
cstack_list: trait
   includes Deque
   implies
        \forall x, result01: E, self, self_pre, self_post: C, result02: Bool
        %% constructor
        isEmpty(self_post) => self_post = empty;
        %% addAtHead : pushc
        self_post = x \precat self_pre =>
                    x = head(self_post) /\ self_pre = tail(self_post);
        %% detachAtHead : popc
        (~(self_pre = empty) => ~isEmpty(self_pre)) /\
         (self_post = tail(self_pre) => self_post = tail(self_pre));
        %% head : topc
        (~(self = empty) => ~isEmpty(self)) /\
         (result01 = head(self) => result01 = head(self));
        %% isEmpty : isEmpty
        result02 = isEmpty(self) => result02 = (self = empty)
```

**Figure 10. Proof Obligations**

reusability of the library component. Figure 11 shows a snapshot of LP while it is discharging the proof obligations for our example. LP is an interactive theorem proving system for multi-sorted first-order logic. In certain cases, user interaction may be required in proving/disproving a conjecture. However, for our example, all the proof obligations are automatically proved by LP. Therefore the assumed mapping is justified and we can reuse the *List* component for satisfying component *CharStack*.

```
LP1.15: prove
  (isEmpty(self_post) => self_post = empty)
  ..
Attempting to prove conjecture cstack_listTheorem.1:
  isEmpty(self_post) => self_post = empty
Conjecture cstack_listTheorem.1
[] Proved by normalization.
Deleted formula cstack_listTheorem.1, which reduced
to 'true'.

LP1.16: prove
  (self_post = (x:E \precat self_pre) =>
  (x:E = head(self_post) ∧ self_pre = tail(self_post)))
  ..
Attempting to prove conjecture cstack_listTheorem.2:
  self_post = x \precat self_pre =>
  x = head(self_post) ∧ self_pre = tail(self_post)
Conjecture cstack_listTheorem.2
[] Proved by normalization.
Deleted formula cstack_listTheorem.2, which reduced
to 'true'.
```

**Figure 11. A Snapshot of LP in resolving proof obligations**

Based on the justified mapping between ports (and behavior of ports), ABRIE will automatically generate code necessary for resolving naming conflicts between the library component and the target component during the system packaging process [3].

## 5. Related Work

Jeng and Cheng [8] proposed specification matching for reuse based on order-sorted predicate logic (OSPL). They defined both exact and relaxed matches. Zaremski and Wing [13] summarized several types of specification matchings to capture different behavioral relations. Both approaches discussed module specification matching, where a module simply refers to a collection of individual functions, rather than an architectural unit such as those that we considered in this paper. In both approaches, there is no support for capturing assumptions that a module requires of its environment, nor are the dependence relations of the functions within a module considered.

Several behavioral notions of subtyping in object-oriented design have recently been proposed [1, 5, 10]. These notions capture the substitutability property of subtypes: Given a program $P$, when an object $o$ of type $T$ in $P$ is substituted by $o'$ of type $S$, where $S$ is a subtype of $T$, the behavior of $P$ will remain unchanged. Similar to our notion for interface generality relation, these definitions for behavioral subtyping not only consider syntactic constraints (such as matches between signatures), but also emphasize the semantic obligations. The main difference between our work and these projects lies in the granularity of components: their components (objects or object types) are relatively fine-grained, whereas we consider architectural units that have structural constraints imposed by *requiring* or *providing* services. In addition to theoretical investigations, we also explored the use of the interface generality relation as a basis for component evaluation and selection in ABRIE. ABRIE integrates tools to facilitate the determination of the interface generality relation.

## 6. Summary and Future Investigations

In this paper, we addressed an important issue in software component reuse: determining the reusability of an existing component for fulfilling a requirement. We first presented a formalism for describing component interfaces, and then we defined a component interface generality relation that explicitly captures the semantic obligations for an existing component to satisfy a requirement. We also showed how this interface generality relation can be used in automating the evaluation and selection of reusable components in a software architecture-based reuse and integration environment.

The interface generality relation defined in this paper is applicable to components that have data or procedures as their services. However, an actual system may consist of other kinds of components, such as filters that interact with their environment through pipes. We are currently investigating the generality relation for these kinds of components.

## References

[1] P. America. Designing an object-oriented programming language with behavioral subtyping. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *LNCS*, volume 489, pages 60–90. Springer-Verlag, 1991.

[2] V. R. Basili, L. C. Briand, and W. L. Melo. Measuring the impact of reuse on quality and productivity in object-oriented systems. Technical Report UMIACS-TR-95-2, University of Maryland, Computer Science Department, 1995.

[3] Y. Chen and B. Cheng. Facilitating an automated approach to architecture-based software reuse. In *Proc. of 12th IEEE Intl. Conference on Automated Software Engineering (to appear)*, November 1997.

[4] Y. Chen and B. H. C. Cheng. Formalizing and automating component reuse. Technical Report MSU-CPS-97-15, Michigan State University, Computer Science Department, May 1997.

[5] K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering (ICSE'18)*, Berlin, Germany, March 1996.

[6] W. B. Frakes and S. Isoda. Success factors of systematic reuse. *IEEE Software*, 11, September 1994.

[7] J. V. Guttag and J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[8] J.-J. Jeng and B. H. C. Cheng. Specification matching for software reuse: A foundation. In *SSR'95*. ACM SIGSOFT, ACM Press, April 1995.

[9] M. R. Laux, R. H. Bourdeau, and B. H. C. Cheng. An integrated development environment for formal specifications. In *Proc. of the IEEE 5th Intl. Conf. on Software Engineering and Knowledge Engineering*, June 1993.

[10] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages*, 16(10), November 1994.

[11] H. Mili, F. Mili, and A. Mili. Reusing software: Issues and research directions. *IEEE Transactions on Software Engineering*, 21(6):528–561, June 1995.

[12] R. Prieto-Diaz and J. M. Neighbors. Module interconnection languages. *J. Systems and Software*, 6(4), Nov 1986.

[13] A. M. Zaremski and J. M. Wing. Specification matching of software components. In *3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, October 1995.