
This is to certify that the

dissertation entitled

AUTOMATING COMPONENT-BASED SOFTWARE DEVELOPMENT

presented by

Yonghao Chen

has been accepted towards fulfillment
of the requirements for

Doctoral degree in Computer Science & Engineering

Betty H. C. Cheng
Major professor

Date 8/27/99

# AUTOMATING COMPONENT-BASED SOFTWARE DEVELOPMENT

By

*Yonghao Chen*

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science

August 27, 1999

## Abstract

### Automating Component-based Software Development

By

*Yonghao Chen*

The ever-increasing demand for quality software has been driving the search for methods to develop higher quality software using lower costs within a shorter time frame. Through the use of existing software components to construct new systems, software reuse is broadly regarded as such a promising approach. However, software reuse in general has not yet fulfilled its promise to significantly improve software development productivity and software quality. In this research, we propose an approach to software reuse. Our approach is based on the premise that effective reuse can be achieved only when reuse issues are considered throughout the software development life cycle and are addressed on the basis of a formal foundation. The overall objective of this research is to develop an architecture-based component reuse framework. This framework addresses software reuse issues in an integrated fashion and is intended to be amenable to automation in such tasks as component evaluation, adaptation, and integration. We also describe the implementation and application of a prototyping system of the proposed reuse framework.

# ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Betty H.C. Cheng, whose support and advice is one of the main reasons that I have finally reached the end of this long process. I also wish to thank Dr. Anthony S. Wojcik, Dr. Jacob Plotkin and Dr. Kurt Stirewalt for their constructive suggestions and commitment for being on my committee.

I am thankful to all the members of the Software Engineering Research Group at MSU, past or present, for their help and encouragement. I also owe thanks to all the great people at MSU who keep things running smoothly.

I extend my heartfelt thanks to Yunyun, for all her company, love, support, understanding and tremendous sacrifice she has made in the past five years.

Finally, I would like to thank my grandma, my parents and my sister for their endless love and unconditional support.

TABLE OF CONTENTS

# List of Figures

# List of Tables

# Chapter 1

# Introduction

With the rapidly expanding role of software in our society, the demand for quality software has been significantly increasing. This demand drives the continuing search for methods to develop higher quality software using lower costs within a shorter time frame. Formal methods [1, 2, 3, 4, 5], automatic programming [6, 7, 8], and object-oriented development [9, 10] are, among others, approaches that have improved software development productivity and software quality over the years. However, as successfully argued by Brooks in his famous 1986 paper [11], there is no "silver bullet". Software development is still a laborious and error-prone process, as manifested by the seemingly endless reports of huge economic or even life losses caused by software or software development errors [12, 13, 14, 15].

One radical solution to the essential difficulties of software development is to minimize the amount of development performed. Software reuse is one technique that supports this approach. Currently, customers have a large number of software packages in the market from which to choose for achieving various purposes ranging from complicated database management to simple communication utilities. The "buy rather than build" philosophy has resulted in the vast use of software due to the low cost and quickness to deployment.

Software reuse is not restricted only to the final executable products. When the concept of reuse is applied to software development, we have another kind of software reuse: the use of existing software components to construct new systems. In a broad sense, any artifact generated during software development may be reused, including requirements specifications, design, implementation, and test plans [16, 17, 18].

## 1.1 Problem Description and Motivation

Achieving reuse in software development has been a much sought after goal ever since McIlroy first proposed it in 1968 [19]. From a high-level view, software reuse involves two types of activities: component acquisition and component use. Component acquisition addresses the creation of new components, representing and classifying them, organizing them into a library, and retrieving them from the library. Component use involves the logical composition of components to satisfying a requirement, and the physical integration of those components.

Although there exists a great deal of successful reuse experience, such as those widely used subroutine libraries [20, 21, 22, 23, 24], software reuse, in general, has not yet fulfilled its promise to significantly improve software development productivity and software quality [25]. In the following subsections, we analyze the reasons from several aspects of reuse, including component evaluation, composition and integration, and technical integration. We also discuss potential solutions to these problems.

### 1.1.1 Component Evaluation

A key step in reuse is to locate the most appropriate components that satisfy a given query requirement. The criteria necessary for a component to satisfy a query requirement

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 [5]. For example, the Larch family of specification languages [26] has been used to specify programs written in C, C++, Modula-3, Smalltalk, Ada, and CLU. Applying formal methods to reuse may not only precisely capture the semantic obligations for an existing component to satisfy a given requirement, but also facilitate a (semi-)automated approach to determine reusability.

A key issue in specification based software reuse is to define a specification matching criterion by which the reusability of an existing component for fulfilling a query specification can be determined. While a number of formal specification based criteria have been proposed [27, 28, 29, 30], and their usefulness have individually been argued in one way or another, there does not exist a general approach to reason about the connections between a matching criterion and its usefulness for determining reusability. Due to the lack of such a general approach, some problems regarding specification matching cannot be solved in an efficient way. For example, given a specification match that happens to be suitable for determining reuse, it is still an open question as to whether there exist *better* specification

match. Intuitively, one specification match is *better* than another if it can identify reusable components for a given query that the other match fails to identify. Another problem with current specification-based evaluation methods is that they are only applicable to functions or modules, rather than architectural components, which we will discuss in next subsection.

### 1.1.2 Composition and Integration

One major obstacle to effective reuse is due to the semantic gap between a query requirement and available components. Even for a requirement of moderate complexity, it is seldom the case that an existing component can exactly implement it. Instead, an appropriate composition of a set of components is usually needed to satisfy the requirement. Conventional software development does not consider component-oriented composition design and specification as a design issue. As a result, reuse is delegated to the implementation stage when the major design decisions have been made and the opportunities of reuse have been considerably constrained. This late consideration for reuse in software development also causes difficulties for component integration. In the cases of subroutine libraries, components are packaged as procedures and can be invoked through the ubiquitously supported integration mechanism *procedure call*. However, it is typical that an existing component may have assumed other integration mechanisms than *procedure call*, such as *data flow*, or *implicit invocation*. In these cases, the integration of those functionally composable components presents a serious obstacle to reuse [31]. To make it worse is the fact that these integration assumptions are usually not well documented.

Recent advances in software architecture design and specification provide a means to incorporate component-oriented composition into software development. Following the conventional approach to requirements analysis and high-level design, software architecture

design generally handles the allocation of system functions to computational components and identifies the inter-relationships among components [32]. Software architectures can be considered to be logical compositions of components, and may serve as a framework into which existing components can be evaluated and integrated [33].

Architecture-based component reuse increases the granularity of artifacts for reuse. Contrary to subroutines, those coarse-grained artifacts, also called architectural components, are usually packaged in a variety of ways, such as a filter in pipe-line systems, a manager in object systems, or as a process in communication systems. For the effective use of these architectural components, we need to consider both their functionalities and packaging properties [34]. Unfortunately, conventional specification techniques focus only on functional aspects. On the other hand, current architectural description languages are generally too high-level and weak in functional specification, since their emphasis is on the specification of architectural properties.

### 1.1.3 Integration of Reuse Techniques

The lack of a seamless integration of component acquisition and component use techniques imposes significant barriers to achieving effective reuse, particularly with respect to component integration. Component integration typically involves conflict identification, component adaptation, and implementation of connections between components. Usually an existing component does not exactly match a given query specification. Component adaptation is required in this case to resolve the conflicts (or mismatches) between an existing component and a requirement. For instance, a typical mismatch is that the order of arguments to an existing procedure may be different from that of a query specification, even though the procedure can satisfy the query functionally. In *ad hoc* approaches to reuse,

component integration is usually considered a separate process from component specification and retrieval tasks. This separation of tasks may make component integration a time-consuming and error-prone process. First, conflicts are not easy to identify at time of integration, since they are usually hidden in the implementation. Second, component adaptation may be quite difficult, particularly for conflicts that are introduced at the design level, such as those regarding packaging properties [31]. Third, connections between software components are usually treated implicitly, rather than encapsulated and described as explicit entities. This approach increases the complexity of a connection implementation, since the implementation is distributed throughout the implementation of components.

## 1.2 Proposed Approach

This research proposes an integrated approach to software reuse. Our approach is based on the belief that effective reuse can be achieved only when reuse issues are considered throughout the software development life cycle and are addressed on the basis of a formal foundation.

**Thesis Statement:** *The overall objective of the proposed research is to develop an architecture-based component reuse framework. This framework will address software reuse issues in an integrated fashion, and is amenable to automation for such tasks as component evaluation, adaptation, and integration.*

## 1.3 Organization of Dissertation

The remainder of this manuscript is organized as follows. In Chapter 2, we overview background material regarding formal methods and software architectures. We examine related

work in Chapter 3. A framework for component reuse is presented in Chapter 4. This framework serves as a technical roadmap to the proposed research. Chapter 5 describes a component interconnect model. Methods for component and composition specifications are introduced in this chapter. In Chapter 6, we present a semantic foundation to specification matching. Based on this semantic foundation, a framework is developed to evaluate the usefulness of various specification matches with respect to determining reusability. In Chapter 7, we extend specification matching to the evaluation of architectural components. Component interface generality relation is introduced as a means to evaluate the reusability of architectural components. Our empirical investigations are described in Chapter 8 and 9. Chapter 8 describes a prototype framework, ABRIE, that we developed to validate our research. A case study that illustrates how our approach and ABRIE can be applied to the development of an actual system is described in Chapter 9. Finally, in Chapter 10, we give a summary of this work, including its impact, as well as suggest potential future investigations.

# Chapter 2

# Background

In this chapter, we introduce relevant background material regarding formal methods, software architectures, and software composition.

## 2.1 Formal Methods

Formal methods used in software development are rigorous techniques for specifying, developing, and verifying computer software [1, 5]. A formal method consists of a well-defined specification language with a set of well-defined inference rules that can be used to reason about a specification. A benefit of formal methods is that their notations are well-defined and thus, are amenable to automated processing. A number of formal methods have been developed, for example, Z [3], VDM [2], Larch [26], and Lotos [35].

In this research, we will use Larch to specify the functional aspects of software components and reason about their reusability. 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 [26]. Second, Larch has support for libraries of specifications, thus promoting the reuse of specifica-

tions. Finally, there is good tool support for developing and analyzing Larch specifications, including a graphically-based browser and editor [36], syntax checker [26], and theorem prover [37].

### 2.1.1 Larch

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 phone book is shown in Figure 2.1.

```
PhoneBook: trait
   includes Integer, String(Char, Name)
   introduces
       new:     → B
       add:     B, Name, Int → B
       find:    B, Name → Int
       __ ∈ __:  Name, B → bool
   asserts
       B  generated by new, add
       ∀ b: B, s, t : Name, n: Int
       ~(s ∈ new);
       s ∈ add(b, t, n) ==
            s = t ∨ s ∈ b;
       find(add(b, s, n), t) ==
           if s = t then n else find(b, t);
```

Figure 2.1: LSL specification for a phone book

The specification starts with the inclusion of other traits, **Integer** and **String**. The **Integer** trait defines the theory of integer, including constants 0, 1, and operator +, and

so on. The **String** defines the string theory. The parameters appearing in **String** mean that **Name** is the defined sort, the string of characters. The **Integer** and **String** traits are specified in the LSL handbook [26], which is a collection (library) of many useful LSL specifications. It is also possible to develop handbooks for specific domains and applications such as networking and graphical user interfaces. The **introduces** clause declares a set of operators, each with 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 B is generated by applying **new** and **add** a finite number of times.

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 [26] is designed to specify C programs, whereas LM3 [26] is a Larch interface language for Modula-3. The components specified by these LILs are usually programming units, such as procedures, classes, and packages. Figure 2.2 shows a LCL specification for inserting an entry into a phone book.

```
uses PhoneBook(phonebook for B, int for Int)

void insert(phonebook b, Name s, int n)
  requires  ~(s ∈ b^);
  modifies  b;
  ensures   s ∈ b' ∧ find(b', s) = n ∧
            ∀ t: Name (if t ∈ b^
             then t ∈ b' ∧ find(b^, t) = find(b', t));
```

Figure 2.2: LCL specification for inserting an entry to a phone book

The **uses** clause integrates the domain theory specifications and the interface specifications. Interface specifications are written using sorts and values defined in LSL traits. As shown in Figure 2.2, a function is formally specified by giving its precondition and postcondition. A precondition (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. In Figure 2.2, `b^` and `b'` represent the values of phone book `b` before and after the execution of function `insert`, respectively.

## 2.2 Software Architectures

Software architectures define the overall structures of software systems. With the increase in the size and complexities of software systems, software architecture design and specification emerges as a critical aspect in successful software engineering activities. Contrary to traditional techniques that focus on the design of algorithms and data structures, software architecture design is concerned with issues such as "gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives" [38].

Disciplined uses of software architectures have a profound impact on the development, evolution and maintenance of software systems [39]. As a design dimension, software architectures provide a high-level design space in which various alternatives for system organization can be explored and analyzed, thus leading to a rational way to system architecting. As a design notation, software architectures may serve as guidance to further development, including refinement, reuse, and implementation. As design abstractions that bridge the

gap between requirements specifications and detailed implementations, architectures provide traceability for software development, which is crucial in the understanding, evolution and maintenance of software systems.

Albeit its importance, software architectures had not been explicitly and systematically used in practice until recently. Traditionally software architectures are described informally as box-and-arrow diagrams [9, 10, 40]. This informality severely restricts the effective use of software architectures. In recent years, a great deal of research efforts have led to the development of formal foundations for software architectures, including architecture modeling, architectural styles, architecture description languages, architectural analysis techniques, and so forth [32, 38, 41, 42, 43, 44, 45, 46, 47, 48, 49].

### 2.2.1 Architecture Model

Although there does not exist a universally accepted definition for software architecture yet, a widely used model for software architecture is to view an architecture as a configuration of two kinds of distinct, identifiable architectural elements: components and connectors [38, 50]. Components are the locus of computation and state, whereas connectors encapsulate the interaction protocols among components. Several widely used interaction protocols are *procedure call*, *data sharing*, *remote procedure call*, *data flow*, and *pipeline*. Components are composed through the mediation of connectors. Both components and connectors can be refined as a composition of lower level elements. This composition itself is an architecture. Thus hierarchical architectures are supported. A component may be implemented directly by a program unit supported in a programming language. Typical program units include Ada packages, C++ classes and objects, procedures, data structures, and so on. An interaction protocol may also be directly supported by programming languages,

operating systems, or middleware. For example, most high-level programming languages support *procedure call* and *data sharing*. Unix operating system supports *pipeline* [51]. Several middleware packages are commercially available to support *remote procedure call* [52].

### 2.2.2 Architectural Style

In practice, many systems have exhibited common architectural characteristics. Architectural styles are introduced to capture these commonalities. An architectural style provides a specialized vocabulary of components, connectors and constraints regarding compositions for a family of systems [38, 41]. For example, a Unix shell pipe-and-filter architecture may only have filters as components and pipes as connectors, and a linear composition topology [51]. Several frequently used architectural styles have been identified and codified, including pipes and filters, data abstraction and object-oriented organization, event-based systems, layered systems, repositories, table driven interpreters, and main program/subroutine organizations [38]. By codifying recurring patterns of software organization, architectural styles promote the reuse of architectural designs [53].

### 2.2.3 Architecture Description Languages

An architecture description language (ADL) is a set of formal notations for representing and analyzing architectural designs. An ADL usually provides a conceptual framework and a concrete syntax for characterizing software architectures [54]. A number of ADLs have been developed with different areas of emphasis in mind: Wright focuses on formalizing connectors [42, 43], SADL for architecture refinement [55, 56], Rapide for architecture simulation and analysis [46], Darwin for specifying the configuration of distributed systems and dynamic architectures [47], UniCon for capturing design abstractions (idioms) used in

practice by software designers [50], Aesop for the rapid construction of style-oriented architecture design environments [57, 58], and ACME for providing a common interchange format for architectural design tools [54].

# Chapter 3

# Related Work

Related work can be broadly divided into two categories: formal methods applied to software reuse, and component-based software development. The latter can further be partitioned into three subcategories: component-oriented programming, architecture-based development, and object-oriented framework. We overview various projects of each category or subcategory in the following discussion.

## 3.1 Formal Methods Applied to Software Reuse

In an attempt to automate software reuse, several projects have investigated the use of formal methods to specify, classify, and retrieve software components.

### 3.1.1 Jeng and Cheng

Adopting order-sorted predicate logic (OSPL) as a reasoning basis for their axiomatic specifications (i.e., pre/post conditions) of software components, Jeng and Cheng [27, 28] developed an automated approach to the classification and organization of reusable software components. At the center of their approach is the classification scheme and algorithms for

automatically constructing a hierarchy of software components that provide a means for representing, storing, browsing, and retrieving reusable components. The hierarchical relationships of the reuse system are based on a *generality* relationship and similarities between software components. The similarities are calculated with respect to a partition of operators into equivalence classes. In order to combine these two concepts into one framework, the component library is structured as a two-tiered hierarchy in two stages. The resulting library structure comprises lower-level and higher-level hierarchies. The lower-level hierarchy is first created by a OSPL subsumption test algorithm that determines whether one component is more general than another. Based on the generality relationship, the most general components are placed at the top of the hierarchy and the more specific or restrictive components at the bottom. Given the lower-level hierarchy, the higher-level hierarchy is then generated by a hierarchical clustering algorithm that groups the most similar components together. The end result is a connected hierarchy of software components organized from the most general to the most specific.

Given the two-tiered hierarchy of reusable software components, the search and retrieval process proceeds from the higher-level hierarchy to the lower-level one, that is, from a coarse-grained search to a fine-grained one for reusable candidates. At the higher-level hierarchy, a query is mapped to some index that indicates the starting nodes within the hierarchy at which the searching algorithm is to begin. After performing the coarse-grained search, the search space may be greatly reduced. The remaining portion of the higher-level hierarchy and the corresponding lower-level is searched using formal reasoning techniques, thus providing an exact determination method. Three kinds of existing specifications may be returned as the results of the logical reasoning based retrieval process: an exact match to the current specification, a more general one, or a more specific one.

A prototype browser that provides a graphical framework for their approach has been developed in the Prolog language. This browser enables the creation of a graphical representation for a component hierarchy from a set of specifications, and provides a user friendly way to traverse the hierarchy as well as retrieve candidate components.

Jeng and Cheng also investigated component adaptation. Two types of adaptation are proposed in their work. One is for those components that are semantically more general than the query specification [59]. The other is for those components that are syntactically analogous to the query specification [60, 61]. Both types of adaptation modify the reusable component at the specification level first instead of at the code level. For the adaptation of more general components, the difference between the existing specification and the query specification is determined by logical reasoning, similar to that used for constructing the two-tiered hierarchy. For the adaptation of analogous components, a recursive matching process is used to establish analogical matches between the existing specification and the query specification. An analogical match is a group of associated pairs between symbols in two logical terms that belong to the existing and query specifications, respectively. Once the changes to the existing specification needed to make it satisfy the query specification have been determined, the information needed to modify the existing specification is used in a formal process for modifying the existing component to fit the query specification.

### 3.1.2 Mili, Mili and Mittermeir

Mili, Mili, and Mittermeir [29] discussed the design and implementation of a software library, based on a representation of software components by means of formal specifications. Their formal specification is a pair, $(S, R)$, where $S$ is the space of the specification, represented by variable declarations; $R$ is a relation defined over $S$. The relation of a specification contains

all the input/output pairs that the specifier considers correct. Therefore, their specification is also called relational specification. They defined a *refinement* ordering between relations over the same space, and then extended the refinement ordering to specifications. Like Jeng and Cheng's *generality* relation [27], their refinement ordering is intended to capture the behavioral requirements of reusability. Given that the refinement ordering is a partial ordering relation, they proceeded to prove that the refinement ordering has lattice properties. Such a lattice structure is used as a basis for ordering software components in their software library, and this ordering, in turn, is used to guide the retrieval of components in the library.

They discussed two types of retrieval operations: *exact retrieval* and *approximate retrieval.* The former is used to find components that are correct with respect to the query specification, whereas the latter seeks to find components that can be modified to satisfy the query specification with minimal effort. Given a query specification, a component is considered an exact match to the specification if and only if the specification of the component is a refinement of the query specification. Given the refinement-based lattice organization of software components in their software library, the search process starts with matching the query specification against the maximal nodes of the lattice, and continues to the descendents of those nodes that are found to be refined by the query specification. Given that only relevant components in the library are inspected, the performance of component retrieval is improved.

With respect to approximate retrieval, they consider a component to be an *optimal approximate match* if and only if it maximizes the functional information that it has in common with the query specification. Based on the formal specification, a predicate is defined to determine if a component maximizes the *meet* with the query specification. Through the

use of the lattice operator *meet*, this criterion is made computable, and implemented in their library system.

### 3.1.3 Zaremiski and Wing

Zaremski and Wing [62] proposed signature matching as a mechanism for retrieving software components from a software library. Given the signature information of software components, they define *signature matching* as the process of determining when a library component "matches" a query. They considered both functions and modules, and thus have two kinds of matching: function matching and module matching. The signature of a function is given by its type. For a module, its signature is a multiset of user-defined types and a multiset of function signatures.

In order to discuss various function matching criteria in a unified framework, they defined a generic form of function match as follows:

$$M(t_l, t_q) = T_l(t_l) R T_q(t_q)$$

where $t_l$ is the type of a function from a component library, $t_q$ is the type of a query, $T_l$ and $T_q$ are transformations (e.g. reordering) and $R$ is some relationship between types (e.g. equality).

By varying $T_l$, $R$, and $T_q$, they defined various signature matches. When $T_l$ is a sequence of variable renaming, $T_q$ is the identity function, and $R$ is the type equality (=) relation, the match $M$ is called an *exact match*, that is, two function types match exactly, if they are equal modulo variable renaming. Relaxed matches are obtained by defining $R$ to be a partial order on types. Partial ordering is defined based on the "generality"of the types using variable substitution. A type $t$ is more general than another type $t'$ if $t'$ is the result of

a (possibly empty) sequence of variable substitutions applied to $t$. They defined two relaxed matches: *generalized match* and *specialized match.* A library function matches a query using *generalized match* if the type of the library function is more general than the query's type. Conversely, the query can be said to match the library function using *generalized match.* When $T_l$ or $T_q$ is varied, another type of match, *transformation* matches, are obtained. They defined two transformation matches using *uncurry* and *reorder* transformations, respectively. They also discussed composite matches obtained by composing the aforementioned matches.

Module matches are defined based on function matches. One new dimension to module matching is to establish a mapping between functions of two modules under consideration. They also discussed both *exact* and *relaxed* module matches, as well as composition of module matches.

Later they extended their approach to consider behavioral specifications of components [30]. They summarized several types of specification matches to capture different behavioral relations between components. They used the Larch specification language for the context of their discussion. Similar to their approach to signature matching, they defined a generic framework to discuss various specification matching criteria. They also discussed the relationships between these matching criteria, and established a lattice of them.

### 3.1.4 Fischer, Kievernagel and Snelting

Fischer, Kievernagel, and Snelting [63] presented a hybrid approach to component retrieval that filtered a library space in three steps: First, signature matching is used to reduce the search space; then model checking techniques are employed to further narrow the search space; finally theorem proving techniques are used to find the semantically "right" components.

### 3.1.5 Discussion

Components considered in the above approaches are generally fine-grained, such as functions or procedures. Modules that correspond to those found in modern programming languages are also considered by Jeng and Cheng, and Zaremski and Wing. These modules are typically a collection of individual functions, rather than architectural components that we address in this research.

A key issue in specification based software reuse is to define a specification matching criterion by which the reusability of an existing component for fulfilling a query specification can be determined. While a number of formal specification based criteria have been proposed in those aforementioned projects, and their usefulness have individually been argued in one way or another, there does not exist a general approach to reason about the connections between a matching criterion and its usefulness for determining reusability. Due to the lack of such a general approach, some problems regarding specification matching cannot be solved in an efficient way. For example, given a specification match that happens to be suitable for determining reuse, it is still an open question as to whether there exist *better* specification match. Intuitively, one specification match is *better* than another if it can identify reusable components for a given query that the other match fails to identify. One of the contributions of this research is to develop a general framework to evaluate various specification matches, and thus enable us to identify the "best" matches with respect to determining reusability.

## 3.2 Component-oriented programming

Software development productivity heavily relies on programming languages. From machine code, assembly languages, to early high-level languages, to the 1970s' structured languages, and to recent object-oriented languages, with the advent of each generation of programming languages, the granularity of entities embodied in a programming language is increased. The mechanisms of interaction between these entities are becoming richer and more complicated, thus yielding improved productivity and quality. Developing software by wiring components, rather than programming statements, has long been a goal. From early programming-in-the-large, to recent megaprogramming, while under different names, the goal is the same: programming with components.

### 3.2.1 MIL

The adoption of hierarchical decomposition and modularization in software development in the 1970s resulted in the need to integrate independently developed subsystems or modules into a complete system. DeRemer and Kron [64] were among the first to realize that assembling a system from modules was essentially a different task from programming. They proposed a module interconnection language (MIL) [64, 65] as a tool for module assembly, the so-called programming-in-the-large. An MIL provides a set of formal grammar constructs to specify modules of a system and how they fit together to implement the system's functionalities. Early MILs use names as the sole representation of a service (or resource in the context of MILs). The binding between provided and required services are simply based on name matching. Recent MILs have been enhanced to include some semantic information in the module interface specification. For example, LILEANNA, a MIL designed specifically for Ada, provides a means to specify module semantics in terms of predicate calculus [66].

### 3.2.2 Megaprogramming

Megaprogramming [67] is a technology for programming with large modules termed megamodules that interlink the functionality of services provided by large organizations. Wiederhold, Wegner and Ceri [68] proposed a megaprogramming language that provides the glue for joining together computations spanning several megamodules. More specifically, the language permits invoking megamodules, supplying to them and extracting from them data and parameters, controlling their execution, transferring and transducing data between megamodules, and achieving asynchrony and parallelism of computations. The key feature of the proposed language is to replace the conventional **CALL** statement that exerts control and provides communication between software components with three distinct statements: **SUPPLY**, **INVOKE**, and **EXTRACT**. **SUPPLY** provides global arguments from a megaprogram to the megamodule. **INVOKE** causes the megamodule to process these arguments and prepare results. **EXTRACT** allows the megaprogramming to extract results from the megamodule. Composition and interaction of megamodules are realized through these three statements together with other control structures. Megaprograms written using the megaprogramming language can be compiled and optimized to reduce the execution cost.

### 3.2.3 Discussion

As a natural evolution of regular programming methods, component-oriented programming aims to develop software by manipulating coarse-grained components instead of statements or functions. Both the above two approaches have developed mechanisms to describe and manipulate coarse-grained components. However, as with regular programming, component-oriented programming is mainly an implementation method. It does not address architectural design that maps requirements to architectures. On the contrary, in

this research, we consider component assembly as one part of component-based software development, and address relevant issues in the context of an integrated framework.

## 3.3 Architecture-based Development

Several projects have explored the roles of software architectures in component-based software development.

### 3.3.1 Andersen Consulting's CBSE project

Researchers of the Component-Based Software Engineering (CBSE) project [69, 70, 71, 72] at Andersen Consulting's Center for Strategic Technology Research (CSTaR) are developing techniques for component integration. The core of their approach is their Architectural Specification Language (ASL). Focusing on formally capturing architectural concerns of systems, ASL comprises three sublanguages: Interface Specification Language (ISL), Glue Specification Language (GSL), and Configuration Specification Language (CSL). ISL is an extension to the CORBA [1] interface definition language (IDL) [73]. In ISL, an interface of a component describes the operations it supports, services it requires, and other external characteristics. In addition to supporting the CORBA IDL, ISL has a richer set of semantic constructs that support descriptions of pre/post conditions of operations, and invariants and protocols of interfaces. GSL specifies interconnections between components. An interconnection binds a component that requires a particular service to another component that provides the service. Interconnections are specified via instances of interfaces, bindings that establish direct references between required and provided services, and connectors that

[1] CORBA stands for Common Object Request Broker Architecture, which is a distributed component architecture developed by Object Management Group.

serve as adaptors or mediate complex patterns of interactions between components. ASL separates component interfaces from their implementations. CSL is used to attach implementations to components and describe platform-specific attributes of the implementations.

In order to support architecture design, a graphical environment has been developed. This design environment supports visual presentation and construction of architecture specifications and component interface specifications. In addition, a library management system based on the *faceted classification* scheme [74] has been associated with the design environment to assist in the storage and retrieval of reusable assets: component specifications, their implementations, architecture specifications, etc. In order to verify design decisions specified in ASL, two types of analysis are supported: interconnection verification and configuration analysis. The objective of interconnection verification is to ensure the *compatibility* between components involved in the connection. Both syntactic and semantic checking are conducted based on the specifications of services. System configuration properties, such as distribution and performance, are analyzed using domain knowledge and general design principles and constraints to ensure that configuration decisions be correct. Finally, the design environment supports *system packaging*, the derivation of procedures from ASL specifications for generating an executable system.

### 3.3.2 Dellarocas

Dellarocas [75, 76] proposed a coordination theory-based approach to component integration. Similar to the recognition of connectors as "first-class" elements in software architectures [32], the key perspective in Dellarocas's approach is to treat interdependencies between software components as a distinct design problem, orthogonal to that of representing and implementing the core functional pieces of an application. Based on this perspective,

a taxonomy of software interconnection problems and solutions was developed to catalog common software interconnection dependencies and sets of alternative coordination protocols to manage them. The taxonomy uses multi-dimensional design spaces to classify both dependencies and coordination protocols. It starts by identifying a small number of generic dependencies. For each generic dependency, it defines a number of design dimensions that can be used to further specialize the relationship. These dimensions form a design space that contains different specializations of a given dependency. Each point in the design space defines a different specialized dependency type. For each dependency, a few generic coordination protocols for managing the dependency are identified. It also defines a design space that contains several related specialized versions of these coordination protocols.

Three most generic dependency families in the taxonomy are: *flow dependencies*, *sharing dependencies*, and *timing dependencies*. Flow dependencies represent relationships between producers and consumers of resources. Sharing dependencies capture relationships among consumers who use the same resource or producers who produce the same resource. Timing dependencies describe constraints on the relative flow of control among a set of activities. Mutual exclusion is an example of timing dependencies. For each family of dependencies, a generic model has been created for classifying dependencies in the family. A framework for designing coordination protocols for these dependencies has also been established. For example, the flow dependencies comprise three subdependencies: *usability*, *accessibility* and *prerequisite*. Design dimensions for usability include: Who is responsible for ensuring usability? When are reusability requirements fixed? For each design dimension, a set of design alternatives have been identified.

The taxonomy provides a handbook for designing interconnection protocols for managing dependencies between software components. A prototype system has been developed to support the use of the taxonomy in integrating software components.

### 3.3.3 Regis and Darwin

Regis [77] is a constructive environment for developing distributed programs. It embodies a constructive approach to the development of programs based on separating program structure from communication and computation. Regis emphasizes the construction of programs from multiple parallel computational components that cooperate to achieve the overall goal. It supports component composition through the architectural description language, Darwin. As a declarative binding language, Darwin is used to define hierarchical compositions of interconnected components. Component implementation and distribution are dealt with orthogonally to system structuring.

As a configuration language, Darwin allows programs to be constructed from hierarchically structured configuration descriptions of the set of component instances and their interconnections. In Darwin, components interact with other components through providing or requiring services. The Darwin component interface specifies the set of services required and provided by a component together with the types of these services. As a declarative language, composite components are defined by declaring both the instances of other components they contain and the bindings between these components. A binding associates a service required by one component with the service provided by another. The Darwin compiler checks that bindings are only made between required and provided services that are compatible. Darwin also supports dynamic configuration. That is, the structure of a system can change as execution proceeds.

### 3.3.4 Discussion

As a design dimension, software architectures provide a high-level design space in which various alternatives for system organization can be explored and analyzed. As a design notation, software architectures may serve as guidance to further development. The three approaches described in this section display various uses of software architectures in addressing issues involving component-based software development. The CBSE project uses software architectures as its component composition language. Dellarocas explores the design space in component integration at architectural level. Darwin is a domain specific architectural language for developing distributed component-based applications. While both Dellarocas and Darwin address only certain aspects of component-based software development, namely integration and domain-specification, respectively, the Andersen Consulting's CBSE project has explored a variety of issues regarding the use of software architectures in implementing component-based software development. However, due to the fact that its ASL is tightly coupled with the CORBA platform, the design space of component compositions is limited. For example, while ISL extends the CORBA IDL to describe both the services provided and required by a component, the gluing (connection) mechanism is limited to the "definition/use" binding of services. Although it uses semantic information as an auxiliary in checking the validity of a connection, the CBSE project does not base its component evaluation on formal methods, nor does it investigate semantic foundations to component evaluation, as we do in this project.

## 3.4 Object-Oriented Framework

An object-oriented framework is a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact [78]. A framework provides a reusable context for components and can be customized for specific applications. Since object-oriented frameworks can be regarded as a specific type of software architecture, this category of work is a specialized case of architecture-based development that is based on object-oriented methodology.

### 3.4.1 Schappert, Sommerlad and Pree

Schappert, Sommerlad and Pree [79] described an industrial research project that attempts to provide automated support for software development based on object-oriented frameworks and prefabricated components. Software development consists of two complementary activities: the construction of frameworks and new components for functionality that are not currently available, and the composition and configuration of existing components.

As with recent work on software architectures that emphasize the interrelationships between software components, they studied component cooperation by using explicit structural information as a basis for their automation approach. They use *relations* to describe software component interrelationships. Three levels of representation for relations and components are distinguished: *visual*, which provides graphical manipulation and adequate presentation; *structural*, which contains information as a foundation for automation; and *code*, which is for (re)use of existing components.

Structural relations are used to extract mechanisms whose implementation is usually distributed in multiple software components. By describing the mechanism explicitly the functionality is isolated and can be reused in different contexts. In the meantime, the

structural relations encapsulating these mechanisms can also be reused in different contexts. A structural relation has three aspects: it declares the interface to be filled by the attending components; it describes in an abstract way the interrelationship, such as an inter-class call-graph; It provides a parameterized implementation of the mechanism to be realized by it, thus facilitates code generation. A structural relation can be a composition of other structural relations and components.

In order to facilitate the use of frameworks, they proposed an active guidance to the developer by an active cookbook presenting recipes that contain information and invoke tools to perform a development task. Frameworks are extended to include recipes for improving their (re)usability.

### 3.4.2 Discussion

Object-oriented framework promotes component-based software development by encapsulating interaction mechanisms among components and providing contexts for components to plug-in and function. Unlike other approaches to component-based software development, object-oriented framework is tightly coupled with object-oriented methodology. Components in this context are classes or objects; interaction mechanisms are those specific to object-oriented methods, such as aggregation, inheritance, method invocation, and so forth. In spite of these differences, object-oriented framework shares the common set of issues involving component-based software development: composition design, component evaluation, and integration, as exhibited by the project we described in this section. Therefore, the development of generic approaches to addressing these issues should be useful in the context of object-oriented framework.

# Chapter 4

# A Framework for Component Reuse

In this chapter, we describe a software reuse framework that emphasizes the assembly of components as a paradigm to software development [33]. Reuse issues, in particular those concerning component composition and integration, are addressed in an integrated fashion in the framework. This framework also provides a context for the remainder of this dissertation, and serves as a roadmap to our research.

## 4.1 Component-based Software Engineering

As a development paradigm, component-based software engineering is different from conventional ones in that it partitions software development into two distinctive engineering activities: one is component engineering, and the other is application engineering. Figure 4.1 shows the high-level view of component-based software engineering.

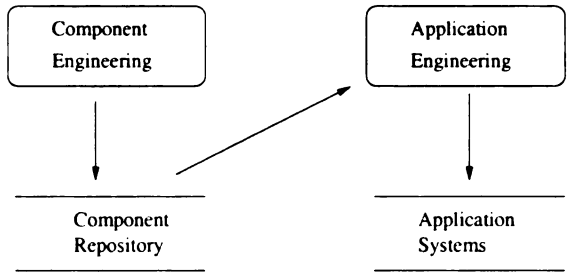



Figure 4.1: CBSD process high level view

*Component engineering* deals with the specification, development and brokerage of reusable components. Components can be extracted and/or reengineered from existing systems [80, 81, 82]. A more systematic and efficient approach is design for reuse. *Domain engineering* [83, 84, 85, 86] is an example of a design-for-reuse process that produces domain-specific reusable artifacts, including domain requirement models (specifications), architectures, and implementations. Figure 4.2 shows the activities of domain engineering and their results. As shown in Figure 4.2, domain engineering is the application of a typical

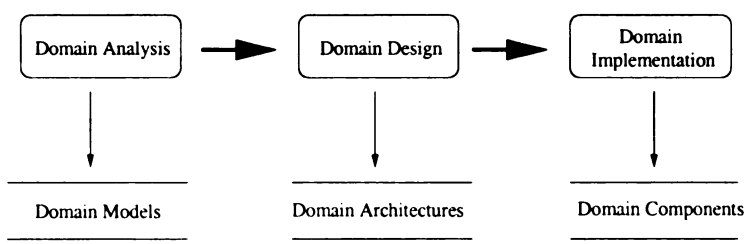



Figure 4.2: Domain engineering process

software process [40, 87] to a specific domain, instead of a particular application.

*Application engineering* focuses on the assembly of components to construct problem-oriented solutions. Figure 4.3 depicts the high-level view of application engineering. As

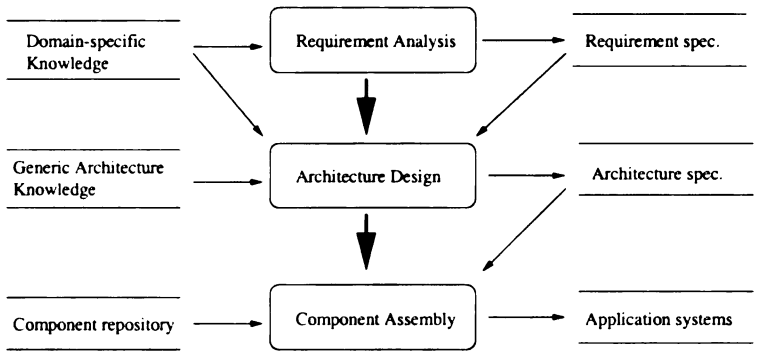



Figure 4.3: Application engineering process

with the *waterfall model* [40], the first step in constructing a software system is collecting and analyzing requirements. Domain specific knowledge such as domain models generated during domain analysis is reused to produce application-specific models and specifications. Architecture design groups functionality into component, and identifies the inter-relations between components. The domain specific architectures may serve as reference architectures for generating the application-specific architectures that embed the application-specific requirements. The generic software architecture knowledge, such as architectural styles, their constraints and performance properties, provide a foundation to architecture design. Given the architecture specifications as a framework, the next step in application development is to assemble components.

## 4.2 Architecture-based Component Assembly

An architecture is a collection of components, connectors, and a configuration of how the components should be integrated via connectors [88]. Figure 4.4 depicts the architecture-based assembly process of software components. In Figure 4.4, the polygons labeled by $I_i$ represent interfaces of components. The cubes represent existing components. The black bars labeled by $C_i$ indicate connectors. The dashed arrows represent an implementation relationship between an existing component and an interface.

Given an architecture specification, the assembly process consists of two steps. First, the existing components are retrieved, evaluated, and matched to the interfaces in the architecture specification.[1] Then an integration process is invoked to integrate these components, generating code as necessary for adapting components and/or implementing connections.[2]

With respect to component reuse, this architecture-based component assembly approach has a number of potential benefits: First, component and composition (i.e., architecture) specification can be planned in a unified framework that may facilitate the consideration of design information (such as architectural properties) in component retrieval and evaluation. Second, the component evaluation process may identify the conflicts between a requirement and an existing component, thus facilitating the automation of component adaptation. Third, as a mediator of interactions between components, the connectors provide a place for recording and resolving conflict assumptions between component interfaces. Furthermore, connectors prevent the changes made to a component from implicitly propagating across the entire system. Fourth, not only can components be reused, connectors

[1] If there are interfaces to which no existing component is matched, then we can construct them from scratch without affecting the other parts of the system.

[2] In reality, a component may simultaneously fulfill multiple interfaces. For convenience of description, in this manuscript we assume that each interface has a component that implements the interface, as shown in Figure 4.4.

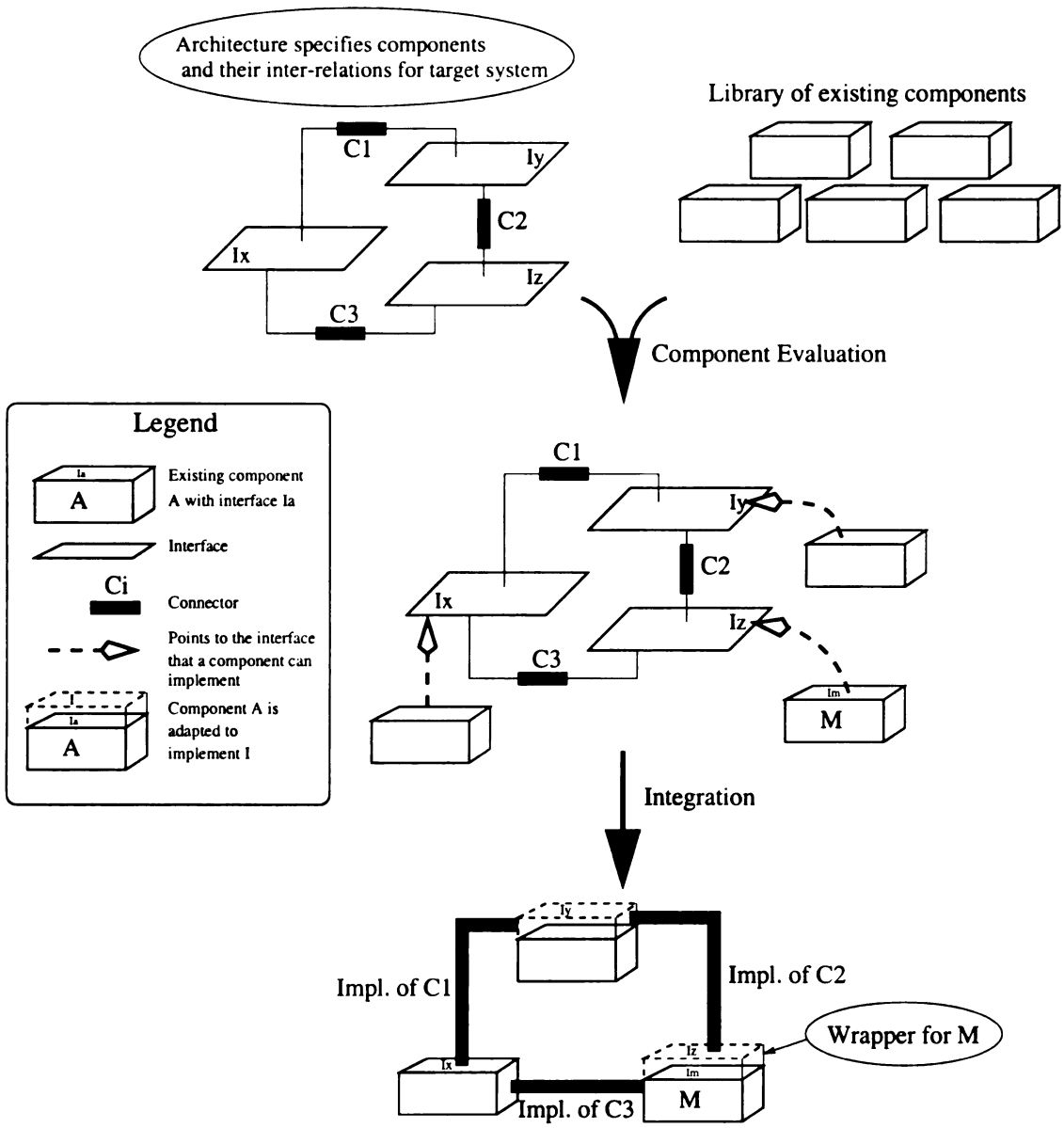



Figure 4.4: Architecture serves as a framework into which components are assembled

that encapsulate recurring interaction styles can also be reused, and their implementation can be automatically generated. Fifth, as illustrated in Figure 4.4, the adaptation of an existing component ($M$) may be localized by requiring the adapted component ($M$ with its wrapper) conform to the requirement $I_z$. This localization prevents the adaptation of $M$ from impacting the connectors, since the connectors can only "see" the interface $I_z$. Finally, a significant benefit comes from the maintenance and evolution of assembled systems, since software architectures, when considered as design specifications, provide a clear description of these systems and support design maintenance [89].

## 4.3 Issues Addressed In This Research

In this research, we develop techniques to solve the following crucial difficulties in the implementation of the aforementioned component reuse framework.

- A specification framework that integrates facilities for describing both structural and functional aspects of a target system.

  Component-based engineering emphasizes the composition of components, that is, the structural properties of a system in terms of its constituent components. On the other hand, in order to enable the effective retrieval and evaluation of existing components for reuse, a certain degree of precision in specifying requirements and/or component functionalities is required. In this research, we develop a flexible framework that integrates software architecture description techniques with traditional formal specification languages.

- Component evaluation methods.

  In order to automate the evaluation process of software components, methods based

on formal description techniques need be developed. The essence of an evaluation method is to capture the concept of *reusability* while keeping low the computational cost. Depending upon the degree of reusabillity captured, the computational cost varies. While the easiest way is based on keyword matching, specification matching based approach captures reusability in a far more precise way. Given the availability of automated proof techniques (and tools) for resolving proof obligations, component evaluation based on specification matching is also a feasible approach. In this research, we first lay a theoretical foundation on the connection between specification matching and reusability. Then we develop a specification matching based approach to evaluate software components. One significant result of our approach is applicability to coarse-grained, architectural components.

# Chapter 5

# Component Interconnect Model

By specifying how a system is composed of a collection of components, software architectures provide the link from requirements to implementation and play a critical role in the development and evolution of software systems. Unfortunately, conventional architectural description techniques are generally too high-level and thus cannot provide sufficient information to evaluate and select appropriate components. In this chapter, we describe a framework that integrates mechanisms for describing both structural and functional aspects of a target system.

## 5.1 Architectural Model

Our specification framework is based on the architecture model described in Section 2.2. That is, an architecture is a configuration of two kinds of distinct, identifiable elements: *components* and *connectors* [38, 50]. Components are the locus of computation and state, whereas connectors encapsulate the interaction protocols among components. Components interact with each other through exchanging resources according to protocols defined by connectors. A resource may be a computation (function), an abstract data type, a data

item, an event, or a bundle of the above resources. The behaviors that a component may exhibit in constructing a system are modeled by *ports*. A port is an interaction point through which a component can provide resources to or require resources from its environment. In order to use the same component specification language to cover the range of abstractions for entities that may be considered as components, we use a broader definition for components than what may typically be used. Examples of components include subroutines, classes, a library of subroutines or classes, and any constituent part of an application. A component interacts with its environment only through its ports. The set of all ports that a component has constitute the essential part of the component's interface. The interface of a component may also contain global properties regarding the behavior of the component. The interface of a component provides a vehicle at an abstract level for communicating the capabilities of a component and the way the component delivers its capabilities among component specifiers, implementers, and users. It should be noted that the interface in this context differs from the conventional definitions like those used in the CORBA IDL [73] in several ways. First, it states not only the capabilities (resources) that a component can provide, but also assumptions (resources) that a component requires. Second, it specifies the ways a component may communicate with its environment and deliver its capability.

While a component embodies computation and/or data, a connector encapsulates interaction protocols that govern the participants of those protocols. In the same way as a component is modeled as a collection of ports, a connector is modeled as a set of placeholders each of which is for a participant of the protocols encapsulated by the connector. These placeholders, also called *roles*, are the only points through which a connector relates to its environment. The collection of roles that a connector has, together with optional global properties regarding the behavior of the connector, constitute the interface of the

connector. Similar to a component interface, the interface of a connector serves as a vehicle at an abstract level for communicating the capabilities of the connector among connector specifiers, implementors, and users.

A system is composed by configuring, i.e., plugging-in the ports of components to the roles of connectors. The intuitive implication of "plugging-in" a port to a role is that the component will participate in the protocol as the role through the port. Figure 5.1 depicts the elements and their relationships in the architectural model.

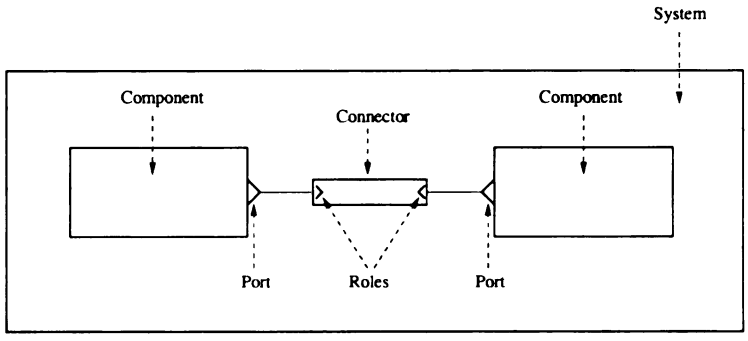



Figure 5.1: Architectural elements

## 5.2 Integrating Behavioral Specifications

In order for software architectures to be useful, architectural elements need to be specified in great detail. A flexible approach to specifying architectural elements is to identify properties that characterize an architectural element. Each type of architectural element has a set of properties whose values together define an instance of this element.

### 5.2.1 Component

As discussed above, the interface of a component is characterized by a collection of ports and global constraints. While ports are the interaction points that a component has with its environment, global constraints of a component specify the relations among ports. One important type of constraint for the purpose of reuse is the dependencies of a port upon others. Intuitively, a port providing resources may assume the availability of certain resources that are either provided or required by other ports. The dependencies between ports capture this intuition and provide a finer-grained description of the relations between capabilities and assumptions of a component. Another type of constraint involves the overall behavior of the component, such as the transformation of data that flows through different ports. We illustrate the specification of constraints in detail later in this Chapter.

In addition to an interface, a component may have an implementation. The implementation of a component should conform to its interface in that the implementation implements the capabilities and delivers them in the way specified in the interface. A formal definition of conformance is given in Chapter 7. A component without an implementation is known as an *abstract* component or simply an *interface*. Each *existing* component must have an implementation. The implementation of a component may be in a variety of forms: executable, source code/file, object code, object library, OS built-in facilities, other existing components, or as a composite component implemented as another architecture. The specification of component implementation is characterized by a set of implementation-related properties that specify implementation method, language, location, platform, architectural mapping, and so forth.

### 5.2.2 Port

A port defines a behavior that a component may exhibit in constructing a system. Therefore a port is the main place where we integrate behavioral specifications. (Another place to integrate behavioral specification is the constraints' specification of a component as mentioned previously.) A port is characterized by three properties:

- *Resource*: the resource that is exchanged through the port. It may be a **computation**, an **abstract data type (ADT)**, a **data item**, a **data stream**, an **event**, or a bundle of the above resources.

- *Direction*: the direction that a resource can flow through the port. A resource can flow in (IN) or flow out (OUT) or both (IO).

- *Connectivity*: the number of connections in which the port may simultaneously participate. It is specified by keywords: **single** representing only one connection is allowed; **multiple** meaning multiple connections are permitted; or a number N for the specific number of possible connections.

The resource that flows through a port is the major property that characterizes the port. In the following we discuss the specifications of various types of resources.

#### Resources

The basic form of computation is a function, which is provided as a basic construct by all modern programming languages. Syntactically, a function has a name and a signature specifying the types of parameters and return value, if any. The behavior of a function, that is, what a function can accomplish, can be specified using formal logic. In this research,

we use a Larch-styled method to specify the functional aspects of software components. Figure 5.2 shows the behavioral specification for computing the square root of an integer.

```
FUNC sqrt(int n) return int {
  uses Integer(int for Int);
  modifies  result;
  requires  n ≥ 0;
  ensures   result' × result' ≤ n ∧
            n ≤ (result'+1) × (result'+1);
}
```

Figure 5.2: Specification of function **sqrt**

In Figure 5.2, **result** represents the return value of the function. As in the Larch interface language for C, LCL [26], a variable superscripted with ^ and ' refers to the values of the variable before and after the execution of a function. A variable without any decorations also refers to its value before the execution of a function. In order to facilitate reasoning about behavioral specifications, theories about typed variables and constants appearing in a specification is needed. The *uses* clause serves this purpose by including relevant LSL traits.

A data item has a name and belongs to a type. A type can be modeled as an algebra by the Larch shared language [26]. For an abstract data type (ADT), we specify its behaviors by a set of function specifications that each specify an externally observable behavior of an object of the type. Figure 5.3 shows the specification for a generic stack. When specifying an ADT, we use **self** to represent the object of the ADT [90].

A data stream is a sequence of data items of the same type. A data flow is specified by giving the type of its elements. For example, a character stream is specified as the following:

```
ADT stack<T> {
    uses Integer(int for Int),
         Stack(T for E, stack for C);

    FUNC stack() return void  {
         modifies self;
         ensures self' = empty;
       }
    FUNC push(T x) return void {
         modifies self;
         ensures self' = push(x, self^);
       }
    FUNC pop() return void {
         requires ~isEmpty(self^);
         modifies self;
         ensures  self' = pop(self^);
       }
    FUNC top() return T {
         requires ~isEmpty(self);
         ensures  result' = top(self);
       }
    FUNC isEmpty() return Bool {
         ensures  result' = isEmpty(self);
       }
    FUNC size() return int {
         ensures result' = size(self);
       }
  }
```

Figure 5.3: Specification of **stack**

STREAM acstream : char;

An event is represented by a name. An event occurs if and only if certain conditions are met. A predicate can be used to specify the conditions. For example,

EVENT minute_tick {PRED: second_tick = 60; }

specifies an event **minute_tick** that will happen when the count of second ticks is 60.

A collection of resources can be bundled together as a whole to be provided or required by a port. Such a bundling is specified as below:

RBUNDLE name { {ResSpec}* }

where RBUNDLE indicates that the specified resource is a bundle of resources, and {ResSpec}* represents multiple resources specifications.

### 5.2.3 Examples

Our first example considers the specification of the middle layer component in a three-layered 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 5.4 describes the interface of the middle layer component, where functions* **read** *and* **write** *are provided for the use of the top layer component through two ports with the same name, respectively. (It is not mandatory that the resource of a port takes the same name as the port. However, for simplicity, we use this convention.) The data object* **x** *of type*

*dtModel is imported through port x from the bottom layer component. In order for the component dataAccess to provide the services read and write, data object x is required, thus both ports read and write depend on port x.*

Our next example specifies a filter component that counts the number of words.

**Example 2** *Figure 5.5 depicts the specification of component wordcounter. Figure 5.5 illustrates the specification of overall behavior of a component through constraints. Several primitive functions are used in the behavior specification. Function len is defined over a stream that returns the length of a stream. Given a stream s, s[i] returns the (i + 1)-th element of s. As shown in Figure 5.5, the behavioral constraints state that the number of words input through port ipw is equal to the first and only element that is output through port oc. This example also illustrates the specification of implementation. As shown in Figure 5.5, component wordcounter is implemented by the Unix command wc.*

### 5.2.4 Typing Architectural Elements

As shown in previous examples, architectural elements exhibit commonalities. Types are introduced to capture those commonalities. Figure 5.6 shows the definitions of several typical port types. Each port type specifies three common properties shared by all ports of this type: *resource type*, *resource flow direction*, and *connectivity*. The type of a port encapsulates the behavior pattern that a component can exhibit through the port. For example, a component may define and export a procedure to its environment through a *ProcDef* port, whereas an *InStream* port in a filter means that the filter will obtain a stream of data from the port. By associating a type with a port, we can simplify the specification of a port by omitting those properties already implied in the type.

```
COMP dataAccess {
    uses Table(dtModel for Tab, int for Ind, int for Val);
    PORT x {
        DIRECTION : IN;
        CONNECTIVITY : SINGLE;
        DATA x : dtModel WITH BEHAVIOR {
           FUNC exists(int recNo) return bool {
                 ensures result = recNo ∈ x; }
           FUNC insert(int recNo, int val) return void {
                 requires ~(recNo ∈ x^);
                 modifies x;
                 ensures  recNo ∈ x' ∧ lookup(x', recNo) = val; }
           FUNC query(int recNo)  return int {
                 requires recNo ∈ x;
                 ensures  result = lookup(x, recNo); }
           FUNC update(int recNo, int newVal) return void {
                 requires recNo ∈ x^;
                 modifies x;
                 ensures  lookup(x', recNo) = newVal; } }
       }
    PORT read {
        DIRECTION : OUT;
        CONNECTIVITY : MULTIPLE;
        FUNC read(int recNo, int recVal) return bool  {
               modifies recVal;
               ensures result = recNo ∈ x ∧
                      (if result then recVal' = lookup(x, recNo)); }
       }
    PORT write {
        DIRECTION : OUT;
        CONNECTIVITY : MULTIPLE;
        FUNC write(int recNo, int recVal) return void {
           modifies x;
           ensures recNo ∈ x' ∧ recVal = lookup(x', recNo); }
      }
    CONSTR { DEPENDENCE { {read, write} : x } }
}
```

Figure 5.4: Specification of dataAccess interface

```
COMP wordcounter {
    PORT ipw {
         DIRECTION : IN;
         CONNECTIVITY : SINGLE;
         STREAM ipw : word;
       }
    PORT oc {
         DIRECTION : OUT;
         CONNECTIVITY : SINGLE;
         STREAM oc : int;
       }
    CONSTR {
         BEHAVIORCSTR { PRED: len(ipw) = oc[0] ∧ len(oc) = 1; }
       }
    IMPLEMENTATION {
         method : executable;
         platform : Unix;
         name : ''wc'';
         location : ''.'';
      }
}
```

Figure 5.5: Specification of component wordcounter

Similarly, there are recurring patterns exhibited in the way that components deliver their functionalities. We use component types [50] as an abstraction to capture these recurring patterns. Specifically, component types are intended to capture the architectural properties. Each component type supports a specific set of port types. For example, the type **filter** specifies that a component can only have ports of type **InStream** and **OutStream**, and thus must be used in a pipe-and-filter system [32]; whereas a **Module** may have ports that provide or require computation, data, or abstract data types, and thus can be used in main program/subroutine system.

```
defporttype ProcDef {
            restype : FUNC;
            direction : OUT;
            connectivity : MULTIPLE
        }
defporttype ProcInvoc {
            restype : FUNC;
            direction : IN;
            connectivity : SINGLE
        }
defporttype DataDef {
            restype : DATA;
            direction : OUT;
            connectivity : MULTIPLE
        }
defporttype DataUse {
            restype : DATA;
            direction : IN;
            connectivity : SINGLE
        }
defporttype ADTDef {
            restype : ADT;
            direction : OUT;
            connectivity : MULTIPLE
        }
defporttype ADTUse {
            restype : ADT;
            direction : IN;
            connectivity : SINGLE
        }
defporttype EventAnnounce {
            restype : EVENT;
            direction : OUT;
            connectivity : MULTIPLE
        }
defporttype EventListen {
            restype : EVENT;
            direction : IN;
            connectivity : SINGLE
        }
defporttype InStream  {
            restype : STREAM;
            direction : IN;
            connectivity : SINGLE
        }
defporttype OutStream {
            restype : STREAM;
            direction : OUT;
            connectivcity : SINGLE
        }
```

Figure 5.6: Definitions for common port types

Figure 5.7 depicts the interface specification of component *wordcounter* with typing information.

```
COMP wordcounter : Filter {
    PORT ipw : InStream{
         STREAM ipw : word;
       }
    PORT oc : OutStream{
         STREAM oc : int;
       }
    CONSTR {
         BEHAVIORCSTR {PRED: len(ipw) = oc[0] ∧ len(oc) = 1; }
       }
 }
```

Figure 5.7: Interface specification of component wordcounter with typing information

### 5.2.5 Connector

The roles of a connector together with global properties regarding the behavior of the connector constitute the interface of the connector to its environment. Global properties are usually rules and constraints, that is, protocols, governing the interactions between the roles of a component. For example, a *procedure call* as a connector has two roles: procedure definer (callee) and procedure user (caller). The typical protocol of a *procedure call* as implemented by most programming languages requires that the callee and the caller have the same (resource) name and use synchronous communication. More complicated protocols, such as remote procedure call (RPC), pipeline and implicit invocation, can be encapsulated in a connector. A connector can be specified by its roles and the protocols governing the roles. Similar to components, connectors can be specified and analyzed based

on formal theory and notations. We previously have used the process specification language LOTOS to precisely capture and analyze connector properties [34]. Some architectural description languages, such as Wright [42, 43], focus on formalizing connectors. However, in this research, since the use and reuse of components are our central issues, the formal aspects of connectors are beyond our scope.

Like component types, connector types are introduced to capture recurring component interaction styles. A connector type embodies the essential properties of the roles and protocols of a connector of this type. One important property of a role is the constraints imposed on the port that may be plugged-in the role. We define the domain of a role to be the set of port types whose instances can be plugged into the role. Figure 5.8 shows the definition of several typical connectors. As shown in Figure 5.8, the definition of a connector specifies the names of roles and their domain as well as constraints applied to the roles.

## 5.3 The Language

Based on the above framework that integrates architectural and functional specifications, we have developed an architectural description language. The examples previously shown in this chapter are written in this language and exhibit certain aspects of the ADL. A concise yet complete BNF description of the syntax of this ADL is given in Appendix A.

```
defconntye CallProc {
    Roles { Definer : {ProcDef};
            Caller  : {ProcInvoc}
          }
    Constraints {
            Definer << Caller
               #Definer is more specific than caller
          }
  }

defconntype AccessData {
    Roles { Definer : {DataDef};
            User    : {DataUse}
          }
    Constraints {
            Definer << User
               #Definer is more specific than caller
          }
  }

defconntype UseADT {
    Roles { Definer : {ADTDef};
            User    : {ADTUse}
          }
    Constraints {
            Definer << User
               #Definer is more specific than caller
          }
  }

defconntype Pipe {
    Roles { Source : {InStream};
            Sink   : {OutStream}
          }
  }

defconntype EventProc {
    Roles { Announcer : {EventAnnounce};
            Listener  : {EventListen}
          }
  }
```

Figure 5.8: Definitions for common connector types

# Chapter 6

# A Semantic Foundation for Specification Matching

Evaluating the reusability of a library component for satisfying a query is a central task for software reuse. A formal methods-based approach has the advantage of being precise and amenable to automation. Previous work [28, 30, 91] has proposed specification matching as a means to component retrieval and evaluation. These approaches have some difficulties: First, the connection between a specification match and its suitability for determining reusability is based on intuition, rather than a formally defined foundation; Second, they only consider functional aspects of components. In this chapter, we establish a semantic foundation for specification matching for reuse. In Chapter 7, we extend specification matching to architectural components [92].

## 6.1 Introduction

Determining the behavioral relationship between software components is a central task for many software engineering activities, such as reuse, maintenance, and object-oriented subtyping. With the ever increasing adoption of formal methods, specification matching [93, 27, 28, 30] has been proposed as a means to evaluate component relations at an abstract level. Specification matching identifies behavioral relationships between software components by checking the logical relations between specifications. Among the possible behavioral relations, one of particular interest is *behavioral refinement*, that is, one component provides all the behavior that another component does. In software reuse, this means that one existing component can be (re)used where a new component is needed. In software maintenance, it means that one component can be substituted for another one without changing the behavior of the whole system. In object-oriented subtyping, the refinement relationship between methods of two classes is an essential requirement for one class to be a behavioral subtype of another [94]. In this chapter, in order to focus our discussion, we study *behavioral refinement* in the context of software reuse. We have concentrated on the software reuse application since it is not only the focus of this research, but also has been the focus of most of the specification matching work [93, 27, 28, 30, 59, 91, 95, 29, 33]. In software reuse, a key issue is to determine if an existing component is reusable for implementing a given query specification.

A number of specification matching criteria have been proposed to capture the notion of *behavioral refinement*, or *reusability* in the context of software reuse. One widely used formal specification method is based on Hoare's axiomatic approach [96]. An axiomatic specification of a component (procedure) $C$ is a 2-tuple of predicates, $\langle C_{pre}, C_{post} \rangle$, where $C_{pre}$ specifies the precondition, and $C_{post}$ specifies the postcondition of the procedure. In

the following discussion, consistent with terms of software reuse, $Q$ represents a query specification, $\langle Q_{pre}, Q_{post} \rangle$, and $A$ is a library component specification, $\langle A_{pre}, A_{post} \rangle$. Zaremski and Wing [30] defined *exact pre/post* match, $M_{exact-pre/post} : (Q_{pre} \leftrightarrow A_{pre}) \wedge (A_{post} \wedge Q_{post})$, and *plug-in* match, $M_{plug-in} : (Q_{pre} \rightarrow A_{pre}) \wedge (A_{post} \rightarrow Q_{post})$. Penix and Alexander [91], and Schumann and Fischer [95] use a more *relaxed plug-in* match in their component retrieval work, $M_{relaxed-plug-in} : (Q_{pre} \rightarrow A_{pre}) \wedge (Q_{pre} \wedge A_{post} \rightarrow Q_{post})$. By defining the characteristic predicate of a component (or specification) as $A_{pre} \rightarrow A_{post}$, Jeng and Cheng [93, 27, 28, 59], Zaremski and Wing [30] discussed *exact* and *generalized predicate* matches, $M_{exact-pred} : (A_{pre} \rightarrow A_{post}) \leftrightarrow (Q_{pre} \rightarrow Q_{post})$ and $M_{gen-pred} : (A_{pre} \rightarrow A_{post}) \rightarrow (Q_{pre} \rightarrow Q_{post})$. Zaremski and Wing [30] also suggest an alternative definition of characteristic predicate, $A_{pre} \wedge A_{post}$, and therefore obtain a different form of predicate matches. In defining behavioral subtyping, America [97] and Liskov and Wing [98] use *plug-in* match as their method rule that governs the behavioral relationships between methods of two classes (or objects). Dhara and Leavens [99] later refine their method rule as *guarded generalized predicate* match, $M_{guarded-gen-pred} : (Q_{pre} \rightarrow A_{pre}) \wedge ((A_{pre} \rightarrow A_{post}) \rightarrow (Q_{pre} \rightarrow Q_{post}))$. Table 6.1 lists some of those specification matches defined in the recent literature [28, 30, 91, 95, 97, 98, 99].

Behavioral refinement has been an important concept in formal programming methodology [100, 101, 102, 103], particularly in refinement calculus [104, 105, 106, 107]. In refinement calculus, programming is a step-by-step development of specifications into (executable) code through the application of refinement laws. A refinement law defines a correctness-preserving refinement relationship between two programs. (In refinement calculus, the term "program" is used polymorphically to represent (abstract) specifications,

| Match | Definition |
|---|---|
| $M_{exact-pre/post}$ | $(Q_{pre} \leftrightarrow A_{pre}) \wedge (A_{post} \leftrightarrow Q_{post})$ |
| $M_{plug-in}$ | $(Q_{pre} \rightarrow A_{pre}) \wedge (A_{post} \rightarrow Q_{post})$ |
| $M_{plug-in-post}$ | $(A_{post} \rightarrow Q_{post})$ |
| $M_{weak-post}$ | $A_{pre} \rightarrow (A_{post} \rightarrow Q_{post})$ |
| $M_{relaxed-plug-in}$ | $(Q_{pre} \rightarrow A_{pre}) \wedge ((Q_{pre} \wedge A_{post}) \rightarrow Q_{post})$ |
| $M_{exact-pred}$ | $(A_{pre} \rightarrow A_{post}) \leftrightarrow (Q_{pre} \rightarrow Q_{post})$ |
| $M_{gen-pred}$ | $(A_{pre} \rightarrow A_{post}) \rightarrow (Q_{pre} \rightarrow Q_{post})$ |
| $M_{exact-pred-2}$ | $(A_{pre} \wedge A_{post}) \leftrightarrow (Q_{pre} \wedge Q_{post})$ |
| $M_{gen-pred-2}$ | $(A_{pre} \wedge A_{post}) \rightarrow (Q_{pre} \wedge Q_{post})$ |
| $M_{guarded-gen-pred}$ | $(Q_{pre} \rightarrow A_{pre}) \wedge ((A_{pre} \rightarrow A_{post}) \rightarrow (Q_{pre} \rightarrow Q_{post}))$ |

Table 6.1: Various specification matches

(executable) codes, and intermediate forms between the two.) Although most of refinement laws involve the refinement of a specification to a code segment, several refinement laws only involve specifications, such as the *weaken precondition* and *strength postcondition* laws, which together are actually the *plug-in* match shown in Table 6.1.

The usefulness of a specification match $M$ for determining reuse lies in the following assumption:[1]

> *Given a query specification $Q$ and a library component $A$, if $M(A, Q)$ holds, then $A$ can be reused for implementing $Q$.*

The validity of the above assumption depends on how the logical relationships between $A$ and $Q$, captured by the match $M$, are related to the reusability of $A$ for $Q$. Although the usefulness of those specification matches depicted in Table 6.1 have individually been argued in one way or another, there does not exist a general approach to reason about the connections between a specification match and its usefulness for determining reusability.

[1]This assessment and this chapter focuses on the correctness of a reusable candidate for a given query specification. It does not address numerous issues related to component adaptation.

Due to the lack of such a general approach, some problems regarding specification matching cannot be solved in an efficient way. For example, given a specification match that happens to be suitable for determining reuse, it is still an open question as to whether there exist *better* specification matches. Intuitively, one specification match is *better* than another if it can identify reusable components for a given query that the other match fails to identify. For example, is there a specification match better than the *relaxed plug-in* match? Or can we further refine the specification match used by Dhara and Leavens [99] in defining object-oriented behavioral subtyping? Another drawback to the lack of a general approach for reasoning about specification matching is that different people may have different perspectives and thus define different matches for capturing certain aspects of reusability, as exhibited in the large variety of proposed specification matches. In this paper, we establish a semantic foundation for reasoning about the connections between a specification match and its usefulness for determining reusability, and provide a framework to evaluate various specification matches. We also study the set of all specification matches suitable for determining reuse, and prove the existence of the *best* ones among those specification matches.

In order to rigorously reason about the usefulness of a specification match, we need a formal definition of *reusability*. The essential requirement of such a definition is correctness: in order for a component to be reusable with respect to a specification, it must "correctly" implement the specification. In this chapter, we only consider components (programs) that terminate, thus by correctness we mean *total* correctness.

The remainder of this chapter is organized as follows. Section 6.2 gives a definition for *reusability* in terms of the correctness formula of Hoare logic [96, 108]. Based on this definition, a special type of specification matching, *reuse-ensuring match*, is defined. A reuse-ensuring match will guarantee the reusability of a component for correctly implementing a

query specification. Section 6.3 gives an overview of relational semantics that will be used as the basis for reasoning about specifications. Then in Section 6.4 we discuss how to prove a specification match is reuse-ensuring. Our proof technique is based on the relational interpretation of programs and specifications [96, 108, 109]. This interpretation model addresses universal quantification, thus simplifying the form of the matches at the specification level. In Section 6.5, we discuss the lattice properties of reuse-ensuring matches. We prove that the set of all equivalence classes of reuse-ensuring matches together with the logic implication ($\Rightarrow$) operator constitute a complete lattice. Moreover, we give the lattice's greatest and least element, which are also the most general and most specific equivalence classes of reuse-ensuring matches, respectively. In Section 6.6, we examine some frequently used concepts in the context of relational interpretations of programs and specification. We show why the usual definition of *characteristic predicate* of a component or specification is flawed, and an alternate definition is given. We also illustrate the essential limitations of signature matching, and show how they can be overcome using specification matching. Finally, we give relevant background notes in Section 6.7 and conclude this chapter in Section 6.8.

## 6.2 Formalizing and Reasoning About Reusability

In this section, we formalize reusability in terms of program correctness with respect to specifications. We then discuss in general term how specification matching is related to reuse. We define *reuse-ensuring* matches to capture the notion of using specification matching to determine reusability.

### 6.2.1 Program Specification and Correctness

The most important property of a program is its functionality, that is, what the program can accomplish. Therefore, a correct functional specification is crucial for the effective (re)use of a program. Many formal specification languages have been proposed [1], such as VDM [2], Z [3], and the Larch family [26]. In this chapter, we do not want to be constrained by the syntactical details of specific specification languages. Instead, we use a general form of specifications that is based on Hoare's axiomatic approach [96]. An *axiomatic specification* of a program is given by a pair of first order assertions about the values of the relevant variables before and after the execution of the program: the *precondition* specifies the initial values, and the *postcondition* specifies the final values and/or their relations with the initial values. In order to differentiate the values of a variable before and after the execution of a program, we use the primed form of a variable to denote the final value of the variable, whereas the non-primed form of a variable denotes its initial value.[2] The precondition of a specification thus is a boolean function of non-primed variables, and the postcondition is a boolean function of primed and/or non-primed variables.

**Example 3** *The specification for computing the quotient and remainder of integer division looks like the following, where $x$, $y$, $q$, and $r$ are all integers.*

$$\begin{array}{ll} \textbf{precond:} & x \geq 0 \wedge y > 0 \\ \textbf{postcond:} & q' \times y + r' = x \wedge r' \geq 0 \wedge y > r' \end{array}$$

It should be noted that in our specification, we assume that by default, the value of each variable may change unless it is explicitly stated that the final value of a variable is

[2]The notion of primed variable for representing final value is borrowed from the Larch interface specification language for C [26].

the same as its initial value. Therefore, we do not have to introduce a *frame* (or *modifies* clause in Larch) in a specification to list the variables whose values may change.

In order to facilitate our discussion in the rest of this chapter, we introduce the following notations.

| | |
|---|---|
| $\mathbf{V}$ | the set of variables that a program operates on. |
| $\mathbf{V'}$ | the set of primed forms of variables in $\mathbf{V}$, i.e., $\mathbf{V'} = \{v' \mid v \in \mathbf{V}\}$. |
| **PreAssert** | the set of first order assertions over $\mathbf{V}$. |
| **PostAssert** | the set of first order assertions over $\mathbf{V} \cup \mathbf{V'}$. |
| **Spec** | the set of specifications. **Spec** = **PreAssert** $\times$ **PostAssert**. |

A specification is usually written as a 2-tuple $\langle p, q \rangle$, where $p \in$ **PreAssert** and $q \in$ **PostAssert** are the precondition and postcondition, respectively. In this chapter, a specification $S$ by default refers to $\langle S_{pre}, S_{post} \rangle$.

The correctness of a program with respect to implementing a specification is captured by a boolean expression, the *correctness formula (CF)* [96, 108, 103]. We use **CF** to denote the set of correctness formulas.

$$\mathbf{CF} = \mathbf{PreAssert} \times \mathbf{P} \times \mathbf{PostAssert}$$

where **P** is the set of programs.

Given a program $A$ and a specification $\langle p, q \rangle$, the correctness formula, written as $\{p\}A\{q\}$, is informally interpreted as follows:

> $\{p\}A\{q\} \equiv$ truth of *program A begun with p satisfied will terminate with q satisfied.*

It should be noted that this definition captures the total correctness of a program: as long as $A$ starts with $p$ satisfied, (1) $A$ is guaranteed to terminate; and (2) $q$ is satisfied when $A$ terminates.

### 6.2.2 Reusability

The reusability of a component (program) for a query specification depends on whether the component satisfies the query specification. The correctness formula provides a semantic measure for "satisfaction".

**Definition 1 (Reusability)** *Given a query specification $Q$: $\langle Q_{pre}, Q_{post} \rangle$, a component $A$ is reusable for implementing $Q$, if $\{Q_{pre}\}A\{Q_{post}\}$ holds.*

Definition 1 emphasizes the semantic correctness of a reusable component implementing a query specification. According to Definition 1, a reusable component is guaranteed to correctly implement the target specification. Although it is possible to apply program correctness proof techniques [108] to check the validity of a correctness formula, this usually requires a great deal of knowledge about the program itself, such as its internal structures and implementation details. Unfortunately, this type of knowledge is seldom available. Furthermore, program correctness proof techniques are typically too computationally expensive to be practical for general use.

In this chapter, we assume that components intended for reuse are delivered with specifications that the components satisfy [82]. (In related investigations, we have applied reverse engineering techniques to obtain predicate-based specifications from existing code [82].) We use $\langle A_{pre}, A_{post} \rangle$ to denote the specification of component $A$, that is, $\{A_{pre}\}A\{A_{post}\}$. In the rest of this paper, depending on the context, a component $A$ refers to either its specification $\langle A_{pre}, A_{post} \rangle$ or the component itself. Specification matching is a method for evaluating and finding reusable components for a query specification by matching component specifications to the query specification. Formally, a specification match is a boolean function defined as below.

$$M\text{: } \mathbf{Spec} \times \mathbf{Spec} \rightarrow \{T, F\}$$

Given a match $M$ and two specifications $S_1$ and $S_2$, if $M(S_1, S_2) = T$, then we say $S_1$ matches $S_2$ according to $M$. Obviously, checking the validity of a match when applied to two specifications is a much easier task than directly proving the correctness of a program.

### 6.2.3 Reuse-Ensuring Match

For a specification match to be useful, it should guarantee the correctness of a component for fulfilling the given query specification. Therefore, we are only interested in those specification matches that when a component matches a query specification, we can be assured that the component is reusable for implementing the query specification. We call these matches *reuse-ensuring.*

**Definition 2 (Reuse-ensuring Match)** *A specification match $M$ is reuse-ensuring, that is, it can ensure that a component $A$ satisfies a query specification $Q$, if and only if for any $A$ and $Q$, $M(A, Q) \wedge \{A_{pre}\}A\{A_{post}\} \Rightarrow \{Q_{pre}\}A\{Q_{post}\}$.*

Once a specification match is proven to be reuse-ensuring, we can use it to evaluate and select components reusable for a query specification by simply checking its validity when applied to candidate components and the query specification.

As an axiomatic proof system for program correctness, Hoare logic [96, 108] has a set of axiom schemata and inference rules, most of which refer to specific program constructs (statements). However, there is one rule, *consequence rule*, that does not involve the specific constructs of a program, as shown below.

$$\frac{p \rightarrow p', \{p'\}C\{q'\}, q' \rightarrow q}{\{p\}C\{q\}}$$

The *consequence rule* states that if the execution of a program $C$ under precondition $p'$ ensures the truth of assertion $q'$, then the execution of $C$ under any precondition that logically implies $p'$ also ensures the truth of any assertion logically implied by $q'$. We can apply the *consequence rule* to prove that certain matches are reuse-ensuring. For example, consider the *plug-in* match, $M(A,Q) = (Q_{pre} \rightarrow A_{pre}) \wedge (A_{post} \rightarrow Q_{post})$. Apply the *consequence rule* with $Q_{pre}$ for $p$, $A_{pre}$ for $p'$, $A_{post}$ for $q'$, $Q_{post}$ for $q$, and $A$ for $C$, we have $\{Q_{pre}\}A\{Q_{post}\}$. Due to the soundness of Hoare logic, we have $M(A,Q) \wedge \{A_{pre}\}A\{A_{post}\} \Rightarrow \{Q_{pre}\}A\{Q_{post}\}$. Therefore, the *plug-in* match is reuse-ensuring.

Unfortunately, Hoare logic is not generally applicable to prove reuse-ensuring matches without involving the internal structures of a component. Thus we turn to the underlying semantics of program specifications and correctness in search of a basis for reasoning about specification matches. One frequently used approach to expressing semantics is based on Dijkstra's weakest precondition functions, *wp* and *wlp* [100]. Given a command $c$, its semantics is defined by $wp.c.p$, the weakest precondition for command $c$ to terminate and establish postcondition $p$, and by $wlp.c.p$, the weakest precondition for command $c$ to establish $p$ if $c$ terminates. Since *wp* and *wlp* act as predicate transformers, this kind of semantics is also called predicate-transformation semantics. Hesselink [110] explored the use of predicate-transformation semantics as a foundation for a formal programming methodology. Although it is possible to use predicate-transformation semantics as a foundation to reason about specification matching, the inclusion of programming language constructs makes it unnecessarily complicated for the purposes of reasoning about specification matching. Therefore in the following discussion, we use a more intuitive method of semantics formalization, called relational semantics, as our basis for reasoning about specification matching. Relational semantics has also been used to interpret Hoare logic [108, 109].

## 6.3 Relational Semantics

An execution step of a program can be considered as a transformation from one state to another. The effects of a program execution can be described by the sequence of states. A terminated execution has a finite sequence of states. In general, the observation of the intermediate states in the execution sequence is unnecessary. The semantics of a program can be directly defined as a relation between the initial and final states of terminated executions. In this section, we first give a brief overview of relation calculus [111]. Then the relational interpretations of programs and specifications are described. Relevant theorems that will facilitate reasoning about specification matches are also discussed.

### 6.3.1 Relation Calculus

**Definition 3 (Relation)** *Given a set $X$, the Cartesian product $X \times X$ is the set of all pairs $\langle x, x' \rangle$, where $x, x' \in X$. A (binary) relation over $X$ is any subset of $X \times X$. The Cartesian product $X \times X$ itself is a special relation, the universal relation.*

In addition to the common set operations: *subset* ($\subseteq$ or $\subset$), *intersection* ($\cap$), *union* ($\cup$) and so forth, we define the following operations that are specific to relations.

**Definition 4 (Domain of a relation)** *Given a relation $R$, $R \subseteq X \times X$, the domain of $R$, denoted as $Dom.R$, is the set of first elements of all pairs in $R$, that is,*

$$Dom.R = \{x \mid x \in X \wedge \exists x'(x' \in X \wedge \langle x, x' \rangle \in R)\}$$

**Definition 5 (Domain restriction)** *Given a relation $R \subseteq X \times X$, and a set $Y \subseteq X$, the domain restricted relation of $R$ by $Y$, denoted by $Y \rceil R$, is the set of pairs in $R$ whose first*

*element is in $Y$, that is,*

$$Y \rceil R = \{\langle x, x' \rangle \mid x \in Y \wedge \langle x, x' \rangle \in R\}$$

**Theorem 1 (Properties of domain restriction)** *Given a set $X$, let $R_1, R_2 \subseteq X \times X$, and $X_1, X_2 \subseteq X$, it is not difficult to show the following properties regarding domain restriction.*

*1.* **Monotonicity of relations.** *If $R_1$ is a subset of $R_2$, then $X_1 \rceil R_1$ is also a subset of $X_1 \rceil R_2$. That is, $R_1 \subseteq R_2 \Rightarrow X_1 \rceil R_1 \subseteq X_1 \rceil R_2$*

*2.* **Monotonicity of sets.** *If $X_1$ is a subset of $X_2$, then $X_1 \rceil R_1$ is also a subset of $X_2 \rceil R_1$. That is, $X_1 \subseteq X_2 \Rightarrow X_1 \rceil R_1 \subseteq X_2 \rceil R_1$*

*3.* **Idempotent.** *Domain restricting an already domain restricted relation by the same set has no effect. That is, $X_1 \rceil (X_1 \rceil R_1) = X_1 \rceil R_1$*

### 6.3.2 Interpretation of Programs and Specifications

The effects of a program execution are recorded in variables operated on by the program. An axiomatic specification is given by stating the initial and final values of these variables, that is, their values in the initial and final states. In general, each variable $v$ belongs to a type $\mathbf{T}_v$ that defines a set of objects (data). For the set of variables $\mathbf{V}$ that a program operates on, we denote the set of (data) objects that all variables in $\mathbf{V}$ can range over as $\mathbf{D}$, that is, $\mathbf{D} = \bigcup_{v \in \mathbf{V}} \mathbf{T}_v$.

**Definition 6 (State)** *A state (or valuation) $s$ is a function with the set $\mathbf{V}$ of variables as the domain and the set $\mathbf{D}$ of (data) objects as the range, such that $\forall v \in \mathbf{V}, s(v) \in \mathbf{T}_v$,*

*where s(v) is called the value of variable v in state s. The set of all the possible states of a program constitute the state space of the program, denoted as* $\mathbf{S}$.

The relational semantics of programs and specifications are given in terms of the following definitions.

**Definition 7 (Interpretation of programs)** *The relational semantics or interpretation of a program P, denoted as* $I_P$*, is a binary relation over the state space* $\mathbf{S}$ *such that* $\langle s, s' \rangle \in I_P$*, if and only if there exists an execution of program P that starts in initial state s and terminates in final state* $s'$.

**Example 4** *The relational semantics of a simple program "x:=3" (where assuming x is declared as an* `Int`*) is* $I_{x:=3} = \{\langle s, s' \rangle \mid s \in \mathbf{S} \wedge s' \in \mathbf{S} \wedge s'(x) = 3\}$*, where* $\mathbf{S} = \{s \mid s : \mathbf{V} \rightarrow \mathtt{Int}\}$, $\mathbf{V} = \{x\}$.

**Definition 8 (Interpretation of preconditions)** *A precondition* $p \in$ **PreAssert** *is interpreted as a subset of state space* $\mathbf{S}$. $S_p = \{s \mid s \in \mathbf{S} \wedge p[\forall v \in \mathbf{V}, v \leftarrow s(v)]\}$*, where* $p[\forall v \in \mathbf{V}, v \leftarrow s(v)]$ *is the predicate obtained by substituting all occurrences of every variable* $v \in \mathbf{V}$ *in p with* $s(v)$.

**Definition 9 (Interpretation of postconditions)** *The interpretation of a postcondition* $q \in$ **PostAssert** *is a relation over state space* $\mathbf{S}$. $R_q = \{\langle s, s' \rangle \mid s \in S \wedge s' \in S \wedge q[\forall v \in \mathbf{V}, \forall v' \in \mathbf{V}', v \leftarrow s(v), v' \leftarrow s'(v)]\}$*, where* $q[\forall v \in \mathbf{V}, \forall v' \in \mathbf{V}', v \leftarrow s(v), v' \leftarrow s'(v)]$ *is the predicate obtained by substituting all occurrences of every variable* $v \in \mathbf{V}$ *in q with* $s(v)$ *and all occurrences of every variable* $v' \in \mathbf{V}'$ *in q with* $s'(v)$.

**Example 5** *The boolean constant T (true) is interpreted as either* $\mathbf{S}$ *or* $\mathbf{S} \times \mathbf{S}$*, depending on whether it is a precondition or postcondition, respectively. The interpretation of F (false)*

*is $\phi$ (empty set). The assertion p: $x \geq 0$, if used as a precondition, is interpreted as a set of states, $S_p = \{s \mid s \in \mathbf{S} \wedge s(x) \geq 0\}$. The assertion q: $z' \times z' \leq x \wedge (z'+1) \times (z'+1) > x$ is interpreted as a relation over* $\mathbf{S}$, $R_q = \{\langle s, s' \rangle \mid s \in \mathbf{S} \wedge s' \in \mathbf{S} \wedge s'(z) \times s'(z) \leq s(x) \wedge (s'(z)+1) \times (s'(z)+1) > s(x)\}$.

Based on the interpretations of assertions given before, we include the following theorem regarding assertions.

**Theorem 2 (Properties of assertions)** *For two assertions, $p_1$ and $p_2$. If both $p_1$ and $p_2$ are in* **PreAssert**, *then $p_1 \rightarrow p_2 \Leftrightarrow S_{p_1} \subseteq S_{p_2}$. Or if both $p_1$ and $p_2$ are in* **PostAssert**, *then $p_1 \rightarrow p_2 \Leftrightarrow R_{p_1} \subseteq R_{p_2}$.*

**Definition 10 (Interpretation of correctness formulas)** *In terms of relational semantics, a correctness formula has the following interpretation: $I[\{p\}A\{q\}] = S_p \subseteq Dom.I_A \wedge S_p \rceil I_A \subseteq R_q$.*

Definition 10 captures the notion of total correctness as described in Section 2.1. The first conjunct states that given $p$ satisfied, $A$ will terminate. The second conjunct ensures that $q$ will be satisfied when $A$ terminates.

An axiomatic specification consists of an assertion of **PreAssert** as the precondition and an assertion of **PostAssert** as the postcondition. However, not any pair of assertions from **PreAssert** $\times$ **PostAssert** can form an *implementable* specification. For example, specification $\langle T, F \rangle$ cannot be implemented by any program, since there is not a final state (terminating state) that satisfies $F$.

**Definition 11 (Implementable specification)** *Given a state assertion p and a relation assertion q, the 2-tuple $\langle p, q \rangle$ is an implementable specification if and only if $S_p \subseteq Dom.R_q$.*

As an example, consider specification $\langle T, x \geq z' \times z' \wedge x < (z'+1) \times (z'+1) \rangle$. This is not an implementable specification, since there does not exist a program that, given any input (for instance, $x = -2$), will establish the truth of $x \geq z' \times z' \wedge x < (z'+1) \times (z'+1)$. In the rest of this chapter, we assume all referenced specifications to be implementable.

**Theorem 3** *Given two implementable specifications, $\langle p, q \rangle$ and $\langle w, u \rangle$. If $p \wedge q \rightarrow w \wedge u$, then $p \rightarrow w$.*

**Proof.** Suppose $p \wedge q \rightarrow w \wedge u$ is true. Since both $p \wedge q$ and $w \wedge u$ can be considered assertions in **PostAssert**, according to Theorem 2, we have $R_{p \wedge q} \subseteq R_{w \wedge u}$, which implies $Dom.R_{p \wedge q} \subseteq Dom.R_{w \wedge u}$. Since $S_p \subseteq Dom.R_q$ (by Definition 11), we have $Dom.R_{p \wedge q} = S_p$. Similarly, we have $Dom.R_{w \wedge u} = S_w$. Therefore, we have $S_p \subseteq S_w$, which, according to Theorem 2, means $p \rightarrow w$. □

From Theorem 3, we can immediately derive the following corollary.

**Corollary 1** *Given two implementable specifications, $\langle p, q \rangle$ and $\langle w, u \rangle$. If $p \wedge q \leftrightarrow w \wedge u$, then $p \leftrightarrow w$.*

**Definition 12 (Interpretation of specifications)** *The interpretation of an implementable specification $\langle p, q \rangle$ is the result of domain restricting $R_q$ by $S_p$, that is, $R_{\langle p,q \rangle} = S_p \rceil R_q$.*

**Theorem 4** *Given an implementable specification $\langle p, q \rangle$, $R_{\langle p,q \rangle} = R_{p \wedge q}$.*

**Proof.**

1. For any $\langle s, s' \rangle \in R_{\langle p,q \rangle}$, it follows that $s \in S_p$ and $\langle s, s' \rangle \in R_{\langle p,q \rangle}$ (by Definition 12),
2. which means both $p[\forall v \in \mathbf{V}, v \leftarrow s(v)]$ and $q[\forall v \in \mathbf{V}, \forall v' \in \mathbf{V}', v \leftarrow s(v), v' \leftarrow s'(v)]$ hold.
3. Since $p$ is a precondition, it does not contain primed variables (i.e., variables in $\mathbf{V}'$), therefore $p[\forall v \in \mathbf{V}, v \leftarrow s(v)] = p[\forall v \in \mathbf{V}, \forall v' \in \mathbf{V}', v \leftarrow s(v), v' \leftarrow s'(v)]$.

4. It immediately follows that $(p \wedge q)[\forall v \in \mathbf{V}, \forall v' \in \mathbf{V}', v \leftarrow s(v), v' \leftarrow s'(v)]$ holds, which means $\langle s, s' \rangle \in R_{p \wedge q}$.

5. On the other hand, for any $\langle s, s' \rangle \in R_{p \wedge q}$, $(p \wedge q)[\forall v \in \mathbf{V}, \forall v' \in \mathbf{V}', v \leftarrow s(v), v' \leftarrow s'(v)]$ is true.

6. Since $(p \wedge q)[\forall v \in \mathbf{V}, \forall v' \in \mathbf{V}', v \leftarrow s(v), v' \leftarrow s'(v)] = p[\forall v \in \mathbf{V}, \forall v' \in \mathbf{V}', v \leftarrow s(v), v' \leftarrow s'(v)] \wedge q[\forall v \in \mathbf{V}, \forall v' \in \mathbf{V}', v \leftarrow s(v), v' \leftarrow s'(v)] = p[\forall v \in \mathbf{V}, v \leftarrow s(v)] \wedge q[\forall v \in \mathbf{V}, \forall v' \in \mathbf{V}', v \leftarrow s(v), v' \leftarrow s'(v)]$, we have $s \in S_p \wedge \langle s, s' \rangle \in R_q$.

7. It follows that $\langle s, s' \rangle \in S_p \rceil R_q$, that is, $\langle s, s' \rangle \in R_{\langle p, q \rangle}$. □

## 6.4 Proving Reuse-Ensuring Matches

Relational semantics provides a basis for reasoning about specification matches. In this section, we show how we can prove a specification match is reuse-ensuring based on relational semantics. In order to show that a match $M$ is reuse-ensuring, we need to prove that for any specification $Q$ and component $A$, as long as $A$ matches $Q$ according to $M$, then $A$ will correctly implement $Q$, that is, $M(A, Q) \wedge \{A_{pre}\}A\{A_{post}\} \Rightarrow \{Q_{pre}\}A\{Q_{post}\}$.

**Example 6** *As a first example, we show* $M_{exact-pred-2} : (A_{pre} \wedge A_{post}) \leftrightarrow (Q_{pre} \wedge Q_{post})$ *is reuse-ensuring.*

**Proof.** Suppose $(A_{pre} \wedge A_{post}) \leftrightarrow (Q_{pre} \wedge Q_{post})$ hold. According to Corollary 1, $A_{pre} \leftrightarrow Q_{pre}$ holds, which implies (1): $S_{A_{pre}} = S_{Q_{pre}}$. In the meantime, according to Theorems 2 and 4, we immediately have (2): $S_{Q_{pre}} \rceil R_{Q_{post}} = S_{A_{pre}} \rceil R_{A_{post}}$ from our assumption. Since $\{A_{pre}\}A\{A_{post}\}$, we have (3): $S_{A_{pre}} \subseteq Dom.I_A \wedge S_{A_{pre}} \rceil I_A \subseteq R_{A_{post}}$. From the second conjunct of (3), we have $S_{A_{pre}} \rceil I_A \subseteq S_{A_{pre}} \rceil R_{A_{post}}$ (according to Theorem 1). Together with (1) and (2), we have (4): $S_{Q_{pre}} \rceil I_A \subseteq S_{Q_{pre}} \rceil R_{Q_{post}}$. Also following the first conjunct of (3) and (1) is (5): $S_{Q_{pre}} \subseteq Dom.I_A$. From (4) and (5) immediately follows $\{Q_{pre}\}A\{Q_{post}\}$. □

**Example 7** *Our second example shows that* $M_{relaxed-plug-in} : (Q_{pre} \rightarrow A_{pre}) \wedge (Q_{pre} \wedge A_{post} \rightarrow Q_{post})$ *is reuse-ensuring.*

**Proof.** Suppose $(Q_{pre} \rightarrow A_{pre}) \wedge (Q_{pre} \wedge A_{post} \rightarrow Q_{post})$ hold. From the first conjunct comes (1): $S_{Q_{pre}} \subseteq S_{A_{pre}}$ (Theorem 2), which immediately leads to (2): $S_{Q_{pre}} \rceil I_A \subseteq S_{A_{pre}} \rceil I_A$ (Theorem 1). Since $\{A_{pre}\}A\{A_{post}\}$, we have (3): $S_{A_{pre}} \subseteq Dom.I_A \wedge S_{A_{pre}} \rceil I_A \subseteq R_{A_{post}}$. From the first conjunct of (3) and (1) immediately follows (4): $S_{Q_{pre}} \subseteq Dom.I_A$. On the other hand, from the second conjunct of (3) and (2), we have $S_{Q_{pre}} \rceil I_A \subseteq R_{A_{post}}$, which implies $S_{Q_{pre}} \rceil I_A \subseteq S_{Q_{pre}} \rceil R_{A_{post}}$ (Theorem 1). From the second conjunct of our assumption, $Q_{pre} \wedge A_{post} \rightarrow Q_{post}$, we have $S_{Q_{pre}} \rceil R_{A_{post}} \subseteq Q_{post}$ (Theorem 4 and Theorem 2). Therefore, we have (5): $S_{Q_{pre}} \rceil I_A \subseteq R_{Q_{post}}$ Combining (4) and (5) follows $\{Q_{pre}\}A\{Q_{post}\}$. □

As illustrated in the above examples, proving a match is reuse-ensuring directly based on relational semantics may be a cumbersome task. The following theorem may greatly simplify the task.

**Theorem 5** *Let $M'$ be a reuse-ensuring match. A match $M$ is reuse-ensuring if for any $A$ and $Q$, $M(A,Q) \Rightarrow M'(A,Q)$.*

**Proof.** The truth of the above claim immediately follows the definition of reuse-ensuring match and the transitivity of logical implication ($\Rightarrow$). □

As long as a reuse-ensuring match is known, Theorem 5 simplifies the proof of reuse-ensuring matches by relying on the logical relations between specification matches, rather than involving the semantics-based interpretations of programs and specifications. However, it should be noted that the fact a match does not logically imply a given reuse-ensuring match does not necessarily mean that the match is not reuse-ensuring, unless the given reuse-ensuring match is the most general. We further discuss this issue in the next section.

**Example 8** *Consider match* $M_{exact-pre/post} : (Q_{pre} \leftrightarrow A_{pre}) \wedge (A_{post} \leftrightarrow Q_{post})$. *It is easy to show that* $M_{exact-pre/post}$ *implies* $M_{exact-pred-2} : (A_{pre} \wedge A_{post}) \leftrightarrow (Q_{pre} \wedge Q_{post})$, *which is reuse-ensuring as shown in Example 6. Thus* $M_{exact-pre/post}$ *is reuse-ensuring.*

**Example 9** *Consider match* $M_{plug-in} : (Q_{pre} \rightarrow A_{pre}) \wedge (A_{post} \rightarrow Q_{post})$. *It can be shown that* $M_{plug-in}$ *does not imply* $M_{exact-pred-2} : (A_{pre} \wedge A_{post}) \leftrightarrow (Q_{pre} \wedge Q_{post})$. *However, this does not exclude* $M_{plug-in}$ *from being reuse-ensuring. In fact,* $M_{plug-in}$ *is reuse-ensuring, since it implies reuse-ensuring match* $M_{relaxed-plug-in} : (Q_{pre} \rightarrow A_{pre}) \wedge (Q_{pre} \wedge A_{post} \rightarrow Q_{post})$ *(see Example 7).*

**Example 10** *Finally, we show that* $M_{guarded-gen-pred}$ *is reuse-ensuring by proving* $M_{guarded-gen-pred} \Rightarrow M_{relaxed-plug-in}$. *In fact,* $M_{guarded-gen-pred}$ *and* $M_{relaxed-plug-in}$ *are logically equivalent.*

**Proof.** For the sake of readability, we assume the normal precedence of logical connectives, i.e., $\neg$, $\wedge$, $\vee$, and $\rightarrow$ in the order of decreasing precedence.

$$
\begin{aligned}
M_{guarded-gen-pred}(A, Q) = \; & (Q_{pre} \rightarrow A_{pre}) \wedge ((A_{pre} \rightarrow A_{post}) \rightarrow (Q_{pre} \rightarrow Q_{post})) \\
\Leftrightarrow \; & (\neg Q_{pre} \vee A_{pre}) \wedge (\neg(\neg A_{pre} \vee A_{post}) \vee (\neg Q_{pre} \vee Q_{post})) \\
\Leftrightarrow \; & (\neg Q_{pre} \vee A_{pre}) \wedge ((A_{pre} \wedge \neg A_{post}) \vee \neg Q_{pre} \vee Q_{post}) \\
\Leftrightarrow \; & (\neg Q_{pre} \vee A_{pre}) \wedge (A_{pre} \wedge \neg A_{post}) \vee \neg Q_{pre} \vee (\neg Q_{pre} \vee A_{pre}) \wedge Q_{post} \\
\Leftrightarrow \; & A_{pre} \wedge \neg A_{post} \vee \neg Q_{pre} \vee A_{pre} \wedge Q_{post} \\
\Leftrightarrow \; & A_{pre} \wedge (\neg A_{post} \vee Q_{post}) \vee \neg Q_{pre} \\
\Leftrightarrow \; & Q_{pre} \rightarrow (A_{pre} \wedge (A_{post} \rightarrow Q_{post})) \\
\Leftrightarrow \; & (Q_{pre} \rightarrow A_{pre}) \wedge (Q_{pre} \rightarrow (A_{post} \rightarrow Q_{post})) \\
\Leftrightarrow \; & (Q_{pre} \rightarrow A_{pre}) \wedge (Q_{pre} \wedge A_{post} \rightarrow Q_{post}) \\
= \; & M_{relaxed-plug-in}(A, Q)
\end{aligned}
$$

Therefore, $M_{guarded-gen-pred} \Leftrightarrow M_{relaxed-plug-in}$. □

Figure 6.1 shows the reuse-ensuring matches discussed in this section and their relations. In Figure 6.1, an arrow represents a logical implication between two matches.

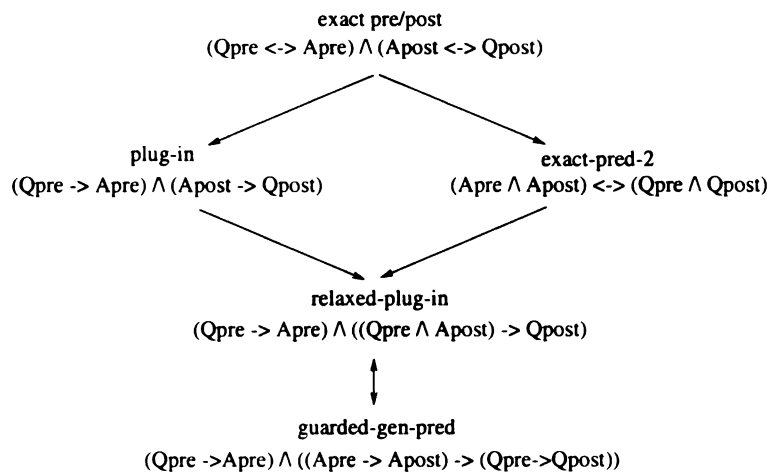



Figure 6.1: Reuse-ensuring matches and their relations

## 6.5 Lattice Properties of Reuse-Ensuring Matches

The set of all reuse-ensuring matches are partitioned into equivalence classes by the logical equivalence operator ($\Leftrightarrow$). Let $REM$ be the set of all equivalence classes of reuse-ensuring matches. We introduce the following operations regarding equivalence class. Let $c$ and $m$ be an equivalence class and a reuse-ensuring match, respectively, then $c^m$ denotes an arbitrary element of $c$, and $m^c$ denotes the equivalence class to which $m$ belongs. Obviously, we have $(c^m)^c = c$. Furthermore, we extend the logical implication operator ($\Rightarrow$) to equivalence classes of reuse-ensuring matches. For two given equivalence classes, $c_1, c_2 \in REM$, we

define $c_1 \Rightarrow c_2$ as $c_1^m \Rightarrow c_2^m$. It is easy to show that logical implication ($\Rightarrow$) over $REM$ is *reflexive* ($\forall c \in REM, c \Rightarrow c$), *antisymmetric* ($\forall c_1, c_2 \in REM, (c_1 \Rightarrow c_2) \wedge (c_2 \Rightarrow c_1) \Rightarrow (c_1 = c_2)$), and *transitive* ($\forall c_1, c_2, c_3 \in REM, (c_1 \Rightarrow c_2) \wedge (c_2 \Rightarrow c_3) \Rightarrow (c_1 \Rightarrow c_3)$), thus $\langle REM, \Rightarrow \rangle$ is a partially ordered set (POSET). Further, we show that $\langle REM, \Rightarrow \rangle$ is a complete lattice [112]. This means that among all the equivalence classes of reuse-ensuring matches, there are two special ones: the most general and the most specific. The former is the greatest element of $REM$, and the latter is the least element of $REM$. Specification matches in the most general equivalence class are accordingly called most general reuse-ensuring matches. Therefore, in order to prove if a match is reuse-ensuring, we only need to check if it implies a most general reuse-ensuring match.

**Lemma 1** *$REM$ is a finite set.*

**Proof.** Each match can be regarded as a logical function of four parameters: $Q_{pre}$, $Q_{post}$, $A_{pre}$, and $A_{post}$, each of which is a predicate with value either *true* or *false*. Thus, the number of all possible input combinations to a match (logical function) is $2^4 = 16$. Given an input, a match can be evaluated to be either *true* or *false*, thus the maximum number of matches that are not logically equivalent is $2^{16}$. (Among those matches, only a portion are reuse-ensuring.) Therefore, $REM$ is a finite set. □

**Theorem 6** *$\langle REM, \Rightarrow \rangle$ is a complete lattice.*

**Proof.** For any subset $\Psi$ of $REM$, we show that the equivalence classes that the conjunction and disjunction of representatives from each member of $\Psi$ belongs to are the greatest lower bound (*glb*) and least upper bound (*lub*) of $\Psi$, respectively. It should be noted that since $REM$ is finite, and so is $\Psi$, the above conjunction and disjunction are well-defined.

1. $lub\Psi = (\bigvee_{\psi\in\Psi}\psi^{\mathsf{m}})^{\mathsf{c}}$: the equivalence class that the disjunction of representatives from each member of $\Psi$ belongs to is the least upper bound (*lub*) of $\Psi$.

   We claim that $(\bigvee_{\psi\in\Psi}\psi^{\mathsf{m}})^{\mathsf{c}} \in REM$ is true. This is because $\forall\psi \in \Psi$, $\psi^{\mathsf{m}}(A,Q) \Rightarrow \{Q_{pre}\}A\{Q_{post}\}$, thus $\bigvee_{\psi\in\Psi}\psi^{\mathsf{m}}(A,Q) \Rightarrow \{Q_{pre}\}A\{Q_{post}\}$, that is, $\bigvee_{\psi\in\Psi}\psi^{\mathsf{m}}$ is reuse-ensuring. Since $\forall\varphi \in \Psi$, $\varphi^{\mathsf{m}} \Rightarrow \bigvee_{\psi\in\Psi}\psi^{\mathsf{m}}$, that is, $(\varphi^{\mathsf{m}})^{\mathsf{c}} = \varphi \Rightarrow (\bigvee_{\psi\in\Psi}\psi^{\mathsf{m}})^{\mathsf{c}}$, therefore $(\bigvee_{\psi\in\Psi}\psi^{\mathsf{m}})^{\mathsf{c}}$ is an upper bound of $\Psi$. Next we show that $(\bigvee_{\psi\in\Psi}\psi^{\mathsf{m}})^{\mathsf{c}}$ implies any upper bound of $\Psi$. Let $\varphi$ be an arbitrary upper bound of $\Psi$. For $\forall\psi \in \Psi$, we have $\psi \Rightarrow \varphi$, and thus $\bigvee_{\psi\in\Psi}\psi^{\mathsf{m}} \Rightarrow \varphi^{\mathsf{m}}$, which means $(\bigvee_{\psi\in\Psi}\psi^{\mathsf{m}})^{\mathsf{c}} \Rightarrow (\varphi^{\mathsf{m}})^{\mathsf{c}} = \varphi$. Thus $(\bigvee_{\psi\in\Psi}\psi^{\mathsf{m}})^{\mathsf{c}}$ is the *lub* of $\Psi$.

2. $glb\Psi = (\bigwedge_{\psi\in\Psi}\psi^{\mathsf{m}})^{\mathsf{c}}$: the equivalence class that the conjunction of representatives from each member of $\Psi$ belongs to is the greatest lower bound (*glb*) of $\Psi$.

   Obviously, $\bigwedge_{\psi\in\Psi}\psi^{\mathsf{m}}$ is a reuse-ensuring match, that is, $(\bigwedge_{\psi\in\Psi}\psi^{\mathsf{m}})^{\mathsf{c}} \in REM$ . Since $\forall\varphi \in \Psi$, $\bigwedge_{\psi\in\Psi}\psi^{\mathsf{m}} \Rightarrow \varphi^{\mathsf{m}}$, that is, $(\bigwedge_{\psi\in\Psi}\psi^{\mathsf{m}})^{\mathsf{c}} \Rightarrow (\varphi^{\mathsf{m}})^{\mathsf{c}} = \varphi$, therefore $(\bigwedge_{\psi\in\Psi}\psi^{\mathsf{m}})^{\mathsf{c}}$ is a lower bound of $\Psi$. Next we prove that $(\bigwedge_{\psi\in\Psi}\psi^{\mathsf{m}})^{\mathsf{c}}$ is implied by any lower bound of $\Psi$. Let $\varphi$ be an arbitrary lower bound of $\Psi$. For $\forall\psi \in \Psi$, we have $\varphi \Rightarrow \psi$, and thus $\varphi^{\mathsf{m}} \Rightarrow \bigwedge_{\psi\in\Psi}\psi^{\mathsf{m}}$, which means $(\varphi^{\mathsf{m}})^{\mathsf{c}} \Rightarrow (\bigwedge_{\psi\in\Psi}\psi^{\mathsf{m}})^{\mathsf{c}}$. Thus $(\bigwedge_{\psi\in\Psi}\psi^{\mathsf{m}})^{\mathsf{c}}$ is the *glb* of $\Psi$.

   Combining the above arguments, we have proved that $\langle REM, \Rightarrow\rangle$ is a complete lattice. □

From the proof of Theorem 6, it immediately follows that $(\bigvee_{\psi\in REM}\psi^{\mathsf{m}})^{\mathsf{c}}$ is the least upper bound (*lub*) of $REM$, or the greatest element of $REM$. In terms of determining reuse, $(\bigvee_{\psi\in REM}\psi^{\mathsf{m}})^{\mathsf{c}}$ is also called the most general equivalence class of reuse-ensuring matches, and $\bigvee_{\psi\in REM}\psi^{\mathsf{m}}$ is a most general reuse-ensuring match. Theoretically, for any given

specification match, we can prove or refute whether it is reuse-ensuring or not by checking if the given specification match implies a most general reuse-ensuring match. However, it is difficult to use $\bigvee_{\psi \in REM} \psi^{\mathsf{m}}$ for this purpose. In the following discussion, we show that the *relaxed plug-in* match is also a most general reuse-ensuring match. [3]

**Theorem 7 (Most general reuse-ensuring match)** $M_{relaxed-plug-in}(A,Q) = (Q_{pre} \rightarrow A_{pre}) \wedge (Q_{pre} \wedge A_{post} \rightarrow Q_{post})$ *is a most general reuse-ensuring match.*

**Proof.** We show that $M^{\mathsf{c}}_{relaxed-plug-in}$ is the least upper bound (*lub*) of $REM$. In Example 7, we have shown that $M_{relaxed-plug-in}$ is reuse-ensuring, thus, $M^{\mathsf{c}}_{relaxed-plug-in} \in REM$. We now show that $M^{\mathsf{c}}_{relaxed-plug-in}$ is an upper bound of $REM$.

1. For any $c \in REM$, we have $c^{\mathsf{m}}(A,Q) \wedge \{A_{pre}\}A\{A_{post}\} \Rightarrow \{Q_{pre}\}A\{Q_{post}\}$. That is, (1): $c^{\mathsf{m}}(A,Q) \wedge S_{A_{pre}} \subseteq Dom.I_A \wedge S_{A_{pre}} \rceil I_A \subseteq R_{A_{post}} \Rightarrow S_{Q_{pre}} \subseteq Dom.I_A \wedge S_{Q_{pre}} \rceil I_A \subseteq R_{Q_{post}}$ (by Definition 10). Notice that in (1), component $A$ is an arbitrary implementation satisfying specification $\langle A_{pre}, A_{post} \rangle$. For discussion purposes, we instantiate $A$ to be $Dom.I_A = S_{A_{pre}}$ and $I_A = R_{A_{post}}$,

2. (1) is simplified as (2): $c^{\mathsf{m}}(A,Q) \Rightarrow S_{Q_{pre}} \subseteq S_{A_{pre}} \wedge S_{Q_{pre}} \rceil R_{A_{post}} \subseteq R_{Q_{post}}$.

3. According to Theorem 2, we have (3): $S_{Q_{pre}} \subseteq S_{A_{pre}} \Leftrightarrow Q_{pre} \rightarrow A_{pre}$.

4. According to Definition 11, Theorems 4 and 2, we have (4): $S_{Q_{pre}} \rceil R_{A_{post}} \subseteq R_{Q_{post}} \Leftrightarrow Q_{pre} \wedge A_{post} \rightarrow Q_{post}$.

5. Combining (2), (3), and (4), we have $c^{\mathsf{m}}(A,Q) \Rightarrow (Q_{pre} \rightarrow A_{pre}) \wedge (Q_{pre} \wedge A_{post} \rightarrow Q_{post})$. That is, $c^{\mathsf{m}}(A,Q) \Rightarrow M_{relaxed-plug-in}(A,Q)$, which means $c(A,Q) \Rightarrow M^{\mathsf{c}}_{relaxed-plug-in}(A,Q)$.

   Therefore, $M^{\mathsf{c}}_{relaxed-plug-in}$ is an upper bound of $REM$. Since $M^{\mathsf{c}}_{relaxed-plug-in} \in REM$, $M^{\mathsf{c}}_{relaxed-plug-in}$ is the least upper bound (*lub*) of $REM$. Thus $M_{relaxed-plug-in}$ is a most general reuse-ensuring match. □

Theorem 7 answers our questions presented in Section 6.1. That is, there does not exist a *better* specification match than the *relaxed plug-in* match in determining reuse, nor can we refine the *guarded generalized predicate* match used in defining object-oriented behavioral subtyping (where in Example 10, we showed that $M_{relaxed-plug-in}$ and $M_{guarded-gen-pred}$ are logically equivalent).

[3] It should be noted that all the most general reuse-ensuring matches are logically equivalent, thus there is only one equivalence class of them in $REM$.

## 6.6 Discussion

The relational interpretation of programs and specification helps to clarify some important concepts, such as *implementable* specifications as discussed in Section 6.3.2. In this section, we examine some other concepts in the context of relational semantics. The relational interpretation provides another perspective to understand the rationale underlying these concepts and their limitations.

### 6.6.1 Characteristic Predicate

When discussing software components, it would be ideal to have a single predicate to exactly characterize the observable behavior of a component. Such a predicate is called *characteristic predicate* of a component [28, 30]. Unfortunately, the axiomatic specification of a software component $A$ is not a predicate, instead a pair of predicates, $\langle A_{pre}, A_{post} \rangle$. Both Jeng and Cheng [28] and Zaremski and Wing [30] have defined the characteristic predicate of a component $A$ as an implication between its precondition and postcondition. That is, $A_{pred} = A_{pre} \rightarrow A_{post}$, where $A_{pred}$ represents the characteristic predicate of $A$. This definition captures the general intuition underlying an axiomatic specification. That is, if $A_{pre}$ holds, then $A_{post}$ will be satisfied after the execution of the component (the termination of the component is assumed). However, the truth of $A_{pred}$ will also be established whenever $A_{pre}$ does not hold.

In terms of a relational interpretation, $A_{pred}$, which is in **PostAssert**, defines a binary relation over **S**, $R_{A_{pred}} = R_{A_{pre} \rightarrow A_{post}} = R_{\neg A_{pre} \vee A_{post}} = R_{\neg A_{pre} \vee A_{pre} \wedge A_{post}} = R_{\neg A_{pre}} \cup R_{A_{pre} \wedge A_{post}} = S_{\neg A_{pre}} \times \mathbf{S} \cup R_{\langle A_{pre}, A_{post} \rangle}$. As shown in the last formula, in addition to the binary relation defined by specification $\langle A_{pre}, A_{post} \rangle$, the characteristic predicate $A_{pred}$ of component $A$ also includes all the tuples $\langle s, s' \rangle$ where $s, s' \in \mathbf{S}$ and $A_{pre}$ is satisfied in

state $s$. This unwanted inclusion may cause the characteristic predicate of a component to fail in uniquely characterizing the behavior of the component. For example, consider two specifications, $A : \langle x \geq 0, T \rangle$ and $B : \langle x \leq 0, T \rangle$. It is not difficult to show that the binary relations defined by the characteristic predicates of $A$ and $B$ are the same, both are the universal relation, $\mathbf{S} \times \mathbf{S}$. However, these obviously are two different specifications.

In order to resolve this problem, we can define the characteristic predicate to be the *logical and* of the precondition and postcondition of a component, that is, $A_{pred} = A_{pre} \wedge A_{post}$. Since, in terms of the relational interpretation, $R_{A_{pre} \wedge A_{post}} = R_{\langle A_{pre}, A_{post} \rangle}$ (Theorem 4), this definition precisely characterizes the behavior of a component.

### 6.6.2 Signature Matching

In general, signature matching is used as a preprocessor to specification matching [95, 113]. However, signature matching sometimes may preclude certain potentially reusable components. For example, consider a real number function $rmax$ that returns the greater value of its two arguments. According to the often-used *argument contra- and result co-variance* rule [98, 114], this function cannot be used for integer numbers. This is obviously too restrictive.[4]

In fact, the restrictions imposed by the *argument contra- and result co- variance* rule are similar to those of the *plug-in* match in specification matching. Both *plug-in* match and *argument contra- and result co- variance* rule consider the domain and the range of a computation (function) separately. However, as illustrated in the relational interpretations of programs and specifications, the essence of a computation (function) is the *mapping*

[4]Due to constraints imposed by machine representations of numbers, there does not always exist a subtype relationship between real numbers ( floating-point numbers) and integers. For the illustrative purposes of this paper, we consider only mathematical numbers where any integer is a real number.

from its domain to its range. That is, given an element of the domain as input, what will be produced as the result of the function's execution? Without considering the mapping of a computation (function), a match criterion cannot capture the *reusability* relation in a precise (neither too restrictive nor too loose) way. Since the mapping of a function is captured in the function's specification, specification matching criterion can be designed to take into consideration the mapping information, as in the case of *relaxed plug-in* match. However, since the signature of a function does not contain the mapping information of the function, the restrictions imposed by the *argument contra- and result co- variance* rule cannot be resolved by solely relying on signatures.

Since signatures can be considered as a special kind of specification that only contains typing information, we can overcome the restrictions imposed by signature matching by extending specification matching to consider typing information. Consider the following specifications for *rmax* and *imax*, where typing information is embedded in the specifications.

**rmax(x, y): z**
**precond:** $x \in real \wedge y \in real \wedge z \in real$
**postcond:** $x \in real \wedge y \in real \wedge z \in real \wedge (z' = x \vee z' = y) \wedge z' \geq x \wedge z' \geq y$

**imax(i, j): k**
**precond:** $i \in int \wedge j \in int \wedge k \in int$
**postcond:** $i \in int \wedge j \in int \wedge k \in int \wedge (k' = i \vee k' = j) \wedge k' \geq i \wedge k' \geq j$

In order to match the two specifications, we first establish the following mappings between the variables of the two functions based on their signature structure (i.e., input, output, and parameter order).

$$x \leftrightarrow i,\ y \leftrightarrow j,\ z \leftrightarrow k$$

Based on the above variable mappings, we can revise one of the specifications so that it is consistent with the other in terms of variable naming. The following is the revised specification of *imax*.

**imax(x, y): z**
**precond:** $x \in int \wedge y \in int \wedge z \in int$
**postcond:** $x \in int \wedge y \in int \wedge z \in int \wedge (z' = x \vee z' = y) \wedge (z' \geq x \wedge z' \geq y)$

Now we apply the *plug-in* match to determine the reusability of *rmax* for implementing *imax*. Since $x \in real \wedge y \in real \wedge z \in real \rightarrow x \in int \wedge y \in int \wedge z \in int$ does not hold, the postcondition of *rmax* will not imply that of *imax*. Therefore, the *plug-in* match fails to establish the reusability of *rmax* for *imax*. Given our previous analysis, this does not surprise us. Now we proceed to apply the most general reuse-ensuring match, $M_{relaxed-plug-in}$. It is not difficult to show that

$$(imax_{pre} \rightarrow rmax_{pre}) \wedge (imax_{pre} \wedge rmax_{post} \rightarrow imax_{post})$$

holds. That is, *rmax* can be reused for implementing *imax*.

In summary, we showed how specification matching can be used to overcome the essential restrictions of signature matching.

## 6.7 Notes

As mentioned previously, many projects have explored the use of specification matching to determine software reuse and object-oriented subtyping [93, 27, 28, 30, 59, 91, 95, 97, 98, 99]. A number of matching criteria have been proposed to capture the reusability of a component for implementing a query specification. This chapter describes a semantic foundation, based on relational semantics, for reasoning about specification matches and reusability, thereby providing a framework to evaluate various specification matches.

Mili, *et al* [29] propose a refinement-based approach to component retrieval. They use relational specifications to represent components. A relational specification directly specifies the input/output relations of a component (function). Then they define refinement relation between two relational specifications as their basis for component storing and retrieval. Despite being expressed in different terms, their refinement relation has the same objective as those of various specification matches that we discussed in this paper, that is, to capture the notion of reusability through comparing two specifications. In essence, their refinement relation is equivalent to relaxed plug-in match. Like other specification matching work, they address the connection between their refinement relation and reusability in a rather intuitive way. Although we used relational interpretation as the basis for reasoning about specification matches, the *relation* in our work is defined over the program state space, rather than the pairs of input/output values. Moreover, the objective of our work is to find a sound foundation and framework to evaluate specification matching for reuse, rather than define yet another matching criteria. This distinguishes this current work from Mili, *et al*'s work and others'.

Relational semantics of programs was first introduced by Hoare and Lauer [109] and later was used to define semantics of Hoare logic [108]. Our discussion of specification match and their semantic interpretation falls in the framework of Hoare logic, but we made some changes in our approach. First, we are not involved in the specific constructs of a program, instead, we are only concerned with the overall behavior. Therefore we did not discuss the interpretation of formulas in Hoare logic regarding specific program constructs. Second, and more importantly, the postcondition in Hoare logic is the function of only the final values of variables. This often makes specifications cumbersome since auxiliary variables have to be introduced. We adopt the approach that most current specification languages

take, that is, we allow both initial and final values of variables to appear in a postcondition. Thus the interpretation of our postcondition is a binary relation over the state space, rather than a subset of the state space, as in the case of Hoare logic. Finally, in Hoare logic, the correctness formula is intended to capture the notion of partial correctness, rather than total correctness as we did in this paper. Therefore, the interpretation of the correctness formula in Hoare logic does not contain the conjunct $S_p \subseteq Dom.I_A$ of Definition 10 that ensures the termination of program execution.

## 6.8 Conclusion

In this chapter, a semantic foundation is established to reason about the connections between a specification match and its usefulness for determining reusability. Based on this semantic foundation, we proved that the set of all equivalence classes of reuse-ensuring matches together with the logical implication ($\Rightarrow$) operator constitute a complete lattice, and showed that the *relaxed plug-in* match is a most general (or a *best*) reuse-ensuring match. We also discussed and clarified some concepts in the context of the proposed semantic foundation. The work described in this chapter provides a formal foundation for applying specification matching-based methods to component evaluation, and simplifies the development of the *best* component evaluation method [33].

# Chapter 7

# Interface Generality Relation

In Chapter 6, we established a semantic foundation to reason about the usefulness of a specification match in determining reusability. However, these specification matches are only applicable to determining fine-grained components, i.e., functions. In this chapter, we extend the notion of specification matching to coarse-grained architectural components. We define an *interface generality relation* to capture the reusability of an architectural component for satisfying an interface, therefore enabling the evaluation of these coarse-grained, architectural components.

A component interacts with its environment by exchanging (i.e. requiring or providing) resources through its ports. The set of all ports that a component has together with constraints that specify the global properties regarding the behavior of the component constitute the interface of the component. The resources that a component provides (i.e., the direction specifier in a port specification is *OUT*) are called the capabilities of the component, whereas the resources that a component requires (i.e., the direction specifier in a port specification is *IN*) are called the assumptions of the component. An implementation

of a component should conform to its interface. We define conformance of a component to its interface as follows.[1]

**Definition 13 (Conformance of components to interfaces)** *A component $M$ 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 component of $I$, and $I$ is a conformed interface of $M$.*

The above conformance definition captures the notion that a conformed interface of a component should completely describe the possible interactions (both providing resources and requiring resources) that the component may have with its environment. At first, it may seem that condition (2) is too strong, however it does ensure that a component interacts with its environment exactly in the same way as it is specified in its conformed interface. An alternative to condition (2) is:

> (2') *In order to implement the capabilities of $I$, $M$ may only use assumptions stated in $I$.*

Condition (2') is more flexible, but may cause the change of a system architecture. We discuss this issue further in later sections.

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 component may conform to multiple interfaces, each of which describes a collection of possible behaviors (providing resources and/or requiring resources) that the component may exhibit in constructing a system.

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

In this chapter, we present an interface generality relation that provides a way to determine the conformance of a component to a given interface. The novelty of our approach is its consideration of the dependences among ports that enable us to capture the semantic relations between coarse-grained, architectural components. In order to facilitate our discussion, we focus on components that exchange resources of type *computation* (FUNC), *data* (DATA) and *abstract data type* (ADT), since these are the most commonly exchanged resources. However, our method should be applicable to other types of resources.

## 7.1 Generality Relation of Function Specifications

In this section and following sections, based on the result of Chapter 6 we establish generality relations between resources. We first define the generality relation of two function specifications.

**Definition 14** *(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* $(T_g^1, T_g^2, \ldots, T_g^n)$*, and that of h be* $(T_h^1, T_h^2, \ldots, T_h^n)$*, then there exists a permutation of the list* $(T_h^1, T_h^2, \ldots, T_h^n)$*, denoted as* $(T_h^{1'}, T_h^{2'}, \ldots, T_h^{n'})$*, 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. $post(h) \wedge pre(g) \rightarrow post(g)$, the conjunct of $h$'s postcondition and $g$'s precondition implies the postcondition of $g$.*

The signature matching process requires that the two functions' range types match and their domain types match after permutation. While there are some essential limitations with signature matching as discussed in Section 6.6.2 of Chapter 6, we use signature matching as a preprocessor to facilitate specification matching. In certain cases where signature matching is too restrictive, we may consider the remedy proposed in Section 6.6.2 of Chapter 6. The specification rule requires that the most general reuse-ensuring match, *relaxed-plug-in* match (see Theorem 7), holds when applied to the two functions.

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.

## 7.2 Generality Relation of Data Specifications

The generality relation of data items or abstract data types (ADTs) is determined based on their behavioral specifications.

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

**Definition 15** *(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.*

## 7.3 Generality Relation of Interfaces

An interface specifies a set of provided resources (capabilities) and a set of required resources (assumptions) that are needed for providing all of the capabilities. The assumptions for a specific capability are specified in the constraints of the interface. An assumption for a specific capability is either an assumption of the interface or another capability of the interface. The relations among the resources (both provided and required) of an interface can be depicted using a *directed acyclic graph* (DAG), called a dependence DAG. For instance, Figure 7.1 is the dependence DAG of the interface *dataAccess* depicted in Figure 5.4.

In Figure 7.1, there are three nodes, each of which corresponds to a port (or more precisely, the resource that the port provides or requires). The dashed circle (node) represents an assumption (a required resource), whereas the solid circle (node) represents a capabil-

[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 [97, 98].*

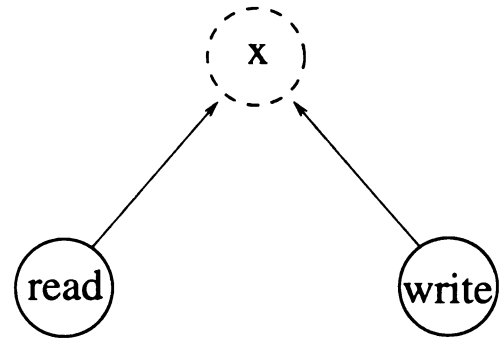



Figure 7.1: Dependence DAG of dataAccess interface

ity (a provided resource). The directed edge $\langle \overrightarrow{u,v} \rangle$ means that $u$ depends on $v$ ($v$ is an assumption needed for providing $u$).

**Definition 16** *(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 resource of $I$ defined both in terms of the capabilities and the assumptions corresponds to a node in $V$ (we name the node using its resource name). If a resource $u$ depends on another resource $v$, then directed edge $\langle \overrightarrow{u,v} \rangle \in E$.*

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 17** *(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 18** *(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 19** *(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.**

  *For all $s_2$, $s_2 \in C_{I_2}$, 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, for all $s_1$, $s_1 \in A_{I_1}^{\pi_{\mathbf{c}}(C_{I_2})}$, 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 19 has the following property:

**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 component* $M$ *conforming to* $I_1$ *also conforms to* $I_2$.

We justify this claim below. This justification consists of two parts, each of which justifies one of the two conditions required by the conformance relation between components and interfaces (see Definition 13).

1. *M implements all of the capabilities of* $I_2$.

   Let $c_2$ be an arbitrary capability of $I_2$. Since $I_1 \preceq I_2$, the definition of the interface generality relation states that there must be a capability $c_1$ specified in $I_1$, such that $c_1 = \pi_{\mathbf{c}}(c_2)$, where $\pi_{\mathbf{c}}$ is the map described in Definition 19. Since $M \models I_1$, $M$ provides the implementation of $c_1$, denoted as $IMP_{c_1}$. On the other hand, according to the capabilities rule, $c_2$ must be more general than $c_1$. Based on the generality relation definitions for data and function specifications, it is not difficult to show that $IMP_{c_1}$ also implements $c_2$.

2. *In order to implement the capabilities of $I_2$, $M$ must use and only use the assumptions stated in $I_2$.*

   Clearly, each capability $c_2$ of $I_2$ can be implemented by the same implementation $IMP_{c_1}$ for $c_1$ of $I_1$, where $c_1 = \pi_{\mathbf{c}}(c_2)$. To implement all of the capabilities of $I_2$, $M$ will use and only use all of the assumptions in $A_{I_1}^{\pi_{\mathbf{c}}(C_{I_2})}$. According to the assumptions rule, for each assumption $a_1$ in $A_{I_1}^{\pi_{\mathbf{c}}(C_{I_2})}$, there is a corresponding assumption $a_2$ in $A_{I_2}$ that is more specific than $a_1$. Therefore, $a_2$ can be used to fulfill $a_1$ when $M$ is used to implement the capabilities of $I_2$. Note that the map $\pi_{\mathbf{a}}$ from $A_{I_1}^{\pi_{\mathbf{c}}(C_{I_2})}$ to $A_{I_2}$ is onto, that is, every assumption in $A_{I_2}$ will be used to fulfill some assumption in $A_{I_1}^{\pi_{\mathbf{c}}(C_{I_2})}$.

As shown above, the requirement that map $\pi_{\mathbf{a}}$ in Definition 19 be "onto" ensures that Condition (2) in the conformance definition (Definition 13) is satisfied. If we use Condition (2') instead of Condition (2) in defining conformance (see the discussion following Definition 13), then we do not need the restriction that the map $\pi_{\mathbf{a}}$ be "onto" in Definition 19.

## 7.4 Determining Reusability

The interface generality relation provides a basis for determining the reusability and substitutability of components. For example, in software maintenance, we may often encounter the question: can a given software component be replaced by another (perhaps a new version, or a faster one, etc.) without affecting the observable behavior of the entire system? The interface generality relation provides a rigorous way to answer this type of question.

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

**Definition 20** *(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, if $I_m \preceq I$, where $M \models I_m$, then $M \models I$, 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 $I$.

There usually exist naming conflicts between $I_m$ and $I$, that is, the pairs of resources that have been mapped by the user have different names. In this case, code necessary for resolving the naming conflicts may be automatically generated.

Next, we illustrate the interface generality relation with examples. In these examples, we consider the selection of existing components for implementing the *dataAccess* component specified in Figure 5.4. For convenience, we denote the interface of *dataAccess* as $I_{access}$.

**Example 11** *The first existing component being considered has the same assumptions as specified in $I_{access}$, but provides more capabilities than that specified in the abstract component. Figure 7.2 depicts the interface of this component, $M_1$, (the interfaces denoted as $I_1$).*

The omission of the specifications means that they are the same as those specified in *dataAccess*. However, component $M_1$ has an additional capability for counting the total number of all entries in table $x$.

According to the interface generality relation definition, it is easy to see that $I_1 \preceq I_{access}$. Therefore, we can integrate component $M_1$ for *dataAccess*. This type of integration is consistent with our intuition. That is, we can use the *read* and *write* capabilities of $M_1$ to fulfill the $I_{access}$ specification, and simply ignore the *total* capability provided by $M_1$.

```
COMP  M1 {
    uses Table(dtModel for Tab, int for Ind, int for Val);
    PORT x : DataUse {
       DATA x : dtModel WITH BEHAVIOR {
             ... ...
          }
      }
    PORT read : ProcDef {
       FUNC read(int recNo, int recVal) return bool  {
             ... ...
          }
      }
    PORT write : ProcDef {
       FUNC write(int recNo, int recVal) return void {
             ... ...
          }
      }
    PORT total : ProcDef {
       FUNC total() return int {
             ensures result = size(x);
          }
      }
    CONSTR { DEPENDENCE { {read, write, total} : x } }
 }
```

Figure 7.2: Component $M_1$ interface specification

**Example 12** *Next consider another existing component, $M_2$, that does not require any service from its environment. The interface of this component (denoted by $I_2$) is specified in Figure 7.3.*

$M_2$ has the same capabilities as specified in $I_{access}$. However, $M_2$ does not require its environment to provide data $x$. That is, $M_2$ implements the data item internally.

```
COMP  M2 {
    uses Table(dtModel for Tab, int for Ind, int for Val);
    PORT read : ProcDef {
       FUNC read(int recNo, int recVal) return bool  {
             ... ...
          }
      }
    PORT write : ProcDef {
       FUNC write(int recNo, int recVal) return void {
             ... ...
          }
      }
 }
```

Figure 7.3: Component $M_2$ interface specification

According to the interface generality relation definition, $I_2 \preceq I_{access}$ does not hold, since $A_{I_2}^{\pi_c(C_{I_{access}})}$ is empty, there does not exist an onto map $\pi_a$ from $A_{I_2}^{\pi_c(C_{I_{access}})}$ to $A_{I_{access}}$. Therefore, we cannot integrate $M_2$ for *dataAccess*.

Suppose we do not require that the map $\pi_a$ in Definition 19 be onto (accordingly, Condition (2'), instead of Condition (2), is used in the conformance definition (Definition 13)), then in this case, we have $I_2 \preceq I_{access}$, and thus allowing $M_2$ to be integrated for *dataAccess*. But since $M_2$ does not require any data from other components, $M_2$ will interact with its environment in a different way than what is expected.

# Chapter 8

# An Architecture-based Reuse and Integration Environment

In this chapter, we describe the design and application of ABRIE, an architecture-based software engineering environment. ABRIE has been developed as a prototype system to validate our theoretical results, as well as serve as a testbed for our case study. This chapter consists of two parts. In the first part, we describe the rationale underlying the design of ABRIE. Our description focuses on the architectural principles of the ABRIE design, and analyzes how they influence the reusability, evolvability and maintainability of ABRIE. In particular, we show how the principled use of software architectural knowledge has facilitated the evolution of ABRIE throughout this project. In the second part, starting with Section 8.4, we describe the use of ABRIE in facilitating component-based software development. Through examples, we illustrate the working process of ABRIE and show how ABRIE supports the architecture-based component assembly framework proposed in this research.

## 8.1 ABRIE: An Introduction

ABRIE (Architecture-Based Reuse and Integration Environment) is an experimental platform designed to explore the use of software architectures as a framework for assembling software components [33]. Given that ABRIE is a prototyping system for this research, the main design objective is to provide an integrated environment to address various component reuse and integration issues: component composition specification, component management and reusability analysis. However, ABRIE has broader applicability than component reuse and integration due to its layered architecture. As we will discuss in later sections, the object-oriented models of architectural elements that constitute the bottom layer of ABRIE provide a reusable toolset for creating software architecture related tools.

In terms of serving as a validating system for this research, 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 compositions (architectures) are supported. ABRIE uses three levels of abstraction in determining the reusability of existing components: (component) types, signatures, and formal specifications. Automation is one main potential benefit of architecture-based reuse. In ABRIE, the component integration process is fully automated. In the reusability analysis of components, automated support is provided through the integration of third-party automated reasoning tools. ABRIE consists of three functional components: architecture design, component management, and system packaging. All of them are integrated and provided through a user-friendly graphical interface.

## 8.2 Design Objectives

In addition to the functionalities mentioned in the previous section, that is, supporting user-friendly visual architectural manipulation, reusability evaluation and integration, there are several important non-functional requirements that have critically influenced the design of ABRIE. The first one is evolvability. As a research prototype, ABRIE has to be sufficiently flexible to accommodate changes and new features proposed throughout the research process. Another requirement is reusability. Reusability is important because one goal in designing ABRIE is to provide a set of components that can be rapidly assembled to implement various prototyping systems for specific purposes regarding architecture-based component reuse. Finally, ABRIE should be open for the integration of third-party tools. Since the component interconnect model (see Chapter 5) integrates third-party specification methods, such as Larch specifications, ABRIE's design should be made open for the integration of tools for syntax checking and the analysis of those specifications.

In order to accommodate these properties, they must be explicitly designed into the system. One set of design issues that may significantly determine non-functional properties like evolvability and reusability involve the overall organization of a system, that is, the architectural design of the system.

## 8.3 ABRIE Architectural Design

Architecturally, ABRIE is a layered system. Figure 8.1 shows the layers of the system. As depicted in Figure 8.1, ABRIE has three layers, each of which uses the services provided by its immediate lower layer and provides services to its immediate upper layer. Specifically, the bottom layer is the foundation models for the ABRIE environment, which are

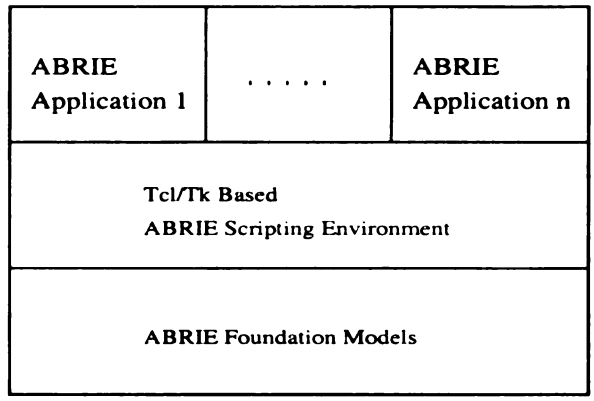



Figure 8.1: Layered architecture of ABRIE

object-oriented models of architectural elements that define the abstractions of resources, ports, components, connectors, constraints, configurations, and so on. The second layer is a Tcl/Tk based ABRIE scripting environment. Tcl (shorthand for Tool Command Language) [115, 116, 117] is an open source scripting language and environment. Tk (shorthand for Tool Kit) [118, 116, 117] is an extension to Tcl that supports rapid graphical user interface (GUI) development by providing a collection of graphical widgets. Tcl/Tk is extensible in that domain-specific commands can be easily integrated into its programming environment. Tcl/Tk can also be embedded into applications due to its simplicity. Tcl/Tk is often used as an integration platform to integrate disparate software technologies, including GUIs, middleware, existing software components, and Internet and networking protocols. In the design and implementation of ABRIE, we extend the standard Tcl/Tk scripting environment (interpreter) by integrating commands that manipulate architectural elements. In the extended Tcl/Tk environment, the implementation of those ABRIE-related commands are directly based on the object-oriented models provided in the bottom layer. As shown in Figure 8.1, upon the second layer, applications for specific purposes can be developed through

Tcl/Tk scripting (programming). In this research, we developed a reuse and integration application that provides graphically user-friendly manipulation of component composition, evaluation, and integration.

Several design decisions underline our objectives regarding evolvability and reusability. The main motivating factor for choosing a layered architecture is the principle that a component at a particular layer only uses services provided by those at the same or immediately adjacent lower layer.Therefore, each layer is insulated from changes made in distant layers. [1] This organizational principle promotes information hiding between layers and increases evolvability. In the design of the **ABRIE Foundation Models**, we intentionally leave out any bias towards a specific use of this layer. The result is a design that is generic and thus reusable for a variety of applications involving architecture manipulation. A similar philosophy has been applied to the design of the second layer, the **ABRIE Scripting Environment**.

We chose Tcl/Tk as the platform for scripting applications based on the following reasons: First, serving as a glue language, Tcl/Tk has a simple yet powerful syntax, as well as a small and extensible kernel implementation. In addition, the Tk part largely simplifies the development of graphical user interfaces. Adding to the advantages is the platform independent scripting environment of Tcl/Tk that makes the applications scripted using Tcl/Tk operate on different platforms. Scripting also allows the rapid development of an application.

[1] Generally speaking, there are two kinds of layered architectures: open and closed. In an open architecture, a component at a particular layer can use services provided by those components at any lower layer. Whereas in a closed architecture, a component can only use services provided by those components at the same or immediately adjacent lower layer. In this chapter, when we discuss layered architectures we mean closed ones.

### 8.3.1 Object-Oriented Modeling of ABRIE

In order to better understand the requirements of ABRIE and manage the complexity encountered in the design, implementation, and maintenance of ABRIE, we have applied object-oriented techniques throughout the development of ABRIE. In particular, we use the *Object Modeling Techniques* (OMT) [9] to model various aspects of ABRIE. OMT is one of several widely used object-oriented modeling techniques [9, 10, 119]. It consists of three complementary models. The *object model* describes what a system is by modeling the static, structural data aspects of the system. It captures the objects of the system and the relationships between the objects. The *dynamic model* depicts when system activities occur, that is, the temporal and behavioral aspects of the system. Finally, the *functional model* describes the services provided by the system by giving the transformation of the data within the system. Respectively, entity-relationship diagrams, state transition diagrams, and data flow diagrams are used to represent the object, dynamic, and functional models.

Figure 8.2 depicts the system level object model of ABRIE using OMT notations. In Figure 8.2, rectangular boxes represent classes. The line connecting these two classes asserts the existence of a relationship; the diamond denotes the *aggregation* relationship, where the class touching the diamond is the aggregate. The filled circle at the opposite end of the line denotes "*many*", where *many* means zero or more; the absence of a filled circle at the endpoint of a line indicates "*one*".

As shown in Figure 8.2, we refine the layered architecture depicted in Figure 8.1 by identifying objects (classes) and their relations. An ABRIE application may have zero or more GUIs, and both the application itself and its GUIs are scripted (programmed) using the Tcl/Tk based ABRIE scripting environment. The Tcl/Tk based ABRIE scripting environment is an ABRIE-specific extension to Tcl/Tk. That is, in addition to the original Tcl/Tk

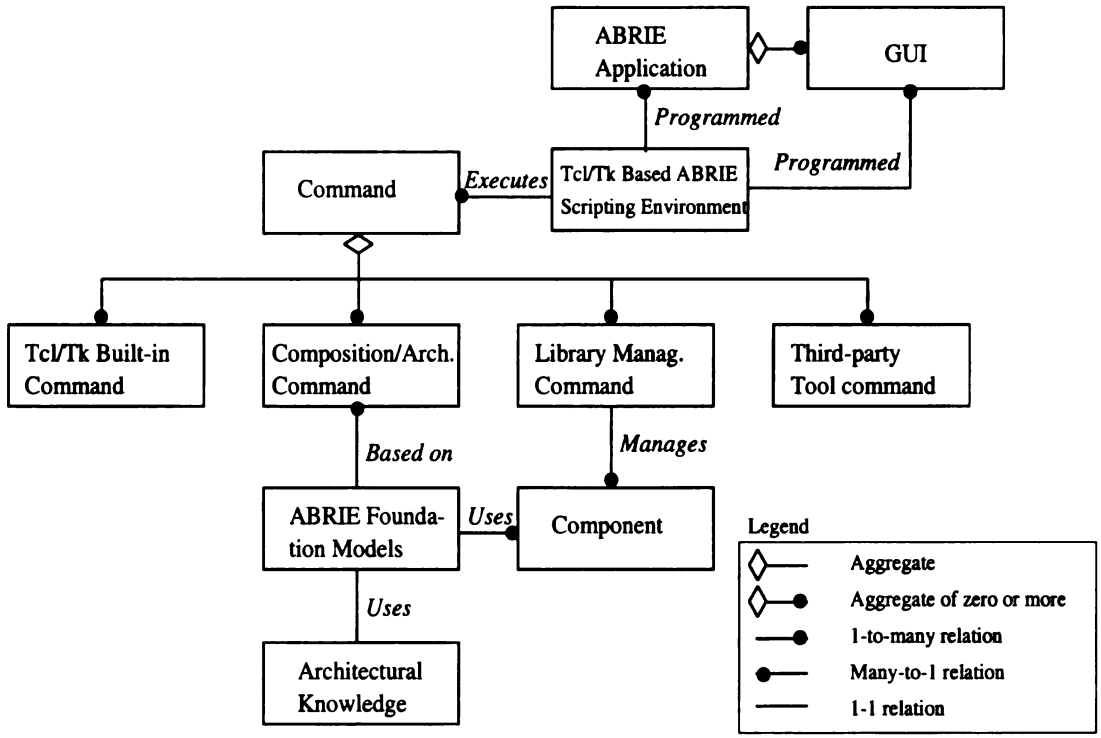



Figure 8.2: System level object model of ABRIE

commands, ABRIE-specific commands are integrated into the Tcl/Tk runtime interpreting system and therefore can be invoked by applications. These ABRIE-specific commands constitute the ABRIE Application Programming Interface (API). Several types of ABRIE-specific commands are depicted in Figure 8.2. The composition (architecture) commands are responsible for manipulating components, connections and compositions. These commands are the main part of the ABRIE API. The library management commands manage reusable components, including classification, retrieval, and evaluation of these components. Finally, third-party tool commands can also be integrated in the ABRIE API. These commands provide a vehicle to use third-party tools, such as the Larch Prover [37] that assists

the evaluation of component reusability. As shown in Figure 8.2, ABRIE API is based on ABRIE foundation models. We will discuss these models in detail in the next subsection.

One issue regarding the design of a software architecture environment is relevant to the immaturity of the software architecture discipline itself. For example, while it is increasingly evident that distinguishing styles among architectures is very important, there is no set of consensus criteria to perform the distinction, and discovering and documenting architectural styles are still active areas of research. Therefore, ABRIE needs to be able to accommodate new development and findings in software architectures. As shown in Figure 8.2, a class that embodies the generic architectural knowledge is introduced. This knowledge, described in a meta-language, is loaded into the system whenever the system is started and is used by the ABRIE foundation models. Since it is straightforward to edit the knowledge description file, ABRIE can easily embody new advances made in software architectures. This approach also makes it feasible to customize the ABRIE environment for supporting specific styles.

### 8.3.2 ABRIE Foundation Models

It is straightforward to model architectural elements using object-oriented classes. Figure 8.3 depicts the object models of the ABRIE foundation elements using the OMT notation. Figure 8.3 uses a feature of OMT notation that has not been explained before. In OMT, rectangular boxes, representing classes, can be partitioned into three layers: the top layer provides the name of the class, the middle layer lists attributes of the class, and the bottom layer enumerates operations associated with the class.

As shown in Figure 8.3, architectures are modeled using the **ArchCls** class that is an aggregate of classes **CompCls** and **ConnCls**. **CompCls** models components and is an

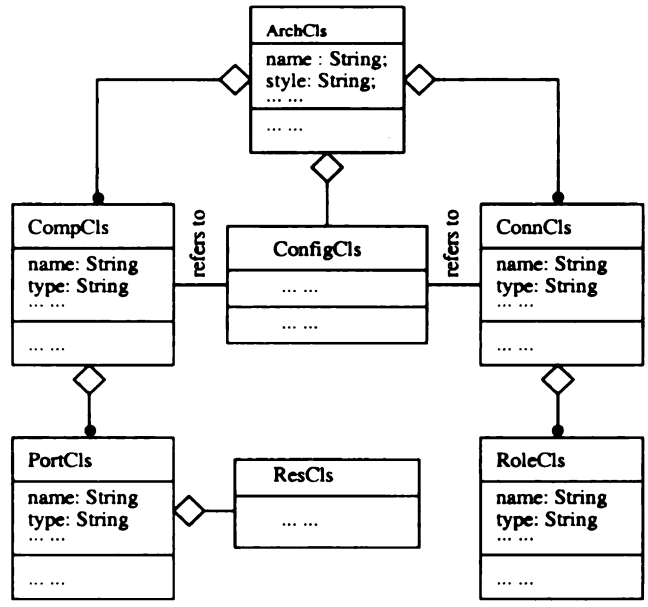



Figure 8.3: Object model of ABRIE foundation elements

aggregate of ports modeled by class **PortCls**. Similarly, **ConnCls** models connections and is an aggregate of class **RoleCls** that models roles.

The introduction of class **ConfigCls** merits further discussion. Unlike other architectural elements, configuration is a relation between components and connections, rather than an entity. Thus a natural and intuitive approach to describing configurations seems to be to annotate components and connectors with configuration information. Unfortunately, such an approach has the configuration information distributed among the components and connectors, which makes the design difficult to understand. Consequently, it is hard to maintain and evolve such a design. For example, consider the addition or deletion of a configuration: this type of change will involve operations for both the component and connection concerned; and the configuration information in both entities has to be updated and kept consistent. Introducing a class **ConfigCls** to encapsulate and localize all the config-

uration information in an architecture largely simplifies the management of configurations, and facilitates the evolution.

A port is the point through which a component exchanges resources with its environment. Therefore, each **PortCls** object has a resource modeled by class **ResCls**. There are several kinds of resources, such as computation (function), data, abstract data type (ADT), or event. Figure 8.4 depicts the object models of these resources and their relationships using OMT notations.

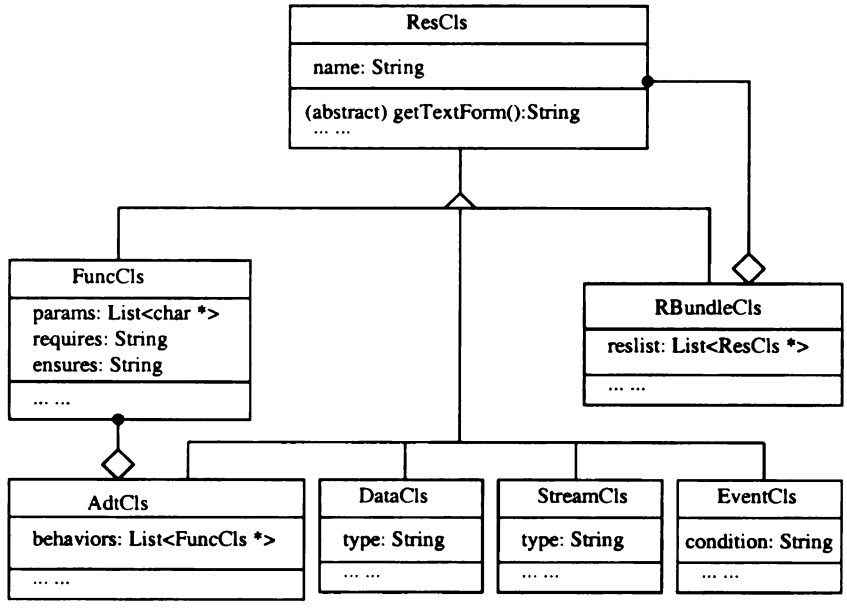



Figure 8.4: Object model of resources

The **ResCls** class demonstrates another type of relation between classes used by the OMT notation. Namely, the inherits or "isa" relation, indicated by a triangle. The class touching the triangle is the superclass, whereas the class at the other end of an inheritance relation is the subclass. As shown in Figure 8.4, there are six kinds of resources modeled: function or computation modeled by class **FuncCls**, abstract data type (ADT) modeled

by **AdtCls**, data item modeled by **DataCls**, data stream modeled by **StreamCls**, event modeled by **EventCls**, and a bundle of resources modeled by **RBundleCls**. Class **ResCls** is the superclass of all these resource modeling classes and captures the commonalities of these classes. Behaviors and attributes of class **ResCls** are inherited by all its subclasses.

### 8.3.3 ABRIE Scripting Environment

**ABRIE Scripting Environment** is an extension to the Tcl/Tk interpreter [116]. As mentioned earlier, such an environment provides a powerful yet flexible way to developing specific applications. The extended commands constitute the application programming interface (API) of ABRIE. The API enables the creation and manipulation of architectural elements, the management of reusable component libraries, as well as the integration of third-party tools. As with the design of the **ABRIE Foundation Models**, the API of the scripting environment is designed without any bias towards specific applications. Nonetheless, commands customized for particular applications can be easily added to the scripting environment given the extensibility of Tcl/Tk.

The ABRIE API provides commands to manipulate the following entities: architecture (composition), component, connector, role, port, library, and the ABRIE meta-system. General operations include creating, inquiring, modifying, and the input/output of an entity. Figure 8.5 shows the command `cfgArch` that manipulates architectures. In Figure 8.5, a semicolon (;) denotes a comment. The identifier prefixed with a dash (-) is a switch or a subcommand. Parameters in a pair of square brackets ([ ]) are optional. As an example, the command

```
cfgArch -addComp container ADT
```

```
cfgArch  ;manipulating (current) architecture
  -name [archname] ;return or set the name
  -type [typename] ;return or set the type
  -addComp compname comptype ;add a component
  -addConn connname conntype ;add a connector
  -compList ;return the list of components
  -connList ;return the list of connectors
  -delComp compname ;delete a component
  -delConn connname ;delete a connector
  -save filename ;save the architecture
  -load filename ;load the architecture
```

Figure 8.5: Sample ABRIE scripting environment API command

will add the component **container** of type **ADT** to the *current* architecture. When an architecture is created (using **crtArch** command) or loaded (using **cfgArch -load** command) from a file, it becomes the *current* architecture of the ABRIE scripting environment until another architecture is created or loaded. All architecture-related operations are, by default, oriented towards the *current* architecture.

### 8.3.4 Discussion

As mentioned earlier, evolvability is one of the main design objectives. We specifically chose a layered organization to enable the accommodation of new demands. As we expected, throughout the research process, many changes have been needed and new requirements have been proposed. One major change is the development of the component interconnect model and language (see Chapter 5). The original ABRIE system was based on a relatively simple ADL. In that ADL, there was no concept of *resources*. The types of port supported were limited to a small set. The whole framework of the ADL had not been well-founded yet.

After the development of the new ADL, we began to evolve ABRIE to accommodate the new ADL. Since only the bottom layer, the **ABRIE Foundation Layer**, involves the modeling of architectural elements, the changes were limited to that layer. By preserving the old API in the ABRIE scripting environment while adding new commands, "legacy" applications are ensured to run correctly, as in the case of our reuse and integration application.

## 8.4 Application: Architecture Design

In this section and the following sections, we describe the application of ABRIE to support the architecture-based component assembly framework proposed in this research. The main working area of ABRIE is a canvas that provides a graphical representation and manipulation of architectural elements. Figure 8.6 shows a pipeline architecture for an example system, *palindrome_word_counter (pwc)* that counts the number of palindrome words in a file, where boxes represent components and bars represent connectors.

Component *pwr* is a filter for recognizing palindrome words, and filter *wordcounter* counts the words recognized by *pwr*. These two filters are connected by a pipe connector *wf2*. In ABRIE, components and connectors are associated with actions for viewing, editing, and configuring them. These actions can be activated through a menu displayed while clicking on a component or a connector. Figure 8.6 shows the "connector property" window for connector *wf2* and the "component property" window for component *pwr*. Component and connector properties can be edited through these windows.

Architectures can also be edited using text editors. ABRIE will generate the graphical representation automatically. Figure 8.7 shows the textual representation of architecture *pwc*.

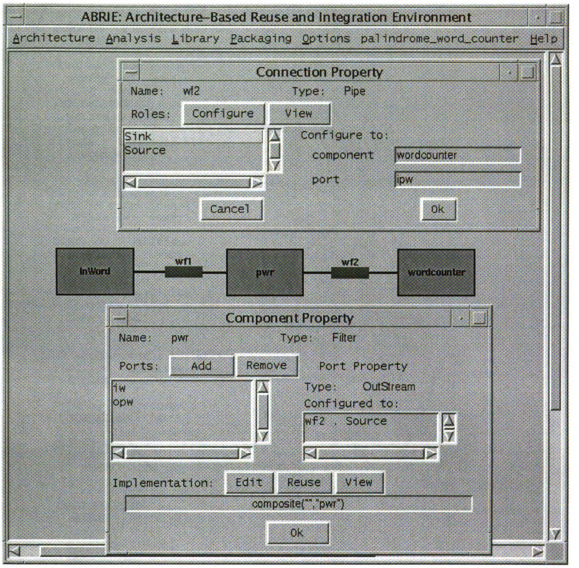



Figure 8.6: ABRIE architecture design

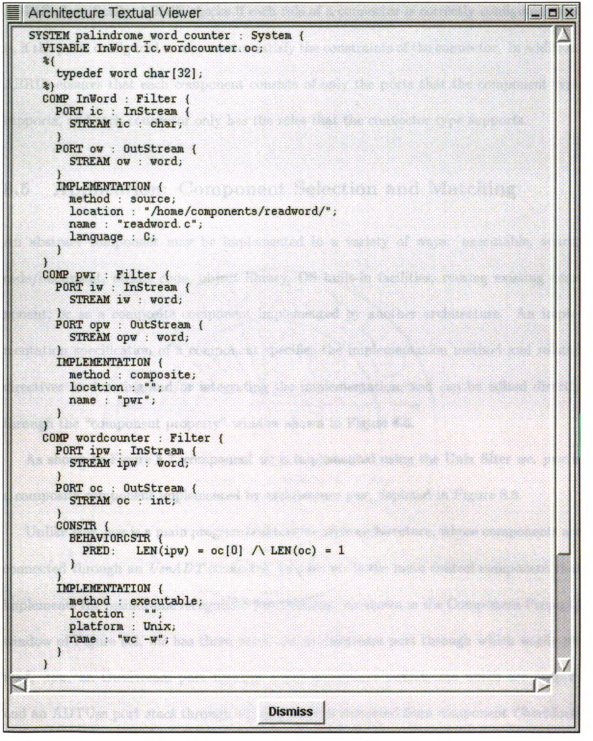



Figure 8.7: Textual representation of architecture *pwc*

The correctness of the connection configuration of an architecture can be analyzed by ABRIE. Specifically, ABRIE checks if each role of a connector is correctly configured, that is, if the ports configured to a connector satisfy the constraints of the connector. In addition, ABRIE ensures that each component consists of only the ports that the component type supports, and each connector only has the roles that the connector type supports.

## 8.5 Application: Component Selection and Matching

An abstract component may be implemented in a variety of ways: executable, source code/document, object code, object library, OS built-in facilities, reusing existing component, or as a composite component implemented by another architecture. An implementation specification of a component specifies the implementation method and related directives for locating and/or integrating the implementation, and can be edited directly through the "component property" window shown in Figure 8.6.

As shown in Figure 8.7, component *wc* is implemented using the Unix filter *wc*. *pwr* is a composite component implemented by architecture *pwr*, depicted in Figure 8.8.

Unlike *pwc*, *pwr* is a main program/subroutine style architecture, whose components are connected through an *UseADT* connector. In *pwr*, *mc* is the main control component that implements the palindrome recognition functionality. As shown in the Component Property window of Figure 8.8, *mc* has three ports: *iw*, an InStream port through which words are read; *opw*, an OutStream port through which recognized palindrome words are output; and an ADTUse port *stack* through which a stack is imported from component *CharStack*. Component *CharStack* defines and exports an ADT *cstack*. The implementation of each component in *pwr* may take various forms as discussed before. Next we describe how ABRIE supports the reuse of existing components.

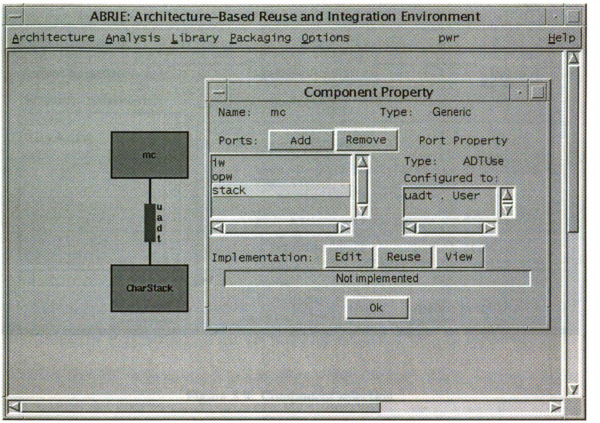



Figure 8.8: Architecture of pwr

ABRIE incorporates a library manager for organizing and managing existing components. Components are classified and retrieved based on their interfaces (i.e., types and ports). When implementing an abstract component in an architecture, a single click on the *reuse* button in the "component property" window (see Figure 8.6) triggers ABRIE to search for the current library (which is loaded through the library manager) and a component selection window is displayed as shown in Figure 8.9.

In the "component selection" window, all candidate components that may match the abstract component are retrieved and presented to the user. For example, in Figure 8.9, three

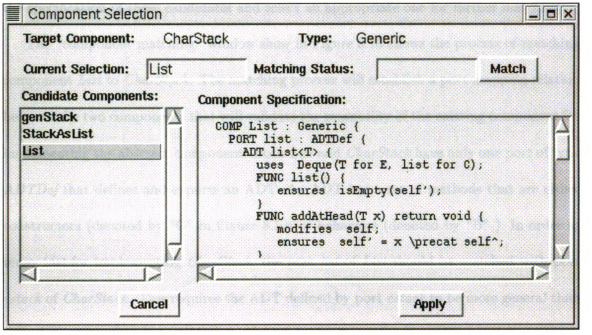



Figure 8.9: Component selection

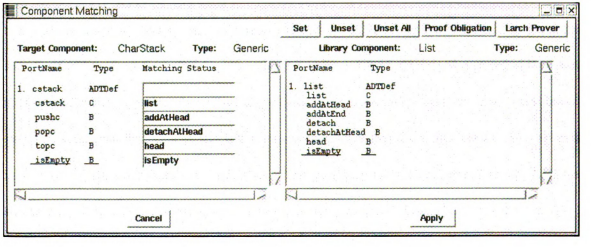



Figure 8.10: Component matching

components are listed as candidates for implementing *CharStack* of *pwr*. Users can browse the specification of these candidates and select an appropriate one for further matching.

The "component matching" window show in Figure 8.10 shows the process of matching component *List* to *CharStack*. The matching process will establish a port mapping relation between the two components that will validate the reusability of the existing component for implementing the abstract component. Both *List* and *CharStack* have only one port of type *ADTDef* that defines and exports an ADT. An ADT has a set of methods that are either constructors (denoted by "C" in Figure 8.10) or behaviors (denoted by "B" ). In order to reuse *List* for implementing *CharStack*, the port *list* of *List* should be matched with port *cstack* of *CharStack*. This requires the ADT defined by port *cstack* to be more general than that defined by port *list* of *List*. In the following discussion, we refer to the two ADTs as *cstack* and *list*, respectively. The generality relation between ADTs requires that for each method of *cstack*, there is a more specific method in *list*. We proceed by assigning a mapping between the methods of the two ADTs. As shown in Figure 8.10, we assume that the constructor *cstack* is more general than constructor *list*, *pushc* than *addAtHead*, *popc* than *detachAtHead*, and so on. Given the mapping, ABRIE will automatically generate the proof obligations for justifying the mapping. Figure 8.11 depicts the proof obligations generated for the mapping shown in Figure 8.10. The proof obligations are generated based on the Larch specifications of the two components. The proof obligations are represented in terms of LSL specifications to facilitate the application of the Larch Prover (LP) for analyzing them. After preprocessing the obligations, we can invoke the Larch Prover (LP) from the "Component Matching" window to assist in proving these obligations. Figure 8.12 shows a snapshot of LP while it is discharging the proof obligations for our example.

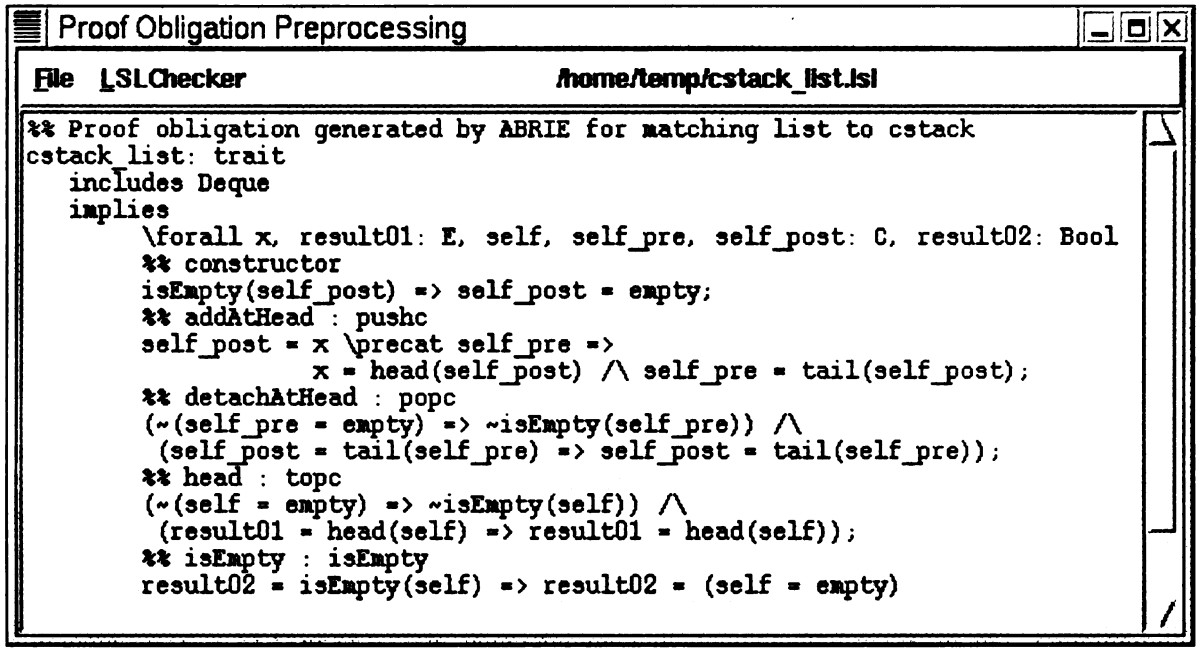



Figure 8.11: Proof Obligations

LP [26, 37] is an interactive theorem proving system for multi-sorted first-order logic. Typically user interaction may be required in proving/disproving a conjecture. However, for our example, all the obligations are automatically resolved, and the user-assigned mapping is justified, thereby establishing the matching between component *List* and *CharStack*.

Once the matching process is successfully completed, meaning that the existing component can be reused, a matching file will be generated for recording all the matching information. As shown in Figure 8.13, after the selection and matching process is finished, the implementation specification for *CharStack* specifies that it be implemented by using library component *List*. Figure 8.14 shows the corresponding matching file.

The file records the mapping between ports and methods of ports. For a parameterized port, the matching process also determines the appropriate instantiation of the parameters

```
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 8.12: A Snapshot of LP in resolving proof obligations

based on the port mapping. For example, the *list* port is mapped to the *cstack* port with its parameter $T$ instantiated to *char*.

## 8.6 Application: System Packaging

Software architectures specify the solution to an application as a logical composition of (abstract) components, and serve as the "blueprint" to component assembly. As depicted in Figure 4.4, the first step of the assembly process is to find a conforming implementation for each of those interfaces (abstract components) specified in the architecture. The inter-

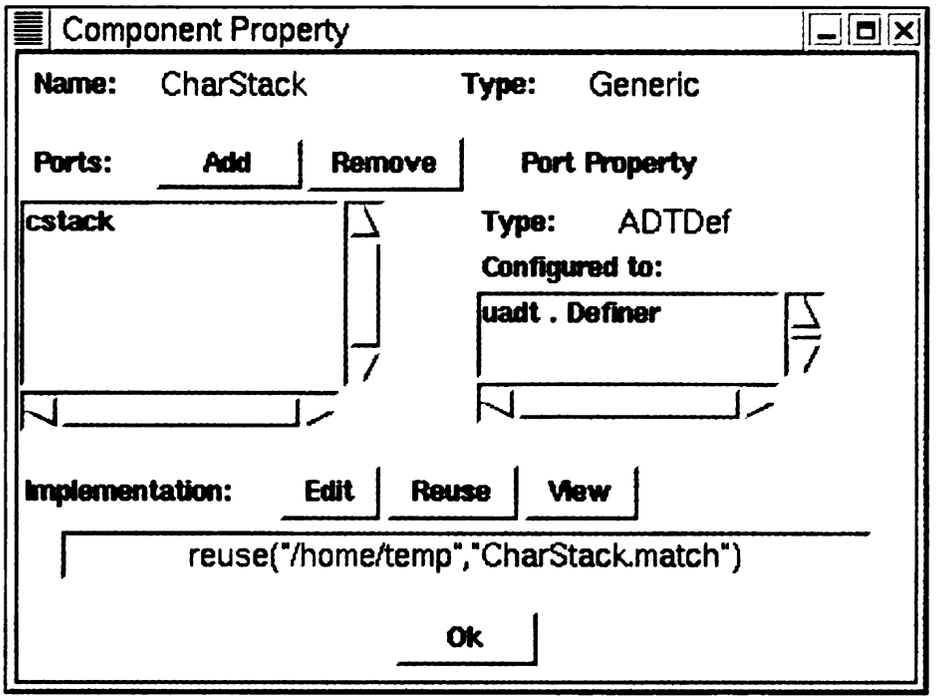



Figure 8.13: Implementation Status

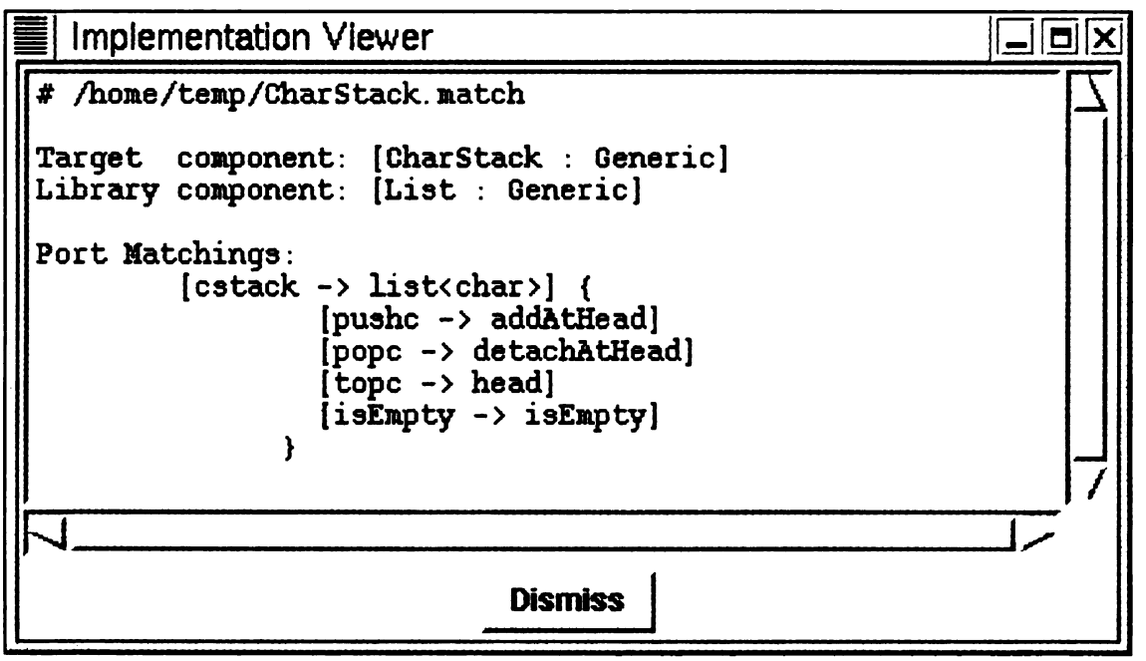



Figure 8.14: Matching file

face generality relation developed in Chapter 7 provides a semantic foundation to locate the appropriate (existing) components for reuse. Once all the interfaces have each been matched with an implementation, a packaging process is needed to physically integrate those conforming components together to form the target system.

In order for a target system to conform to its architecture, the packaging process observes the following rules:

- Each (component) interface in the architecture should physically have a conforming implementation.
- The implementation of each connector in the architecture should be explicit and packaged individually.

According to the first rule, for a reused component, we should adapt the component itself to conform to the interface to which it is matched, rather than modifying those components interacting with it. As shown in Figure 4.4, a wrapper for a reused component should be generated for accomplishing this adaptation. The second rule requires the connector implementation to be localized. These rules facilitate the automation of component adaptation and connector implementation, and most importantly ensure that the generated system has a physical structure that is well-defined (abstracted) by the architecture. The latter, in turn, yields many benefits of a component-based system such as maintenance and evolution, by providing traceability between design and implementation.

System packaging mainly handles two tasks: component adaptation and connector implementation. The packaging process checks the implementation of each component, ensuring that it conforms to the abstract component (interface). In the case that mismatches exist, software wrappers need to be generated for adapting the reused components to con-

form to the abstract components (interfaces). For the simple mismatches such as naming conflicts, adaptation can be implemented based on the port mappings generated by the component evaluation process.

The implementation of a connector depends upon its types and role configurations. The implementation for those primitive connectors is typically supported by programming languages or operating systems facilities, such as a procedure call or pipe. In this case, the implementation of a connector is either implicit or directly uses corresponding facilities. The implementation of some connectors may involve third party software, such as RPC packages and object brokers. In this case, program templates for specific connectors may be established and reused for the automatic generation of a connector implementation. In the case that a connector encapsulates protocols that are customized for a specific application, the implementation needs to be implemented manually. In addition to establishing connections, the implementation of a connector may need to resolve mismatches between the ports that are configured to the connector. A typical mismatch is a naming conflict that can be resolved by automatically generating code for mapping conflicting names.

The final system is described in a construction file (such as *makefiles* or a script file) that describes the construction process for the executables of the target system. Figure 8.15 shows the wrapper for adapting component *List* for implementing *CharStack*. The wrapper is generated based on the port mappings recorded in the matching file. Figure 8.16 shows the Unix shell script generated by ABRIE to implement architecture *pwc*.

## 8.7 Lessons Learned

Our experience with the ABRIE prototype work helps to validate our approach to component reuse. Component acquisition and use are seamlessly integrated in our framework.

```
// CharStack.h
// Generated by ABRIE for adapting List to implement CharStack

#define addAtHead pushc
#define detachAtHead popc
#define head topc

#include "/home/components/list/list.h"
#define cstack list<char>
```

Figure 8.15: Wrapper for adapting List to implement CharStack

```
#!/usr/bin/sh
# pwc
# Generated by ABRIE for implementing architecture pwc

# Create named pipe for connection wf2
mkfifo wf2_pipe
# Create named pipe for connection wf1
makefifo wf1_pipe
# Start up filter wordcounter
wc -w <wf2_pipe &
# Start filter pwr
pwr >wf2_pipe <wf1_pipe &
# Start filter InWord
readword >wf1_pipe &
```

Figure 8.16: Unix shell script for implementing pwc

Because (syntactic) mismatches are identified and recorded during the component evaluation process, adaptation can be automatically conducted based on mismatch information. Formal methods applied to software development might not be scalable in general, but our hybrid approach to integrating formal and informal methods provides the flexibility. Whenever necessary, formal reasoning automated by a theorem prover can be employed to improve the rigorousness and confidence of reusability. Although we have only implemented

primitive interaction abstractions, we have gained insights as to the principles needed to implement more complicated ones.

# Chapter 9

# Case Study

In order to gain empirical insights regarding the feasibility of our research, we conducted a case study that applied our approach to an industrial project, the *Environmental Information System* (ENFORMS) [120, 121]. ENFORMS is an object-oriented distributed multimedia information system developed by the Software Engineer Research Group (SERG) at Michigan State University over a three year period (1993-1995) involving 15 software developers. In this chapter, we describe the background, the scope, the process, and the results of this case study.

## 9.1 ENFORMS Project

During this decade, NASA will launch many new platforms into earth orbit, including the satellites that will make up the Earth Observing System (EOS). The remotely sensed data obtained from EOS can be used to promote global and national security, extend international cooperation, and improve our ability to understand and manage global environmental, economic, and social problems. In the past, NASA and other agencies have focused on the acquisition of data rather than the integration or the dissemination of data. Many organiza-

tions addressing grand challenge problems, such as those defined by earth sciences, require the integration of both physical and human resource databases in an interactive manner. Such a capability allows reasonably informed policy analysts and related staff to query an "Environmental Science Workstation" so as to better understand how human uses impact our natural resource base.

The ENvironment inFORMation Systems (ENFORMS) has been designed to provide access to data and data integration utilities for the purposes of facilitating decision making processes relevant to environmental policies. The user constraints included a user-friendly system, usable by non-computer scientists, focused search capabilities, and transparency of the distributed and heterogeneous nature of the data and analysis utilities. ENFORMS contains a wide variety of items, such as image files, research papers, executable environmental models, and research data sets. Using the graphical user interface, the user is able to examine these items interactively through the archiving software and display images and execute environmental models. Figure 9.1 shows a high-level view of ENFORMS architecture, where the rectangles represent potential participating sites, and the filled circles represent users.

ENFORMS has been developed using object-oriented technology in order to facilitate its extensibility, flexibility, and maintainability. Specifically, the *Object Modeling Technique* (OMT) [9] has been employed to conduct object-oriented analysis and design. Figure 9.2 shows the object models of an archive represented using OMT notations.

An archive is composed of a number of items, each of which has a set of operations associated with it that can be performed on it. This structure is captured in Figure 9.2 (a), which shows the basic model of an archive. Figure 9.2 (b) depicts the high level object model of ENFORMS, which extends the basic object model of an archive by describing

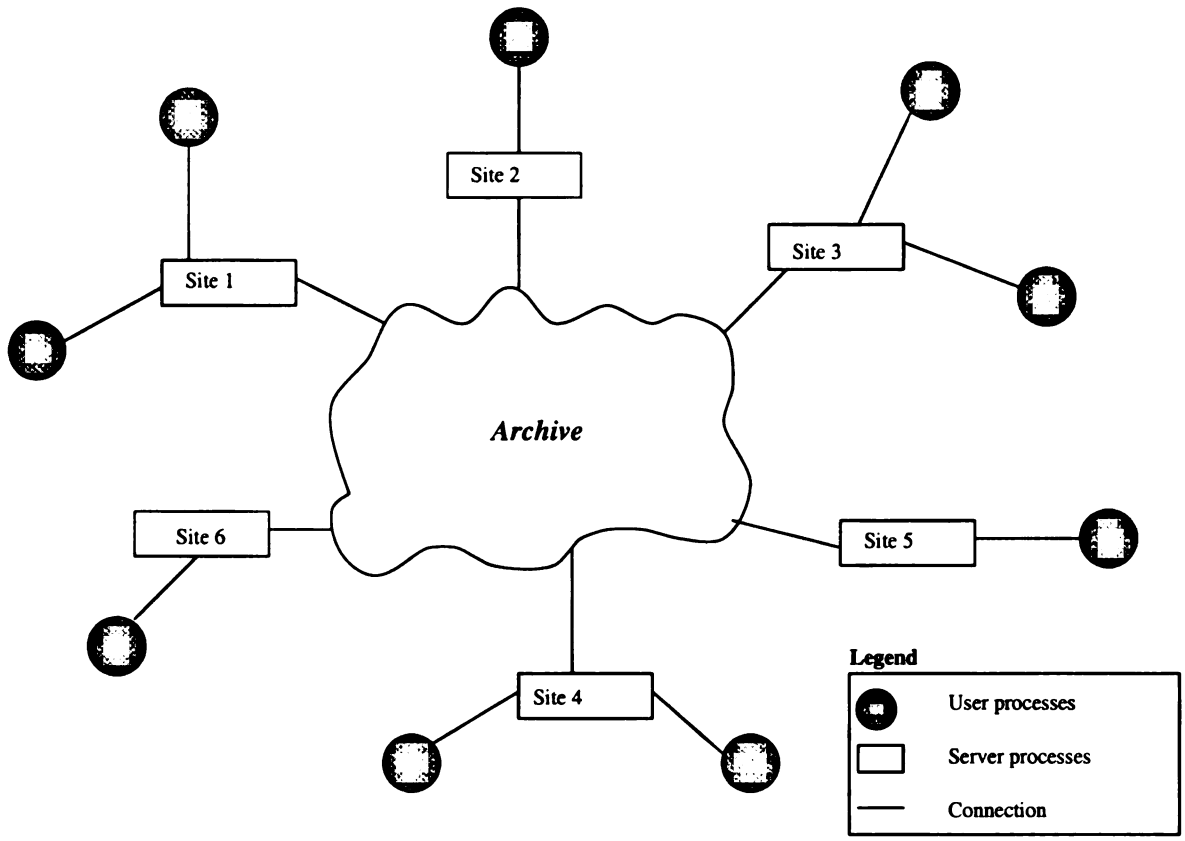
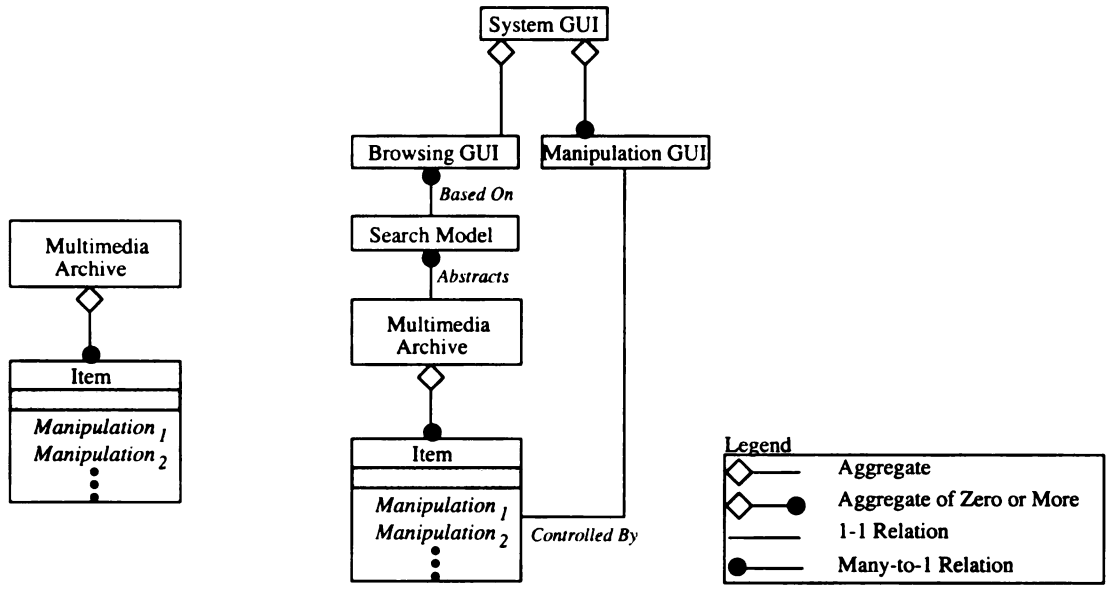



Figure 9.1: A high-level view of ENFORMS architecture.

entities that provide access to the archive. Atop the **Multimedia Archive** is the **Search Model** that provides a browsing paradigm for the archive. A search model determines how a collection of items (either atomic or aggregate) are classified so that a system can be browsed and items can be retrieved using a specific search engine. The search model also determines how new items are added to an archive. Examples of search models are *hierarchical*, *temporal*, and *spatial*. The *many-to-one* relation *Abstracts* between the **Search Model** and the **Multimedia Archive** means that "many different **Search Models** can provide abstractions of a single **Multimedia Archive**".

As shown in Figure 9.2 (b), ENFORMS has three interface entities: a **Browsing GUI**, a **Manipulating GUI**, and a **System GUI**. A given **Search Model** can have many




(a) Basic object model (b) Object model with access mechanisms

Figure 9.2: Object models of the archive

**Browsing GUIs** based upon it. **Items** are *Controlled by* multiple **Manipulation GUIs**. The interface of the entire system, the **System GUI**, is modeled as an aggregation of a single **Browsing GUI** and zero or more **Manipulation GUIs**.

In general, a multimedia archive manages access to items. This relationship is captured by modeling the **Multimedia Archive** class as a collection of **Items**, each of which has its own set of associated access methods (as shown in Figure 9.2(a)). However, archive items are actually objects that exist external to the archive software, that is, in the domain of the operating system. Given this constraint, it is intuitive to model items abstractly using indirection. Figure 9.3 shows an object model for **Multimedia Archive**, where the **Items** are indirectly managed by (*Registered By*) **Item Descriptors**.

For each **Item** in the **Multimedia Archive**, there exists an **Item Descriptor** that contains all relevant information for the **Item**, which may only be examined by allowable

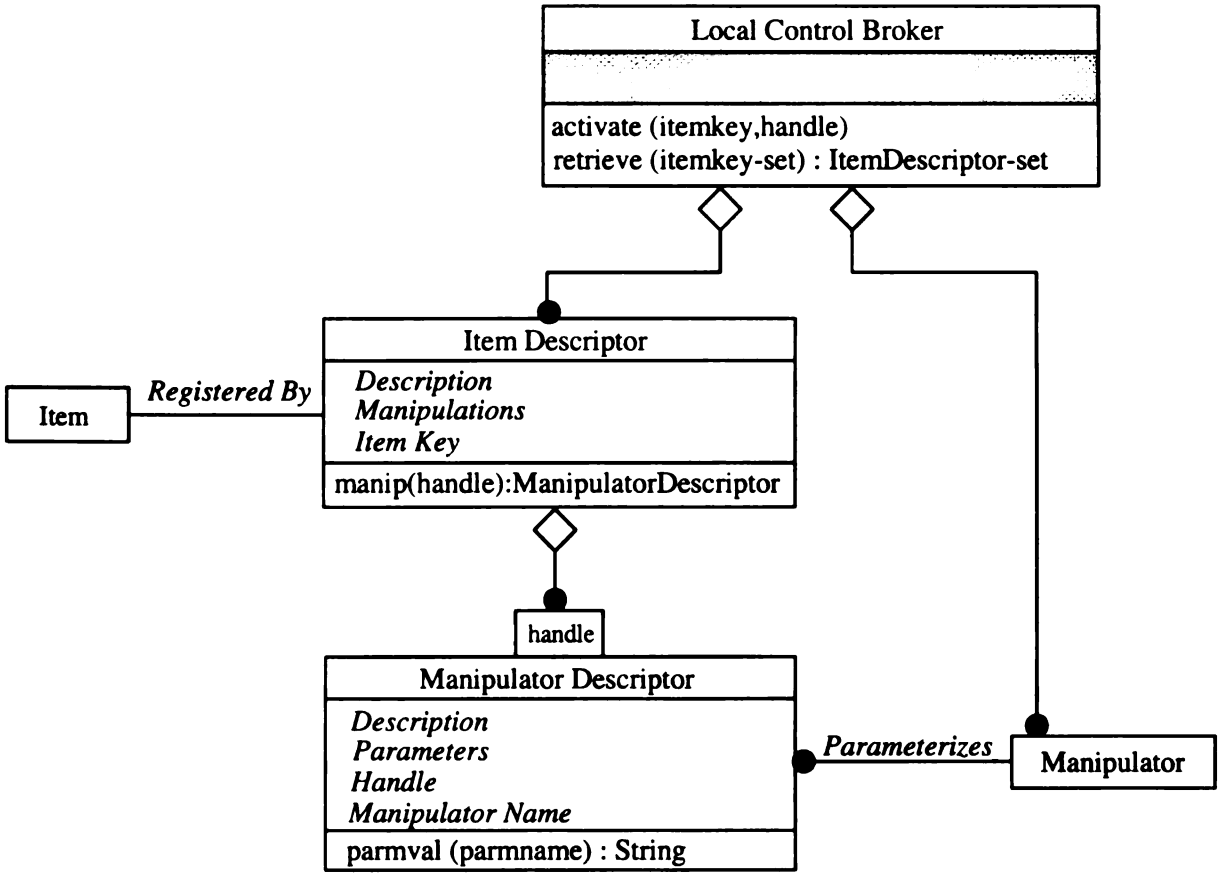



Figure 9.3: Analytic model of **Multimedia Archive**

manipulators. Manipulations for an aerial photograph item, for example, might be to *describe the photograph* and to *display the photo*, while those of a bibliography might be *search by keyword*, *sort by author*, and so on. In order to perform such manipulations, the appropriate software tools, or **Manipulators**, must be available for use.

The **Local Control Broker** realizes the conceptual aggregation of a **Multimedia Archive** and its collection of items by managing **Items** and **Manipulators**. Access is supported in two ways: through the retrieval of **Item Descriptors** and the activation of **Manipulators**. Some of the types of manipulators supported include: a GIS analysis tool, a text file display tool, an image viewing utility, an audio player, an MPEG animation

player, and a generic application launching tool for applications that have their own GUI (e.g., data analysis models, pre-defined animations, etc).

## 9.2 Focus of the Case Study

ENFORMS has been instantiated for use in four different projects, each involves a specific set of data and manipulating tools. While all the instantiations share the same general architecture described in Figure 9.2, there are some requirements and features that are specific for an instantiation. For example, while most instantiations must deal with the distributed nature of multiple archives across a network, certain instantiation only has a single site of data storage and operations that lends it to be a standalone rather than a client-server architecture. Due to the differences in the nature of data, the search models used for classifying and retrieving the data, as well as the operation semantics applied to the data, may differ radically from one instantiation to another. These differences across various instantiations present challenges in reusing the design and implementation of the shared features. In the original implementation of ENFORMS, reuse is implemented at the code level where chunk of code is copied from earlier instantiation to form the new one, and new features are added directly at the code level. As a result, not only is the effectiveness of reuse limited, there also lacks a macro and structural view of the relations between different instantiations. This in turn makes it difficult to maintain and evolve various instantiations of ENFORMS. In this case study, we apply the methods developed in this research to the ENFORMS design and implementation. We describe how our methods can lead to a reuse-oriented design, and demonstrate the positive impact brought about by our methods on implementing reuse, and on the maintenance and evolution of produced systems.

In this case study, we focus on one of the key components of ENFORMS, the *Local Control Broker* (LCB). Specifically, we study issues involving the reuse of the *Local Control Broker* throughout the design and development of four ENFORMS instantiations. The case study demonstrates:

- How reuse can be effectively integrated into software development based on the component-based software development (CBSD) framework developed in this research.
- The use of our compositional specification techniques in achieving CBSD.
- The use of our evaluation techniques in determining reusability as well as providing heuristics for composition design and component adaptation.
- The roles and features that a software engineering tool should have in supporting CBSD through the use of ABRIE in this case study.

The four instantiations studied are: standalone ENFORMS I (code named as SI), distributed ENFORMS I (code named as DI), standalone ENFORMS II (code named as SII), and distributed ENFORMS II (code named as DII). As far as the LCB component is concerned, the difference between a standalone and a distributed version is the support for distribution functionality, whereas the difference between an ENFORMS I and an ENFORMS II instantiation is that ENFORMS II supports richer data sets and more complex data manipulations, such as a Geographical Information System (GIS) utility.

## 9.3 Local Control Broker

In this section, we describe the four versions of the *Local Control Broker* (LCB). We specify them using our component specification language (see Chapter 5) that enables the formal analysis of these components.

As mentioned earlier, a LCB realizes the conceptual aggregation of a multimedia archive and its collection of items by managing items and manipulators. It provides search models with an interface to access archives. The major functionalities of a LCB include: interfacing with the item registry database to read and write all of the item descriptors; retrieving item descriptors from the item registry database; activating the manipulators applied to an item. The specifics of a LCB may vary from one instantiation to another. Figure 9.4 depicts the LCB component of standalone ENFORMS I.

In Figure 9.4, we specify both behavioral and structural properties of the SI LCB component. The *uses* section introduces traits that define basic sorts and operators used by the component specification. These traits are defined using the Larch Shared Language (LSL) and can be found in Appendix E. The specification described in Figure 9.4 is based on the object model of archives given in Figure 9.3. As we discussed in Chapters 4 and 5, our component interconnect model is designed to bridge the gap between high-level design and implementation, and intended to serve as a framework for integrating components. It is usually derived from the results of front-end system analysis and design methods, such as structural analysis and design techniques [122, 123] or object-oriented methods [9, 10]. As shown in Figure 9.4, the LCB component provides its environment three access functions, *initialize*, *retrieve* and *activate*, to the archives that are managed through an *ItemRegistry* database. In order to correctly function, the LCB component requires a file that contains item descriptors. Upon initialization, the LCB will upload all of these item descriptors into the *ItemRegistry* database. The LCB also requires a collection of manipulators. The SI version include manipulators for handling images, videos, audio, text, geographical information, as well as applications. As shown in Figure 9.4, these capabilities and requirements

```
# si.lcb.tex
# ABRIE specification for the local control broker component of
# ENFORMS I standalone version

COMP si_lcb : Module {

  uses itemreg(ItemRegistry for IR),
       lac(List for iList, List for sList),
       String(String for C),
       system,imview,app,grass,textdisp;

  PORT initialize : ProcDef {
       FUNC initialize(String& serverName) return int {
          requires ~isEmpty(servName);
          modifies database, result;
          ensures (if access(fullname(Registry_Filename))  /\
                      validRegistry(open(fullname(Registry_Filename)))
                   then database' = set_registry(database^, getItemDescriptor(
                        open(fullname(Registry_Filename)), servName)) /\ result' = 1
                   else result' = 0);
        }
    }
  PORT retrieve : ProcDef {
       FUNC retrieve(List& itemList) return List& { ... }
    }
  PORT activate : ProcDef {
       FUNC activate(String& manipHandle, String& itemKey,
                        String& target_IP_Address) return int {
          requires itemkey \in database.registry /\ (manipHandle = ''imagview''
                   \/ manipHandle = ''textdisplay'' \/ manipHandle = ''grass''
                   \/ manipHandle = ''audio'' \/ manipHandle = ''video''
                   \/ manipHandle = ''app'') /\ validIP(target_IP_Address);
          modifies result;
          ensures  ...
        }
    }
  PORT Registry_Filename : DataUse {
       DATA Registry_Filename : String;
    }
  PORT database : DataDef {
       DATA database : ItemRegistry;
    }
  PORT manipulators : RBundleUse {
       RBUNDLE manipulators {
           RBUNDLE imageview {
               FUNC init() {...};
               FUNC activate(...) {...};
            }
           RBUNDLE textdisply {...}
           RBUNDLE grass {...}
           RBUNDLE audio {...}
           RBUNDLE video {...}
           RBUNDLE app {...}
        }
    }
  CONSTR {
       DEPENDENCE {initialize : {database, Registry_FileName}}
       DEPENDENCE {retrieve : database}
       DEPENDENCE {activate : {database, manipulators}}
     }
}
```

Figure 9.4: Specification of SI LCB component

of the LCB component are encapsulated by different types of ports. Dependence relations between them are captured in the *CONSTR* section.

A distributed instantiation of the ENFORMS system has to handle network communications between archive servers and clients. Item descriptors retrieved from the archive server have to be packed as strings for transportation. On the other hand, once a transportation is completed, strings must be converted to its original form, a list of item descriptors. The LCB component of a distributed ENFORMS instantiation provides two functions to handle these tasks. Figure 9.5 gives the specification of a DI LCB component. The specification of a DI LCB is the same as that of a SI LCB except that the DI LCB has two ports, *retrieveAsString* and *convertString*, that provide the capability to deal with the communication requirements described previously. For the sake of brevity, from now on we omit behavioral specifications where they are not essential for the current discussion.

ENFORMS II evolved from ENFORMS I to meet new requirements. One of the main new requirements is to process more versatile data sets, which means that new kinds of manipulations are needed. Among the new types of data added to ENFORMS II are HTML documents and spatial data sets. Consequently, the LCB of ENFORMS II has to provide support for the activation of relevant manipulators. For HTML documents, the manipulator is HTML viewers, such as a web browser; for spatial data sets, it is boundary (arc) and point tools, such as the Arc/Info GIS system. Figure 9.6 describes the LCB component of a standalone ENFORMS II instantiation.

As shown in Figure 9.6, in addition to supporting more manipulations, the SII LCB component also provides additional ways to access the database. Ports *retrieveOne* and *retrieveAll* provide the capability for retrieving one specific item descriptor and retrieving all item descriptors, respectively.

```
# di.lcb.tex
# ABRIE specification for the local control broker component of
# ENFORMS I distributed version

COMP di_lcb : Module {

  PORT initialize : ProcDef {
       FUNC initialize(String& serverName) return int;
    }
  PORT retrieve : ProcDef {
       FUNC retrieve(List& itemlist) return List&;
    }
  PORT activate : ProcDef {
       FUNC activate(String& manipHandle, String& itemKey,
                     String& target_IP_Address) return int;
    }
  PORT Registry_Filename : DataUse {
       DATA Registry_Filename : String;
    }
  PORT database : DataDef {
       DATA database : ItemRegistry;
    }
  PORT manipulators : RBundleUse { ... }

  PORT retrieveAsString : ProcDef {
       FUNC retrieveAsString(List& itemlist) return String&;
    }
  PORT convertString : ProcDef {
       FUNC convertStringToIDList(String& s) return List&;
    }

  CONSTR {
       DEPENDENCE {initialize : {database, Registry_FileName}}
       DEPENDENCE {retrieve : database}
       DEPENDENCE {retrieveAsString : database}
       DEPENDENCE {activate : {database, manipulators}}
     }
}
```

Figure 9.5: Specification of DI LCB component

```
# sii.lcb.tex
# ABRIE specification for the local control broker component of
# ENFORMS II standalone version

COMP sii_lcb : Module {

  PORT initialize : ProcDef {
       FUNC initialize(String& serverName) return int;
    }
  PORT retrieve : ProcDef {
       FUNC retrieve(List& itemlist) return List&;
    }
  PORT activate : ProcDef {
       FUNC activate(String& manipHandle, String& itemKey,
                     String& target_IP_Address) return int;
    }
  PORT Registry_Filename : DataUse {
       DATA Registry_Filename : String;
    }
  PORT database : DataDef {
       DATA database : ItemRegistry;
    }
  PORT retrieveOne : ProcDef {
       FUNC retrieveOne(String& s) return ItemDescriptor&;
    }
  PORT retrieveAll : ProcDef {
       FUNC retrieveAll() return List&;
    }
  PORT manipulators : RBundleUse {
       RBUNDLE manipulators {
               RBUNDLE imageview {
                   FUNC init();
                   FUNC activate(...);
                 }
               RBUNDLE textdisply {...}
               RBUNDLE grass {...}
               RBUNDLE audio {...}
               RBUNDLE video {...}
               RBUNDLE app {...}
               RBUNDLE html {...}
               RBUMDLE arc {...}
               RBUNDLE pointtool {...}
          }
    }

  CONSTR {
       DEPENDENCE {initialize : {database, Registry_FileName}}
       DEPENDENCE {retrieve : database}
       DEPENDENCE {retrieveOne : database}
       DEPENDENCE {retrieveAll : database}
       DEPENDENCE {activate : {database, manipulators}}
    }
}
```

Figure 9.6: Specification of SII LCB component

Finally, the distributed instantiation of ENFORMS II contains the additional features of both DI and SII instantiations. Figure 9.7 depicts the LCB component of a distributed ENFORMS II instantiation.

## 9.4 Component-Based Development of Local Control Brokers

Given a target specification, the objective of reuse-oriented development is to develop an implementation that reuses existing components. Thus the main task is to develop a compositional specification equivalent to the target specification that enables the reuse of existing components. In this section, we first describe an iterative process for developing such a compositional specification. We then show how this process leads to a component-based development of the LCBs.

### 9.4.1 Reuse Oriented Decomposition

The key to developng a reuse-enabling composition for a target specification is to decompose the target specification into sub-specifications that are implementable by existing components. Figure 9.8 depicts an iterative process for reuse-oriented decomposition using Data Flow Diagram (DFD) like notations, where the circles represent activities, the parallel lines represent data stores, and the arrows represent flow of data.

As shown in Figure 9.8, a target specification is first evaluated to determine if it can be satisfied using existing components. Component evaluation techniques developed in Chapters 6 and 7 are used. When the result of this evaluation is negative, the target specification is decomposed into sub-specifications that are again evaluated. This iteration continues until either an existing implementation is found for a sub-specification or the

```
# dii.lcb.tex
# ABRIE specification for the local control broker component of
# ENFORMS II distributed version

COMP dii_lcb : Module {

  PORT initialize : ProcDef {
       FUNC initialize(String& serverName) return int;
    }
  PORT retrieve : ProcDef {
       FUNC retrieve(List& itemlist) return List&;
    }
  PORT activate : ProcDef {
       FUNC activate(String& manipHandle, String& itemKey,
                       String& target_IP_Address) return int;
    }
  PORT Registry_Filename : DataUse {
       DATA Registry_Filename : String;
    }
  PORT database : DataUse {
       DATA database : ItemRegistry;
    }
  PORT retrieveOne : ProcDef {
       FUNC retrieveOne(String& s) return ItemDescriptor&;
    }
  PORT retrieveAll : ProcDef {
       FUNC retrieveAll() return List&;
    }
  PORT retrieveAsString : ProcDef {
       FUNC retrieveAsString(List& itemlist) return String&;
    }
  PORT convertString : ProcDef {
       FUNC convertStringToIDList(String& s) return List&;
    }
  PORT manipulators : RBundleUse {
       RBUNDLE manipulators {
           RBUNDLE imageview {
               FUNC init();
               FUNC activate(...);
            }
           RBUNDLE textdisply {...}
           RBUNDLE grass {...}
           RBUNDLE audio {...}
           RBUNDLE video {...}
           RBUNDLE app {...}
           RBUNDLE html {...}
           RBUMDLE arc {...}
           RBUNDLE pointtool {...}
       }
    }
  CONSTR {
       DEPENDENCE {initialize : {database, Registry_FileName}}
       DEPENDENCE {retrieve : database}
       DEPENDENCE {retrieveOne : database}
       DEPENDENCE {retrieveAll : database}
       DEPENDENCE {retrieveAsString : database}
       DEPENDENCE {activate : {database, manipulators}}
    }
}
```

Figure 9.7: Specification of DII LCB component

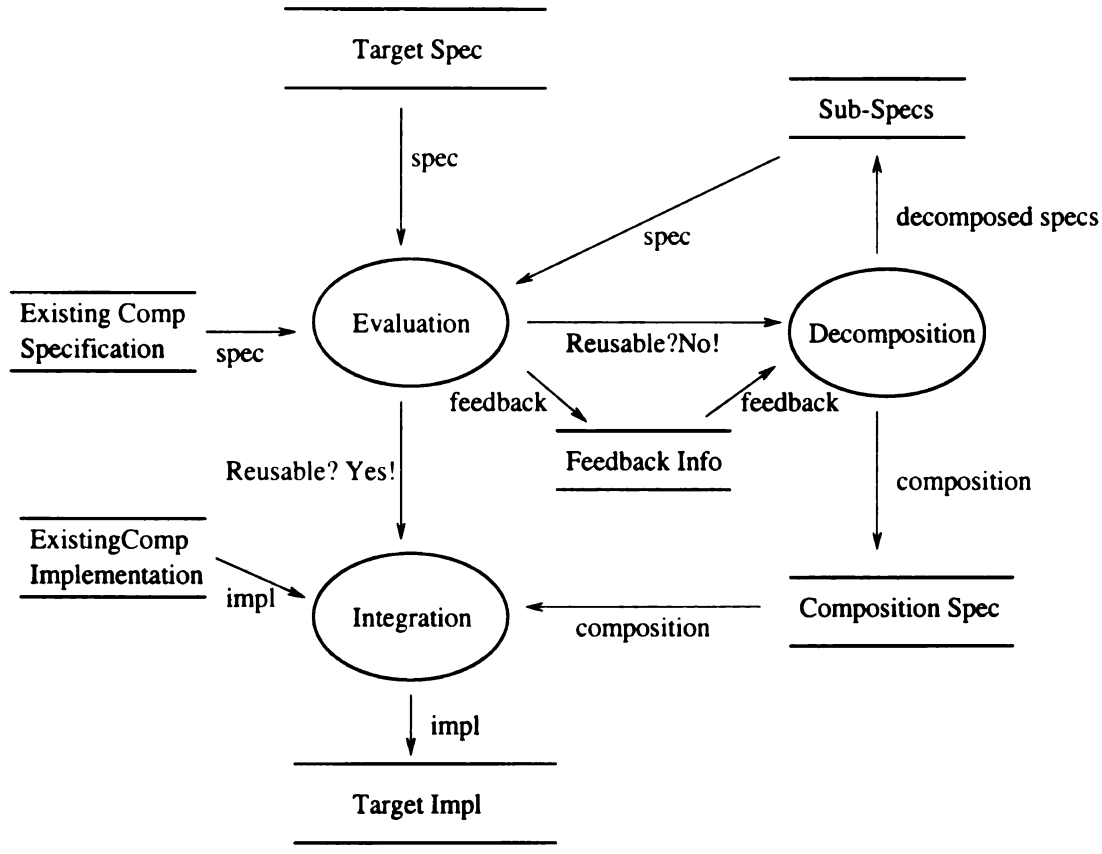



Figure 9.8: Reuse-oriented decomposition

sub-specification can not (or does not need to) be further decomposed. In the former case, an instance of reuse is achieved; in the latter case, implementation has to be developed from scratch. In general, a given specification can be decomposed in a variety of ways, few of them are suitable for achieving the reuse of existing components. This is the driving reason that feedback information derived from the evaluation process should be used to guide the decomposition. Example feedback information includes mismatches between a target (sub-)specification and an existing component specification, dependences between features in a specification, and so forth. The use of feedback information helps to identify features that will be bundled in a sub-specification that can be satisfied using a single existing component. Once sub-specifications are identified, a composition of them that is

equivalent to the original target specification can be created based on the relations between these sub-specifications. Such a composition is used in guiding the assembly of existing components to generate an implementation for the target specification.

### 9.4.2 Compositional Design of LCBs

In order to develop compositional specification for a LCB, we apply the decomposition process described in the previous subsection to LCB specifications.

**DI LCB**

We first consider the implementation of DI LCB assuming that the implementation of SI LCB is available for reuse. In order to evaluate the reusability of SI LCB for implementing DI LCB, we start with the dependence *Directed Acyclic Graphs* (DAGs) of the two component interfaces (see Chapter 7). Figure 9.9 (a) and (b) show the dependence DAG of SI LCB and DI LCB, respectively. In Figure 9.9, each node represents a port, the dashed circle (node) denotes a resource to be required, whereas the solid circle (node) means a resource to be provided. The directed edge $\langle \overrightarrow{u, v} \rangle$ means that $u$ depends on $v$ ($v$ is required for providing $u$).

Since there are capabilities that are provided by DI LCB but not by SI LCB, according to the *interface generality relation* definition (see Definition 19, Chapter 7), it is obvious that SI LCB can not be directly used for implementing DI LCB. However, observing that in Figure 9.9, (a) is a subgraph of (b), we can decompose DI LCB based on this information so that one sub-component of DI LCB can be implemented by SI LCB. Figure 9.10 depicts this decomposition using the dependence DAGs of the two sub-components.

As shown in Figure 9.10, we partition DI LCB into two components: *basic* and *distri*. The *basic* component encapsulates the basic features of a LCB that is also shared by SI

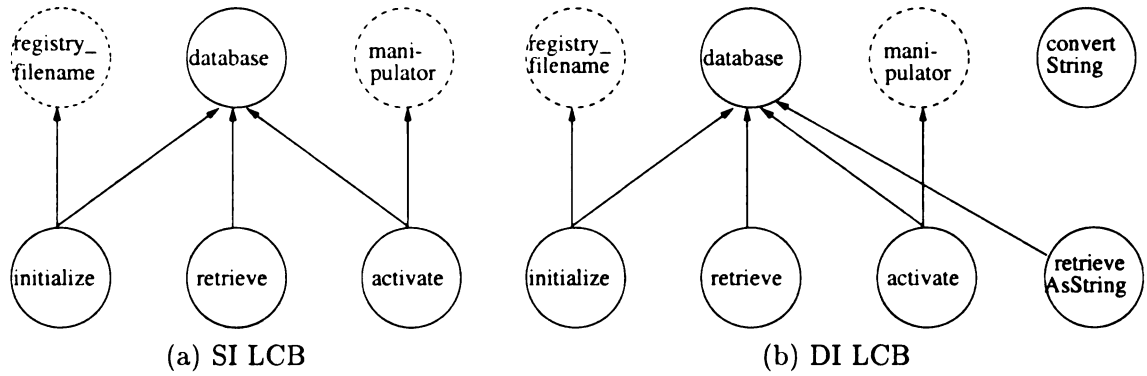



Figure 9.9: Dependence DAG of SI LCB and DI LCB

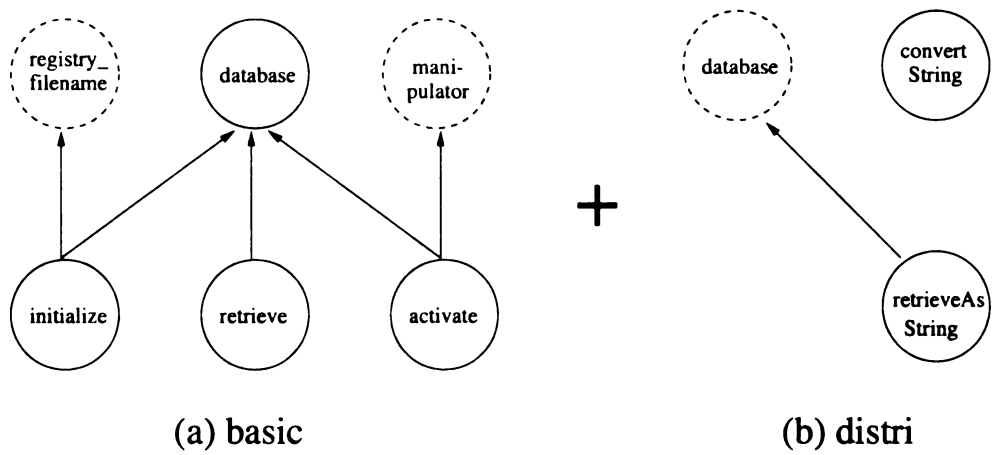



Figure 9.10: Decomposition of DI LCB

LCB. The *distri* component captures the features specific for a distributed ENFORMS instantiation. It should be noted that the *database* node in the *distri* component is dashed, meaning that this resource is required from the other component. Figure 9.11 gives the compositional specification of DI LCB.

```
# di.composite.tex
# ABRIE compositional specification for the local control broker
# component of ENFORMS I distributed version

SYSTEM di_lcb : Module {

   COMP basic : Module {
       PORT initialize : ProcDef {
             FUNC initialize(String& serverName) return int;
           }
       PORT retrieve : ProcDef {
             FUNC retrieve(List& itemlist) return List&;
           }
       PORT activate : ProcDef {
             FUNC activate(String& manipHandle, String& itemKey,
                        String& target_IP_Address) return int;
           }
       PORT Registry_Filename : DataUse {
             DATA Registry_Filename : String;
           }
       PORT database : DataDef {
             DATA database : ItemRegistry;
           }
       PORT manipulators : RBundleUse { ... }
       CONSTR {
             DEPENDENCE {initialize : {database, Registry_FileName}}
             DEPENDENCE {retrieve : database}
             DEPENDENCE {activate : {database, manipulators}}
           }
     }

   COMP distri : Module {
       PORT retrieveAsString : ProcDef {
             FUNC retrieveAsString(List& itemlist) return String&;
          }
       PORT convertString : ProcDef {
             FUNC convertStringToIDList(String& s) return List&;
          }
       PORT database : DataUse {
             DATA database : ItemRegistry;
          }
       CONSTR {
             DEPENDENCE {retrieveAsString : database}
          }
     }

   CONN ud : AccessData

   CONFIGURE {
       basic.database TO ud.Definer;
       distri.database TO ud.User;
    }
}
```

Figure 9.11: Compositional specification of DI LCB

Given such a decomposition, it is not difficult to show that the *basic* component is implemented by the SI LCB implementation. The *distri* component can be constructed from scratch. The complete implementation of the DI LCB thus can be generated based on the compositional specification.

**SII LCB**

We next consider the design and implementation of SII LCB. Similar to the design of DI LCB, we consider the reuse of SI LCB in implementing SII LCB. As with DI LCB, we first analyze the specifications of the two components, and conclude that SI LCB cannot be used to fully fulfill SII LCB. However, we find that part of SII LCB may be implemented by SI LCB. Based on this feedback information, we thus decompose SII LCB into two components in a similar fashion to that used to decompose DI LCB. Figure 9.12 depicts this decomposition.

We then proceed to verify our assumption that the *basic* component can be implemented by SI LCB. Based on the the definition of the *interface generality relation* (Definition 19), we then match the ports of SI LCB to those of SII LCB and proof obligations can be generated for each match. Figure 9.13 shows the proof obligation for matching the *activate* port of SI LCB to the *activate* port of SII LCB using the *relaxed-plug-in* match.

As previously pointed out in Chapters 7 and 8, the reusability of a component is established only if all the relevant proof obligations are resolved. The Larch Prover (LP) is integrated into ABRIE and can be conveniently invoked for assisting the proving process. Unfortunately, this specific proof obligation cannot be resolved, which indicates that the *activate* port of SI LCB may not be used to fulfill the *activate* port of SII LCB. Further analysis reveals that the *activate* port of SII LCB must provide capabilities for handling additional manipulations other than that handled by the *activate* port of SI LCB. This

```
# sii.composite.tex
# ABRIE compositional specification for the local control broker component of
# ENFORMS II standalone version

SYSTEM sii_lcb : Module {

  COMP basic : Module {
      PORT initialize : ProcDef {
           FUNC initialize(String& serverName) return int;
         }
      PORT retrieve : ProcDef {
           FUNC retrieve(List& itemlist) return List&;
         }
      PORT activate : ProcDef {
           FUNC activate(String& manipHandle, String& itemKey,
                       String& target_IP_Address) return int;
         }
      PORT Registry_Filename : DataUse {
           DATA Registry_Filename : String;
         }
      PORT database : DataDef {
           DATA database : ItemRegistry;
         }
      PORT manipulators : RBundleUse {
           RBUNDLE manipulators { ... }
         }

      CONSTR {
             DEPENDENCE {initialize : {database, Registry_FileName}}
             DEPENDENCE {retrieve : database}
             DEPENDENCE {activate : {database, manipulators}}
         }
     }

  COMP verii : Module {
      PORT retrieveOne : ProcDef {
           FUNC retrieveOne(String& s) return ItemDescriptor&;
         }
      PORT retrieveAll : ProcDef {
           FUNC retrieveAll() return List&;
         }
      PORT database : DataUse {
           DATA database : ItemRegistry;
         }
      CONSTR {
           DEPENDENCE {retrieveOne : database}
           DEPENDENCE {retrieveAll : database}
         }
     }
   CONN ud : AccessData

   CONFIGURE {
       basic.database TO ud.Definer;
       verii.database TO ud.User;
    }
}
```

Figure 9.12: Compositional specification of SII LCB

```
%% Proof obligation for matching the activate port of si_lcb to
%% the activate port of sii_lcb
si_lcb_sii_lcb: trait
  includes itemreg(ItemRegistry for IR),
           lac(List for iList, List for sList),
           String(String for C)
  implies
    \forall manipHandle, itemKey, target_IP_Address: String,
            result_post : int, database : ItemRegistry
    ((itemkey \in database.registry /\ (manipHandle = ''imagview''
      \/ manipHandle = ''textdisplay'' \/ manipHandle = ''grass''
      \/ manipHandle = ''audio'' \/ manipHandle = ''video''
      \/ manipHandle = ''app'' \/ manipHandle = ''html''
      \/ manipHandle = ''arc'' \/ manipHandle = ''pointtool'')
      /\ validIP(target_IP_Address))
     =>
     (itemkey \in database.registry /\ (manipHandle = ''imagview''
      \/ manipHandle = ''textdisplay'' \/ manipHandle = ''grass''
      \/ manipHandle = ''audio'' \/ manipHandle = ''video''
      \/ manipHandle = ''app'')
      /\ validIP(target_IP_Address)))
 /\ ((itemkey \in database.registry /\ (manipHandle = ''imagview''
      \/ manipHandle = ''textdisplay'' \/ manipHandle = ''grass''
      \/ manipHandle = ''audio'' \/ manipHandle = ''video''
      \/ manipHandle = ''app'' \/ manipHandle = ''html''
      \/ manipHandle = ''arc'' \/ manipHandle = ''pointtool'')
      /\ validIP(target_IP_Address))
      /\ (if activated(value(lookup(manipHandle,value(lookup(
         itemKey, database.registry)).theManipulators)),target_IP_Address)
         then result_post = 1 else result_post = 0)
     => (if activated(value(lookup(manipHandle,value(lookup(
         itemKey, database.registry)).theManipulators)),target_IP_Address)
         then result_post = 1 else result_post = 0))
```

Figure 9.13: Proof obligations for matching *activate* ports of SI LCB and SII LCB

feedback information leads us to refine the original decomposition of SII LCB. The idea is to split the *activate* port into two ports: one for manipulation capabilities shared by both ENFORMS I and ENFORMS II instantiations, the other encapsulates capabilities specific to ENFORMS II. Figure 9.14 depicts the refined decomposition.

It has been established previously that the implementation of SI LCB also implements the *basic* component. As with DI LCB, we can implement the version II specific features encapsulated by component *verii* from scratch, and then based on the compositional specification, generate the complete implementation of SII LCB.

```
# sii.composite.2.tex
# ABRIE compositional specification for the local control broker component of
# ENFORMS II standalone version

SYSTEM sii_lcb : Module {

  COMP basic : Module {
      PORT initialize : ProcDef { ... }
      PORT retrieve : ProcDef { ... }
      PORT activate : ProcDef { ... }
      PORT Registry_Filename : DataUse { ... }
      PORT database : DataDef { ... }
      PORT manipulators : RBundleUse {
           RBUNDLE manipulators {
               RBUNDLE imageview {
                   FUNC init();
                   FUNC activate(...);
                 }
               RBUNDLE textdisply {...}
               RBUNDLE grass {...}
               RBUNDLE audio {...}
               RBUNDLE video {...}
               RBUNDLE app {...}
            }
         }

      CONSTR {
             DEPENDENCE {initialize : {database, Registry_FileName}}
             DEPENDENCE {retrieve : database}
             DEPENDENCE {activate : {database, manipulators}}
         }
     }

  COMP verii : Module {
      PORT retrieveOne : ProcDef { ... }
      PORT retrieveAll : ProcDef { ... }
      PORT database : DataUse { ... }
      PORT activate : ProcDef { ... }
      PORT manipulators : RBundleUse {
           RBUNDLE manipulators {
               RBUNDLE html {...}
               RBUMDLE arc {...}
               RBUNDLE pointtool {...}
            }
         }
      CONSTR {
           DEPENDENCE {retrieveOne : database}
           DEPENDENCE {retrieveAll : database}
           DEPENDENCE {activate : {database, manipulators}}
         }
     }
   CONN ud : AccessData

   CONFIGURE {
       basic.database TO ud.Definer;
       verii.database TO ud.User;
    }

  }
```

Figure 9.14: Refined compositional specification of SII LCB

**DII LCB**

Finally, we consider the design and implementation of DII LCB. A similar process can be applied to the reuse analysis and decomposition of DII LCB. However, this time we have three components that are available for reuse: SI LCB, *distri*, and *versii*. When we search for the implementation of DII LCB with reuse as a main goal, we can obtain a decomposition shown in Figure 9.15.

Obviously, the implementation of the aforementioned three existing components can be reused to implement the three sub-components in the compositional specification of DII LCB, respectively.

### 9.4.3 Discussion

We showed how reuse can be effectively integrated into software design and implementation. As we have discussed and stated in Chapter 1, an important position of this research is that, *efficient reuse can not be achieved unless it is integrated into the early phases of the software development lifecycle.* The later reuse is considered, the more design decisions have been made, and the more constraints exist for restricting the reusability of a component. In this case study, we integrate reuse analysis into the design phase of the LCBs, and thus discover (or in a sense, create) some opportunities for reusing existing components that otherwise would be ignored in an *ad hoc*, code scavenging approach. While the reuse-oriented design analysis approach has evident improvement in implementing reuse, the process described in Figure 9.8 is inadequate in terms of implementing reuse. The main problem is that reuse is constrained by the availability of appropriate components and hinges upon the somewhat accidental matching between available components and the target requirements. In other words, reuse is not systematically planned. In Chapter 4, we proposed a domain

```
# dii.composite.tex
# ABRIE compositional specification for the local control broker component of
# ENFORMS II distributed version

SYSTEM dii_lcb : Module {

  COMP basic : Module {
      PORT initialize : ProcDef { ... }
      PORT retrieve : ProcDef { ... }
      PORT activate : ProcDef { ... }
      PORT Registry_Filename : DataUse { ... }
      PORT database : DataDef { ... }
      PORT manipulators : RBundleUse { ... }

      CONSTR {
             DEPENDENCE {initialize : {database, Registry_FileName}}
             DEPENDENCE {retrieve : database}
             DEPENDENCE {activate : {database, manipulators}}
         }
     }

  COMP distri : Module {
       PORT retrieveAsString : ProcDef { ... }
       PORT convertString : ProcDef { ... }
       PORT database DataUse { ... }
       CONSTR {
             DEPENDENCE {retrieveAsString : database}
          }
     }

  COMP verii : Module {
      PORT retrieveOne : ProcDef { ... }
      PORT retrieveAll : ProcDef { ... }
      PORT database : DataUse { ... }
      PORT activate : ProcDef { ... }
      PORT manipulators : RBundleUse {
           RBUNDLE manipulators {
               RBUNDLE html {...}
               RBUMDLE arc {...}
               RBUNDLE pointtool {...}
            }
         }
      CONSTR {
           DEPENDENCE {retrieveOne : database}
           DEPENDENCE {retrieveAll : database}
           DEPENDENCE {activate : {database, manipulators}}
         }
     }
   CONN ud1, ud2 : AccessData

   CONFIGURE {
       basic.database TO ud1.Definer;
       verii.database TO ud1.User;
       basic.database TO ud2.Definer;
       distri.database TO ud2.User;
    }
  }
```

Figure 9.15: Compositional specification of DII LCB

specific approach to solve this problem. That is, as early as the system analysis phase, domain analysis is conducted to identify the commonalities across various products within one domain, as well as the variants among these products. Domain engineering is followed to develop domain-specific architectures that are reusable for systems within the domain. Domain-specific architectures also provide guidelines or frameworks that facilitate or enable the reuse of domain implementations. In such an approach, effective reuse is achieved by careful planning. In the case of the LCB component, instead of following an iterative process to come up with a decomposition for a specific instantiation, we can first conduct the analysis of the LCB domain: identify various potential instantiations of ENFORMS, their common requirements, and their differences. For example, we may identify that the requirements of SI LCB are shared by a number of instantiations. We can also identify features shared by all distributed instantiations, as well as plan the evolution of ENFORMS to embody more types of data and manipulations. Such an analysis will lead us to a generic architecture design for the LCB component that can be customized for specific instantiations. Components encapsulating specific features can also be in better shape for reuse, since by design they are part of the generic architecture, and are ready for integration into the architecture.

## 9.5 Summary

One tangible result of the decomposition process as described in Section 9.4.3 is the compositional specification of a LCB. In addition to identify the constituent components, this specification specifies how these (sub-) components are composed to satisfy the original specification. This compositional specification bridges the gap between high-level design and implementation, and provides design to implementation traceability necessary for efficient system maintenance and evolution. Another merit with our approach lies in the

fact that specific features are identified and encapsulated in reusable components. The implementations for various instantiations of LCB are composed from these components. Therefore, each instantiation of the LCB component is no longer a monolithic, closed program, instead they share common primitive components. Since the implementation of a LCB is composed from components, the composition design as a design dimension provides extra flexibility and alternatives in developing a solution. It also enables design maintenance and facilitates evolution. Maintenance and evolution can be conducted at the composition level. For example, we can change the way that components are composed to accommodate new demands. On the other hand, given a compositional design, when a specific component is modified or replaced, it is not difficult to derive its effects on the overall system.

Throughout this case study, we have shown how reuse can be effectively integrated into software development. However, we should emphasize the importance of the techniques developed in this research in facilitating the integration. As we have seen throughout this case study, the compositional specification technique plays a central role in achieving component-based design. The evaluation technique that integrates both formal (semantic-based) and semi-formal (keyword- and signature-based) methods not only facilitates the determination of reusability, but also provides feedback to component adaptation as well as composition design.

# Chapter 10

# Conclusions and Future Investigations

Achieving reuse in software development has been a much sought after goal. Software reuse involves the creation, classification, retrieval, composition and integration of software components. While there exists a great deal of successful reuse experience, software reuse, in general, has not yet fulfilled its promise to significantly improve software development productivity and software quality [25]. After an extensive literature review and case analysis, we identified several major obstacles to effective reuse. A key step in reuse is to locate the most appropriate components that satisfy a given query requirement. Unfortunately, the criteria necessary for a component to satisfy a query requirement is usually implicit and not precisely captured as in the case of *keyword* or *signature* based retrieval algorithms. Specification matching has been proposed as one type of evaluation criteria for reuse. While formal methods have the advantage of being precise and amenable to automation, there does not exist a general approach to reason about the usefulness of a specification match for determining reusability. Another problem with current specification-based evaluation

methods is that they are only applicable to functions or modules, rather than architectural components.

Another obstacle to effective reuse is due to the semantic gap between a query requirement and available components. Even for a requirement of moderate complexity, it is seldom the case that an existing component can exactly implement it. Instead, an appropriate composition of a set of components is usually needed to satisfy the requirement. This raises the need for a specification language to represent composition. Ideally, a composition specified in this language may serve as a framework into which existing components can be evaluated and integrated. This means that the specification language should specify both structural and functional aspects of a target system.

Lastly, but not the least important, the lack of a seamless integration of reuse techniques imposes significant barriers to achieving effective reuse.

## 10.1 Summary of Contributions

Based on the premise that effective reuse can be achieved only when reuse issues are considered throughout the software development life cycle and are addressed on the basis of a formal foundation, the overall objective of this research is to develop an architecture-based component reuse framework. This framework should address software reuse issues in an integrated fashion, and be amenable to automation for such tasks as component evaluation, adaptation, and integration. Towards this end, the research described in this dissertation has made the following contributions.

- **Proposed an integrated framework for component reuse**

  A software architecture-based component reuse framework is proposed. This framework defines a process to component-based software engineering. In particular, we em-

phasize the role of software architectures as compositions of components that enables component assembly. This framework provides an integrated approach to address issues involving component composition, evaluation, and integration, while serving as a technical map for the reminder of this research.

- **Developed a component interconnect model and specification language**

  Component-based engineering emphasizes the composition of components, that is, the structural properties of a system in terms of its constituent components. On the other hand, in order to enable the effective retrieval and evaluation of existing components for reuse, a certain degree of preciseness in specifying requirements and/or component functionalities is required. In this research, we developed a flexible specification framework that integrates software architecture description techniques with traditional formal specification languages.

- **Established a semantic foundation for specification matching**

  While considering formal methods-based specification matching as an evaluation method for reuse has merits of being precise and amenable to automation, it suffers from the lack of a general approach to reason about the usefulness of a specification match for determining reusability. In this research, a semantic foundation is established to reason about the connections between a specification match and its usefulness for determining reusability. Based on this semantic foundation, we proved the existence of the *best* reuse-ensuring matches, and showed exactly what they are. As a result, we provided a formal foundation for applying specification matching-based methods to component evaluation, and simplified the development of the *best* component evaluation method. The influence of this result is not limited to component

reuse. It also applies to any area that involves specification matching. For example, in defining object-oriented subtyping [97, 98, 99], this result provides a basis for selecting specification matching criterion to determine behavioral relationships between methods of two classes.

- **Extended specification-based evaluation methods to architectural components**

  Specification matching has been applied to evaluate fine-grained components, i.e., functions or modules. In order to scale up reuse, architectural components need to be considered. In this research, a formal specification-based method has been developed to evaluate the reusability of architectural components. Specifically, a *generality* relation between component interfaces is developed to capture the reusability of an architectural component for satisfying a query interface. The uniqueness in the definition is its consideration of the dependencies among features of a component. Because these dependencies are architectural features, rather than functionalities, this interface *generality* relation embodies not only functional requirements, but also structural requirements, for evaluating the reusability of a component.

- **Developed a reusable toolkit for facilitating the development of applications involving software architectures and component reuse**

  As one means for validating and facilitating this research, we developed an architecture-based component reuse and integration environment (ABRIE). Through its graphical environment, ABRIE supports composition manipulation, reusability analysis and component integration. However, ABRIE has broader applicability due to its layered architecture design. The object-oriented models of architectural el-

ements that constitute the bottom layer of ABRIE provide a reusable toolset for creating software architecture and component reuse related tools.

## 10.2 Impact of Research and Future Investigations

Component-based software engineering is an emerging area of active research and practice. It promises to radically reduce the time and cost of software development through systematic reuse. However, in order to enable component-based software engineering, the engineering process needs to be changed from conventional product-oriented, waterfall model to a domain-oriented, architecture based, component assembly model. The integrated framework for component reuse proposed in this research provides a roadmap for implementing component-based software engineering. Techniques developed in this research enable the implementation of this component-based software engineering paradigm. In particular, by integrating formal methods into this paradigm, we explore and demonstrate the feasibility of automating component-based software development. The impact of some of the results obtained in this research may go beyond the domain of software reuse or component-based software engineering. For example, the semantic foundation for specification matching developed in this research provides a generic framework to reason about specification matches, which, in addition to determining reuse, have been used in many other software engineering activities, such as maintenance, reengineering [82], and object-oriented subtyping [97, 98, 99].

Several areas of research appear promising for future investigations. Each is briefly described.

### 10.2.1 Domain specific support

As pointed out in Chapter 4, domain engineering [83, 84, 85, 86] is a design-for-reuse process that produces domain-specific reusable artifacts, including domain requirement models (specifications), architectures, and implementations. In application engineering, these domain-specific assets are reused to construct specific applications within the domain. In this research, we leave the possibility open to support domain-specific development. Specifically, in the design of the component interconnect model and the prototype system, ABRIE, we provide meta-mechanisms for specifying architectural knowledge such as styles that may be domain specific. Therefore, by customizing the specification language and the ABRIE environment using domain specific architectural knowledge and assets, we should be able to obtain domain-specific design environments. It would be interesting to apply the results of this research to a particular domain, and explore how domain orientation can facilitate the evaluation, composition, and integration of software components.

### 10.2.2 Composition design

One important aspect of our component-based software development paradigm is composition (or architecture) design. One intention of introducing composition (or architecture) design is to bridge the gap between detailed implementation and analysis models or high-level designs that are generally developed using specific "front-end" analysis or design methodology, such as object-oriented methods [9, 10, 119] or structured techniques [122, 123]. Although we have presented a scheme for specifying composition, it is not clear how a reuse-enabling component composition can be derived from relevant analysis models or high-level designs. In the case study described in Chapter 9, we used an intuitive, iterative

decomposition process to derive a composition design. Obviously, a more formal process is needed.

Another promising line of work in supporting composition-based design is to enrich the set of design abstractions. For example, it would be ideal to support style-specific architecture design by providing a set of common abstractions specific to this style. Another approach is to have domain-specific abstractions for the design of applications within a domain, as we discussed earlier. In fact, those style-specific or domain-specific abstractions of design elements have been accruing in recent years [124, 125, 126] . Enabling the integration and use of those abstractions into our framework will facilitate composition design through the potential reuse of those design abstractions.

### 10.2.3 Packaging adaptation

The evaluation method described in Chapter 7 is sufficient for finding the conforming components to a given interface. However, it is typical that available components may not exactly conform to an interface. Instead, certain mismatches may exist. There are generally three kinds of mismatches. The simplest one is *syntactical mismatch*, such as different orderings of parameters to functions, and naming conflicts. Another kind of mismatch is due to the semantic difference, that is, a component may simply deliver a different behavior than that specified by the interface. The third kind of mismatch is due to the packaging difference, that is, a component delivers the same capability as those specified by the interface, but in a different way. For example, a sorting component packaged as a filter cannot be used as a procedure.

In the component evaluation method proposed in Chapter 7, we emphasize the use of semantics, thus allowing syntactic mismatches to be accommodated. In fact, they are

identified and recorded in the component evaluation process and used later in the packaging process to generate wrappers for adaptation purposes. The evaluation method will discard those components that semantically do not match with the requirement. Although it is possible to adapt those components, we believe that the computational cost for doing so is prohibitively high due to the nature of the problem [127, 128]. On the other hand, many existing components, particularly those coarse-grained architectural components, cannot be reused solely due to the constraints imposed by their packaging style [31]. An interesting avenue of further research would be to develop methods to resolving packaging mismatches. The objective of packaging adaptation is to convert one packaging style to another while keeping the capabilities of a component unchanged. For example, most existing libraries use basic packaging styles such as procedure and class, but when applied to an architectural context, they need to be packaged in an appropriate form consistent with the architecture. It would also be interesting to establish a classification of various packaging styles and develop templates for converting among these styles. These templates may provide a basis for automating packaging adaptation.

# Bibliography

# Bibliography


[1] J. M. Wing, "A specifier's introduction to formal methods," *IEEE Computer*, vol. 23, September 1990.

[2] C. B. Jones, *Systematic Software Development using VDM.* Prentice Hall International, 1990.

[3] J. B. Wordsworth, *Software Development with Z.* Addison-Wesley Longman Ltd., 1992.

[4] B. H. C. Cheng, "Applying formal methods in automated software development," *Journal of Computer and Software Engineering*, vol. 2, no. 2, 1994.

[5] E. M. Clarke and J. M. Wing, "Formal methods: State of the art and future directions," *ACM Computing Surveys*, vol. 28, December 1996.

[6] D. R. Smith, "Kids: A semiautomatic program development system," *IEEE Transaction on Software Engineering*, vol. 16, September 1990.

[7] H. A. Partsch, *Specification and Transformation of Programs.* Springer-Verlag, 1990.

[8] M. R. Lowry and R. D. McCartney, *Automating Software Design.* AAAI Press/MIT Press, 1991.

[9] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Objected-Oriented Modeling and Design.* Prentice Hall, 1991.

[10] G. Booch, *Object-Oriented Analysis and Design.* Addison-Wesley, 2 ed., 1994.

[11] F. P. Brooks, "No silver bullet – essence and accidents of software engineering," *IEEE Computer*, vol. 20, April 1987.

[12] N. G. Leveson and C. S. Turner, "An investigation of the therac-25 accidents," *IEEE Computer*, vol. 26, July 1993.

[13] W. W. Gibbs, "Software's chronic crisis," *Scientific American*, September 1994.

[14] S. Flowers, *Software Failure: Management Failure: Amazing Stories and Cautionary Tales.* Wiley, 1996.

[15] P. G. Neumann, "The risk digest forum." WWW site, http://catless.ncl.ac.uk/Risks/.

[16] T. J. Biggerstaff and A. J. Perlis, *Software Reusability*, vol. 1,2. ACM Press, 1989.

[17] C. W. Krueger, "Software reuse," *ACM Computing Surveys*, vol. 24, June 1992.

[18] H. Mili, F. Mili, and A. Mili, "Reusing software: Issues and research directions," *IEEE Transactions on Software Engineering*, vol. 21, pp. 528–561, June 1995.

[19] M. D. McIlroy, "Mass produced software components," in *Software Engineering: Report on a conference by the NATO Science Committee* (P. Naur and B. Randell, eds.), pp. 138–150, October 1968.

[20] G. Booch, *Software Components with Ada.* Benjamin/Cummings, 1987.

[21] A. Stepanov and M. Lee, "The standard template library." ANSI/ISO document, October 1995.

[22] D. B. Musser, A. Saini, and A. Stepanov, *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library.* Addison-Wesley, 1996.

[23] Web site, http://www.asset.com. A large collection of software libraries are available for download.

[24] G. Shepherd and S. Wingo, *MFC Internals.* Addison-Wesley, 1996.

[25] R. Prieto-Diaz, "Status report: Software reusability," *IEEE Software*, May 1993.

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

[27] J.-J. Jeng and B. H.C.Cheng, "Using formal methods to construct a software component library," in *Lecture Notes in Computer Science*, vol. 717, pp. 397–417, September 1993.

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

[29] R. Mili, A. Mili, and R. T. Mittermeir, "Storing and retrieving software components: A refinement based system," *IEEE Transactions on Software Engineering*, vol. 23, July 1997.

[30] A. M. Zaremski and J. M. Wing, "Specification matching of software components," in *Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, October 1995.

[31] D. Garlan, R. Allen, and J. Ockerbloom, "Architectural mismatch: Why reuse is so hard," *IEEE Software*, vol. 12, November 1995.

[32] M. Shaw and D. Garlan, *Software Architectures: Perspectives on an Emerging Discipline.* Prentice Hall, 1996.

[33] Y. Chen and B. Cheng, "Facilitating an automated approach to architecture-based software reuse," in *Proceedings of the 12th IEEE International Conference on Automated Software Engineering*, November 1997.

[34] Y. Chen and B. H. C. Cheng, "Formally specifying and analyzing architectural and functional properties of components for reuse," in *WISR8*, 1997.

[35] P. H. J. van Eijk, C. A. Vissers, and M. D. (editors), *The Formal Description Technique LOTOS.* Elsevier Science Publishers B.V., 1989.

[36] 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.

[37] S. J. Garland and J. V. Guttag, *LP, the Larch Prover: User and Reference Manual.* MIT, December 1994.

[38] D. Garlan and M. Shaw, "An introduction to software architecture," in *Advances in Software Engineering and Knowledge Engineering* (V. Ambriola and G. Tortora, eds.), vol. I, World Scientific Publishing Company, 1993.

[39] D. Garlan and D. Perry, "Introduction to the special issue on software architecture," *IEEE Transaction on Software Engineering*, vol. 21, April 1995.

[40] R. S. Pressman, *Software Engineering.* McGraw-Hill, Inc., 3 ed., 1992.

[41] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *ACM SIGSOFT Software Engineering Notes*, vol. 17, October 1992.

[42] R. Allen and D. Garlan, "Formalizing architectural connection," in *Proc. 16th International Conference on Software Engineering*, (Sorrento, Italy), May 1994.

[43] R. Allen and D. Garlan, "A formal basis for architectural connection," *ACM Transactions on Software Engineering and Methodology*, July 1997.

[44] M. Moriconi and X. Qian, "Correctness and composition of software architectures," in *Proc. ACM SIGSOFT'94*, December 1994.

[45] D. Garlan and D. E. Perry, eds., *Special Issue on Software Architecture, IEEE Transaction on Software Engineering*, vol. 21, IEEE Computer Society, April 1995.

[46] D. C. Luckham and et al., "Specification and analysis of system architecture using rapide," *IEEE Transactions on Software Engineering*, vol. 11, April 1995.

[47] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, "Specifying distributed software architectures," in *Proceedings of the Fifth European Software Engineering Conference (ESEC'95)*, (Barcelona), September 1995.

[48] D. Garlan, "What is style," in *Proceedings of Dagshtul Workshop on Software Architecture*, February 1995.

[49] M. Shaw, R. DeLine, and G. Zelenik, "Abstractions and implementations for architectural connections," in *Proc. 3rd Intl. Conf. on Configurable Distributed Systems*, 1996.

[50] M. Shaw, R. DeLine, D. Klein, T. Ross, D. Young, and G. Zelesnik, "Abstractions for software architecture and tools to support them," *IEEE Transactions on Software Engineering*, vol. 21, April 1995.

[51] G. Glass, *Unix for Programmers and Users: A Complete Guide.* Prentice Hall, 1993.

[52] J. Bloomer, *Power Programming with RPC.* O'Reilly & Associates, Inc., 1991.

[53] R. T. Monroe and D. Garlan, "Style-based reuse for software architectures," in *Proceedings of 4th International Conference on Software Reuse*, April 1996.

[54] D. Garlan, R. T. Monroe, and D. Wile, "Acme: An architecture description interchange language," in *Proceedings of CASCON'97*, November 1997.

[55] M. Moriconi, X. Qian, and R. A. Riemenschneider, "Correct architecture refinement," *IEEE Transactions on Software Engineering*, vol. 21, pp. 356–372, April 1995.

[56] M. Moriconi and R. A. Riemenschneider, "Introduction to sadl 1.0: A language for specifying software architecture hierarchies," Tech. Rep. SRI-CSL-97-01, Computer Science Laboratory, SRI International, March 1997.

[57] D. Garlan, R. Allen, and J. Ockerbloom, "Exploiting style in architectural design environments," in *Proceedings of SIGSOFT '94 Symposium on the Foundations of Software Engineering*, December 1994.

[58] R. T. Monroe, "Capturing design expertise in customized software architecture design environments," in *Proceedings of the Second International Software Architecture Workshop*, October 1996.

[59] J.-J. Jeng and B. H.C.Cheng, "A formal approach to reusing more general components," in *The Proceedings of IEEE 9th Knowledge-Based Software Engineering Conference*, September 1994.

[60] J.-J. Jeng and B. H.C.Cheng, "Using analogy and formal methods for software reuse," in *The Proceedings of IEEE 5th International Conference on Tools with AI*, pp. 113–116, November 1993.

[61] B. H. C. Cheng and J.-J. Jeng, "Reusing analogous components," *IEEE Transaction on Knowledge and Data Engineering*, vol. 9, pp. 341–349, March/April 1997.

[62] A. M. Zaramski and J. M. Wing, "Signature matching: A key to reuse," in *Proceedings of the ACM SIGSOFT'93 Symposium on the Foundations of Software Engineering*, December 1993.

[63] B. Fischer, M. Kievernagel, and G. Snelting, "Deduction based software component retrieval," in *Proceedings of IJCAI Workshop on Reuse of Proofs, Plans and Programs*, (Montreal, Quebec, Canada), June 1995.

[64] F. DeRemer and H. Kron, "Programming-in-the-large versus programming-in-the-small," *IEEE Transactions on Software Engineering*, June 1976.

[65] R. Prieto-Diaz and J. M. Neighbors, "Module interconnection languages," *J. Systems and Software*, vol. 6, Nov 1986.

[66] W. Tracz, "Lileanna: A parameterized programming language," in *Proc. of Second International Workshop on Software Reuse*, 3 1993.

[67] B. Boehm and B. Scherlis, "Megaprogramming," in *Proceedings of the DARPA Software Technology Conference*, (Arlington, VA), 1992.

[68] G. Wiederhold, P. Wegner, and S. Ceri, "Towards megaprogramming: A paradigm for component-based programming," *Communications of the ACM*, June 1992.

[69] W. Kozaczynski, E. S. Liongosari, J. Q. Ning, and A. Olafsson, "Architecture specification support for component integration," in *Proc. of 7th International Workshop on CASE*, July 1995.

[70] J. Q. Ning, "Component-based software engineering (panel): Cbse enabling technologies (position statement)," in *Fourth Intl. Conference on Software Reuse*, IEEE Computer Society Press, April 1996.

[71] J. Q. Ning, "A component-based software development model," in *The Proc. of COMPSAC'96*, 1996.

[72] F. Bronsard, D. Bryan, W. Kozaczynski, E. Liongosari, J. Q. Ning, A. Olafsson, and J. Wetterstrand, "Toward software plug-and-play," in *Proceedings of the 1997 Symposium on Software Reusability (SSR'97)*, May 1997.

[73] O. M. Group, "Corba 2.0 specification," 1996.

[74] R. Prieto-Diaz, "Implementing faceted classification for software reuse," *Communication of ACM*, vol. 34, May 1991.

[75] C. Dellarocas, "Toward a design handbook for integrating software components," in *Proceedings of the 5th International Symposium on Assessment of Software Tools and Technologies (SAST'97)*, June 1997.

[76] C. Dellarocas, "A coordination perspective on software system design," in *Proceedings of the 9th Intl. Conference on Software Engineering and Knowledge Engineering (SEKE'97)*, June 1997.

[77] J. Magee, N. Dulay, and J. Kramer, "Regis: A constructive development environment for distributed programs," *Distributed Systems Engineering Journal*, vol. 1, no. 5, pp. 304–312, 1994.

[78] R. E. Johnson, "Frameworks=(components + patterns)," *Communications of the ACM*, vol. 40, October 1997.

[79] A. Schappert, P. Sommerlad, and W. Pree, "Automated support for software development with frameworks," in *Proc. of the ACM SIGSOFT Symposium on Software Reusability*, ACM Press, April 1995.

[80] T. J. Biggerstaff, "Design recovery for maintenance and reuse," *IEEE Computer*, July 1989.

[81] E. J. Chikofsky and J. H. Cross, "Reverse engineering and design recovery: A taxonomy," *IEEE Software*, vol. 7, January 1990.

[82] G. C. Gannod, Y. Chen, and B. Cheng, "An automated approach for supporting software reuse via reverse engineering," in *Proceedings of 13th IEEE International Conference on Automated Software Engineering (ASE'98)*, (Honolulu, Hawaii, USA), October 1998.

[83] R. Prieto-Diaz, "Domain analysis for reusability," in *Proceedings of 1987 IEEE Computer Software and Applications Conference (COMPSAC'87)*, (Tokyo, Japan), October 1987.

[84] G. Arango, *Domain Engineering for Software Reuse.* PhD thesis, University of California at Irvine, 1988.

[85] R. Prieto-Diaz and G. Arango, eds., *Domain Analysis and Software Systems Modeling.* IEEE Computer Society Press, 1991.

[86] J. Meekel, T. B. Horton, R. B. France, C. Mellone, and S. Dalvi, "From domain models to architecture frameworks," in *Proceedings of the 1997 Symposium on Software Reusability (SSR'97* (M. Harandi, ed.), May 1997.

[87] W. Humphrey, *Managing Software Process.* Reading, MA: Addison-Weslry, 1989.

[88] N. Medvidovic and R. N. Taylor, "A framework for classifying and comparing architecture description languages," in *Proceedings of ESEC/FSE97*, September 1997.

[89] I. Baxter, "Design maintenance systems," *Communications of the ACM*, April 1992.

[90] G. T. Leavens, "An overview of Larch/C++: Behavioral specifications for C++ modules," Tech. Rep. TR96-01a, Dept. of Computer Science, Iowa State University, March 1996.

[91] J. Penix and P. Alexander, "Toward automated component adaptation," in *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering*, June 1997.

[92] Y. Chen and B. H. C. Cheng, "Formalizing and automating component reuse." Proceedings of the 9th IEEE International Conference on Tools with Artifical Intelligence, November 1997.

[93] J.-J. Jeng and B. H. C. Cheng, "Using automated reasoning techniques to determine software reuse," *International Journal of Software Engineering and Knowledge Engineering*, vol. 2, December 1992.

[94] G. T. Leavens and W. E. Weihl, "Reasoning about object-oriented programs that use subtypes (extended abstract)," in *OOPSLA ECOOP '90 Proceedings* (N. Meyrowitz, ed.), vol. 25(10), pp. 212–223, Oct. 1990.

[95] J. Schumann and B. Fischer, "Nora/hammr: Making deduction-based software component retrieval practical," in *Proceedings of the 12th IEEE International Automated Software Engineering Conference (ASE97)*, (Incline Village, Nevada), November 1997.

[96] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, October 1969.

[97] P. America, "Designing an object-oriented programming language with behavioral subtyping," in *LNCS* (J. W. de Bakker, W. P. de Roever, and G. Rozenberg, eds.), vol. 489, pp. 60–90, Springer-Verlag, 1991.

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

[99] 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.

[100] E. Dijkstra, *A Discipline of Programming.* Prentice-Hall, 1976.

[101] C. Jones, *Software Development: A Rigorous Approach.* Prentice-Hall International, 1980.

[102] D. Gries, *The Science of Programming.* Springer-Verlag, 1981.

[103] E. Cohen, *Programming in the 1990s.* Springer-Verlag, 1990.

[104] R. Back, "A calculus of refinements for program derivations," *Acta Information*, vol. 25, pp. 593–624, 1988.

[105] J. Morris, "A theoretical basis for stepwise refinement and the programming calculus," *Science of Computer Programming*, vol. 9, no. 3, pp. 287–306, 1987.

[106] C. Morgan, *Programming from Specifications.* Prentice Hall, 1990.

[107] R. Back and J. von Wright, *Refinement Calculus: A Systematic Introduction.* Springer Verlag, 1998.

[108] P. Cousot, "Methods and logics for proving programs," in *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.), vol. B: Formal Models and Semantics, ch. 15, pp. 841–993, The MIT Press / Elsevier, 1990.

[109] C. A. R. Hoare and P. Lauer, "Consistent and complementary formal theories of the semantics of programming languages," *Acta Information*, vol. 3, pp. 135–155, 1974.

[110] W. H. Hesselink, *Programs, Recursion and Unbounded Choice.* Cambridge University Press, 1992.

[111] P. A. Fejer and D. A. Simovici, *Mathematical Foundations of Computer Science*, vol. I: Sets, Relations and Induction. New York: Springer-Verlag, 1990.

[112] G. Gratzer, *General Lattice Theory.* Basel: Birkhauser, 1978.

[113] A. M. Zaremski and J. M. Wing, "Signature matching: A key to reuse," in *Proceedings of the ACM SIGSOFT'93 Symposium on the Foundations of Software Engineering*, December 1993.

[114] L. Cardelli, "A semantics of multiple inheritance," *Information and Computation*, vol. 76, pp. 138–164, February/March 1988.

[115] J. Ousterhout, "Tcl: An embeddable command language," in *Proceedings of the 1990 Winter USENIX Conference*, 1990.

[116] J. Ousterhout, *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[117] B. Welch, *Practical Programming in Tcl and Tk.* Prentice Hall, 1997.

[118] J. Ousterhout, "An x11 toolkit based on the tcl language," in *Proceedings of the 1991 Winter USENIX Conference*, 1991.

[119] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide.* Addison-Wesley, 1999.

[120] G. C. Gannod and B. H. C. Cheng, "The object-oriented development of multimedia information systems," in *Multimedia Information Storage and Management* (S. M. Chung, ed.), Kluwer Academic Publishers, 1996.

[121] B. Cheng, Y. Chen, P. Fraley, G. Gannod, D. Judd, J. Kusler, H. Rither, S. Schafer, J. Sharnowski, S. Wagner, and E. Wang, "Design document for enforms ii: Decision support system for great lakes regional environmental information system," Tech. Rep. MSU-CPS-95-24, Michigan State University, 1995.

[122] T. DeMarco, *Structured Analysis and System Specification.* Prentice-Hall, 1979.

[123] E. Yourdon, *Modern Structured Analysis.* Englewood Cliffs, NJ: Yourdon Press, 1989.

[124] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Pattern.* Addison Wesley, 1994.

[125] M. Shaw, "Making choices: A comparison of styles for software architecture," *IEEE Software*, vol. 12, no. 6, pp. 27–41, 1995.

[126] M. Shaw, "Some patterns for software architecture," in *Proc. of Second Annual Conference on Pattern Language of Programming*, September 1995.

[127] C. H. Smith, *A Recursive Introduction to The Theory of Computation.* Springer-Verlag, 1994.

[128] C. H. Papadimitriou, *Computational Complexity.* Addison-Wesley Publishing Co., 1994.

[129] M. J. Morin, "Final report for enforms: Lac layer specification," 1993. MSU CPS814 term project report.

# Appendices

# Appendix A

# BNF Syntax of ABRIE V 2.0 ADL

We use an extended BNF grammar to define the syntax of the architectural description language (ADL) described in Chapter 5 and implemented in Chapter 8. The meanings of the notations are explained below.

```
  ::=    meaning ''is defined as''
  |      meaning ''or''
  <>     angle brackets used to surround non-terminal names
  []     items enclosed in [ and ] can appear 0 or 1 times
  []+    items enclosed in [ and ] can appear 1 or more times
  []*    items enclosed in [ and ] can appear 0 or more times
  '' ''  quotes ('') and ('') are used to surround terminals in order to
         distinguish these terminals from BNF meta-symbols
  ...    wildcards for anything before an ''ending symbol''

#system architecture
<system> ::= SYSTEM <idn> ['':'' <idn>] ''{'' <system_body> ''}''
<system_body> ::= [<visables>] [<auxiliarydef>] [<comp>]+ [<conn>]+ <config>
<visables> ::= VISABLE <v_elem> ['','' <v_elem>]* '';''
<v_elem> ::= <idn> ''.'' <idn>
<auxiliarydef> ::= ''%{'' ... ''%}''
<idn> ::= <alpha> [<alphanum>]*
<alpha> ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|
            A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|_
<numeric> ::= 0|1|2|3|4|5|6|7|8|9
<alphanum> ::= <aplha> | <numeric>

#component
<comp> ::= COMP <idn> ['':'' <idn>] <comp_rest>
<comp_rest> ::= '';'' | ''{'' [<uses>] [<port>]+ [<constr>] [<impl>] ''}''

#connector
<conn> ::= CONN <idn> ['':'' <idn>] <conn_rest>
<conn_rest> ::= '';'' | ''{'' <properties> ''}''

#configuration
<config> ::= CONFIGURE ''{'' <cfgs> ''}''
<cfgs> ::= [<cfg>] | <cfg> '';'' <cfgs>
<cfg> ::= <idn> ''.'' <idn> TO <idn> ''.'' <idn>

#port
<port> ::= PORT <idn> ['':'' <idn>] <port_rest>
<port_rest> ::= '';'' | ''{'' <resource> <properties> ''}''
<properties> ::= [<property>] | <property> '';'' <properties>
<property> ::= <idn> '':'' <property_value>
<property_value> ::= <idn> | <string> | <restype>
```

```
$ component constraints
<constr> ::= CONSTR ''{'' [<cstr>]* ''}''
<cstr> ::= <dependence> | <behaviorcstr>
<dependence> ::= DEPENDENCE ''{'' <idn> : <dpn> ''}''
<dpn> ::= <idn> | ''{'' <idn> ['','' <idn>]* ''}''
<behaviorcstr> ::= BEHAVIORCSTR ''{'' [<predicate>] ''}''
<predicate> ::= PRED: ... '';''

$implementation
<impl> ::= IMPLEMENTATION ''{'' properties ''}''

$resources
<resource> ::= <func> | <data> | <adt> | <stream> | <event> | <rbundle>
<restype> ::= FUNC | DATA | ADT | STREAM | EVENT | RBUNDLE

$function
<func> ::= FUNC <idn> <func_rest>
<func_rest> ::= '';'' | ''{'' [<params>] ''}'' [RETURN <type>] <func_rest2>
<params> ::= <param> | <param> '','' <params>
<param> ::= <type> | <type> <idn>
<type> ::= <idn> | <idn> ''*''
<func_rest2> ::= '';'' | ''{'' [<uses>] [<requires>] [<modifies>] [<ensures>] ''}''
<uses> ::= USES ... '';''
<requires> ::= REQUIRES ... '';''
<modifies> ::= MODIFIES ... '';''
<ensures> ::= ENSURES ... '';''

$data
<data> ::= DATA <idn> '':'' <idn> <data_rest>
<data_rest> ::= '';'' | WITHBEHAVIOR <behaviorspec>

$adt
<adt> ::= ADT <idn> <ptheader> <adt_rest>
<adt_rest> ::= '';'' | <behaviorspec>
<behaviorspec> ::= ''{'' [<uses>] [<func>]* ''}''
<ptheader> ::= ''<'' <idn> ['','' <idn>]* ''>''

$event
<event> ::= EVENT <idn> '';'' | EVENT <idn> ''{'' <predicate> ''}''

$stream
<STREAM> ::= STREAM <idn> '':'' <idn> '';''

$resource boundle
<rbundle> ::= RBUNDLE <idn> ''{'' [<resource>]* ''}''
```

# Appendix B

# Input File for Generating ABRIE V 2.0 ADL Lexical Analyzer

This appendix contains the input file to a lexical analyzer generator (Lex or Flex) for generating ABRIE V 2.0 ADL lexical analyzer. The generated lexical analyzer is called by the ABRIE ADL parser that itself is generated by a parser generator (Yacc or Bison). The parser generator also generates the C/C++ header file, `y.tab.h`, that defines the token constants returned by the lexical analyzer to represent various lexical tokens.

```
/*
 adl.l
 for ABRIE V 2.0 ADL

*/
%{
#include <stdio.h>
#include <string.h>
#include "y.tab.h"
%}

identifier       [_a-zA-Z][_a-zA-Z0-9]*
white            [ \t\n]
%option noyywrap

%%
\"[^"]*   {
           char s[1024];
           yyinput(); /* read the ending mark */
           strcpy(s, yytext);
           strcat(s, "\"");
           printf("\n%s", s);
           yylval.sval = strdup(s);
           return tkSTRING;
         }
LIBRARY|LIB  {
               printf("\n%s", yytext);
               return tkLIB;
             }
SYSTEM|SYS {
               printf("\n%s", yytext);
               return tkSYS;
             }
VISABLE {
          printf("\n%s", yytext);
          return tkVISABLE;
```

```
        }
COMPONENT|COMP   {
                 printf("\n%s", yytext);
                 return tkCOMP;
                }
CONNECTOR|CONN   {
                 printf("\n%s", yytext);
                 return tkCONN;
                }
CONFIGURE {
           printf("\n%s", yytext);
           return tkCONFIGURE;
          }
PORT        {
              printf("\n%s", yytext);
              return tkPORT;
             }
ROLE        {
              printf("\n%s", yytext);
              return tkROLE;
             }
CONSTR      {
              printf("\n%s", yytext);
              return tkCONSTR;
             }
DEPENDENCE  {
              printf("\n%s", yytext);
              return tkDEPENDENCE;
             }
BEHAVIORCSTR {
              printf("\n%s", yytext);
              return tkBEHAVIORCSTR;
             }
TO          {
              printf("\n%s", yytext);
              return tkTO;
             }
IMPLEMENTATION {
        printf("\n%s", yytext);
        return tkIMPLEMENTATION;
      }
FUNC  {
        printf("\n%s", yytext);
        yylval.sval = strdup(yytext);
        return tkFUNC;
      }

STREAM  {
        printf("\n%s", yytext);
        yylval.sval = strdup(yytext);
        return tkSTREAM;
      }
EVENT {
        printf("\n%s", yytext);
        yylval.sval = strdup(yytext);
        return tkEVENT;
      }
DATA  {
        printf("\n%s", yytext);
        yylval.sval = strdup(yytext);
        return tkDATA;
      }
ADT {
      printf("\n%s", yytext);
      yylval.sval = strdup(yytext);
      return tkADT;
    }
RBUNDLE {
```

```
      printf("\n%s", yytext);
      yylval.sval = strdup(yytext);
      return tkRBUNDLE;
    }
WITHBEHAVIOR {
      printf("\n%s", yytext);
      return tkWITHBEHAVIOR;
    }
RETURN|[Rr]eturn {
      printf("\n%s", yytext);
      return tkRETURN;
    }
[uU]ses|USES { char s[1024]; int i=0;
          printf("\n%s", yytext);
          while ((s[i++]=yyinput()) != ';');
          s[i-1] = '\0';
          yylval.sval = strdup(s);
          return tkUSES;
        }
TRAITS {
     printf("\n%s", yytext);
     return tkTRAITS;
    }

[Ff]or|FOR {
     printf("\n%s", yytext);
     return tkFOR;
    }

[rR]equires|REQUIRES  {
               printf("\n%s", yytext);
               char s[1024];
               int i=0;
               while ((s[i++]=yyinput()) != ';');
               s[i-1] = '\0';
               yylval.sval = strdup(s);
               return tkREQUIRES;
             }
[mM]odifies|MODIFIES  {
               printf("\n%s", yytext);
               char s[1024];
               int i=0;
               while ((s[i++]=yyinput()) != ';');
               s[i-1] = '\0';
               yylval.sval = strdup(s);
               return tkMODIFIES;
             }
[eE]nsures|ENSURES   {
               printf("\n%s", yytext);
               char s[1024];
               int i=0;
               while ((s[i++]=yyinput()) != ';');
               s[i-1] = '\0';
               yylval.sval = strdup(s);
               return tkENSURES;
             }
"PRED:"          {
               printf("\n%s", yytext);
               char s[1024];
               int i=0;
               while ((s[i++]=yyinput()) != '}');
               s[i-1] = '\0';
               yylval.sval = strdup(s);
               return tkPRED;
             }
defporttype {
          printf("\n%s", yytext);
          return tkDEFPORTTYPE;
```

```
      }
defcomptype {
         printf("\n%s", yytext);
         return tkDEFCOMPTYPE;
      }
ports   {
         printf("\n%s", yytext);
         return tkPORTS;
      }
defconntype {
         printf("\n%s", yytext);
         return tkDEFCONNTYPE;
      }
roles {
         printf("\n%s", yytext);
         return tkROLES;
     }

"."          {
               printf("\n%s", yytext);
               return tkDOT;
             }
","          {
               printf("\n%s", yytext);
               return tkCOMMA;
             }
":"          {
               printf("\n%s", yytext);
               return tkCOLON;
             }
";"          {
               printf("\n%s", yytext);
               return tkSEMICOLON;
             }
"*"          {
               printf("\n%s", yytext);
               return tkASTERISK;
             }
"("          {
               printf("\n%s", yytext);
               return tkLP;
             }
")"          {
               printf("\n%s", yytext);
               return tkRP;
             }
"<"          {
               printf("\n%s", yytext);
               return tkLAP;
             }
">"          {
               printf("\n%s", yytext);
               return tkRAP;
             }
"{"          {
               printf("\n%s", yytext);
               return tkLBP;
             }
"}"          {
               printf("\n%s", yytext);
               return tkRBP;
             }
"%{"         {
               printf("\n%s", yytext);
               char s[1024];
               int i=0;
               s[i]=yyinput();
               i++;
```

```
                while (!(((s[i]=yyinput()) == '}') && (s[i-1] == '%')))
                     i++;
                s[i-1] = '\0';
                yylval.sval = strdup(s);
                return tkAUXIDEF;
              }
{identifier} {
                printf("\n%s", yytext);
                yylval.sval = strdup(yytext);
                return tkIDENTIFIER;
              }


#[^\n]*\n  /* eat up one line comment beginning with # */

{white}+   /* eat up white space */

.           {
               printf("\nUnrecognized character: %s\n", yytext);
            }
%%

/* Following is testing code */
/*
YYSTYPE yylval;

int main(int argc, char *argv[])
 {
               printf("This is the begining ....\n");
    if (argc > 1)
             yyin = fopen(argv[1], "r");
    else
             return 1;
               while (yylex());
               fclose(yyin);
               printf("This is the end ... ...\n");
 }
*/
```

# Appendix C

# Input File for Generating ABRIE V 2.0 ADL Parser

This appendix contains the input file to a parser generator (Yacc or Bison) for generating ABRIE V 2.0 ADL parser. The parser calls the function yylex(), the ADL lexical analyzer generated by Lex or Flex, and is integrated with other components of the ABRIE system. The embedded C++ code in this file describes the semantics of the ADL.

```
/*
 adl.y
 for ABRIE V 2.0 ADL
*/
%{
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include "../model/rescls.h"
#include "../model/libcls.h"
#include "../model/archcls.h"
#include "../model/compcls.h"
#include "../model/portcls.h"
#include "../model/conncls.h"
#include "../model/sysbase.h"
#include "../model/trait.h"

extern FILE *yyin;

extern int yylex(void);
extern int yyerror(char *);

// 1111 for lib, 2222 for arch, 3333 for component
int _ParsedType = 0;
LibClass  *_currLib;
ArchClass *_currArch;
CompClass *_currComp;
ConnClass *_currConn;
PortClass *_currPort, *_thePort;
ResCls *_currRes;
FuncCls  *_currFunc;
StreamCls *_currStream;
EventCls *_currEvent;
AdtCls *_currAdt;
DataCls *_currData;
RBundleCls *_currRBundle;
traitsClass *_currTraits;
trait *_currTr;
```

```
int _inIMPL = 0;
int _inRBundle = 0;
int _inBehaviorBody = 0;
int _inPORT =0;

// for predefs sys info base
ptTypeSt *pt;
cpTypeSt *cp;
cnTypeSt *cn;
rlTypeSt *rl;

SysBase *_currSysBase;


%}

%union {
     char *sval;
     struct namepair *pval;
}


%token tkLIB
%token tkSYS
%token <sval> tkIDENTIFIER
%token <sval> tkSTRING
%token tkRETURN

%token tkVISABLE
%token tkCOMP
%token tkCONN
%token tkCONFIGURE
%token tkPORT
%token tkROLE

%token tkCONSTR

%token tkDEPENDENCE
%token tkBEHAVIORCSTR
%token tkIMPLEMENTATION

%token tkWITHBEHAVIOR


%token tkDEFPORTTYPE
%token tkDEFCOMPTYPE
%token tkPORTS
%token tkDEFCONNTYPE
%token tkROLES

%token tkTRAITS
%token tkFOR

%token <sval> tkSTREAM
%token <sval> tkEVENT
%token <sval> tkRBUNDLE
%token <sval> tkDATA
%token <sval> tkADT
%token <sval> tkFUNC

%token <sval> tkUSES
%token <sval> tkREQUIRES
%token <sval> tkMODIFIES
%token <sval> tkENSURES
%token <sval> tkPRED
%token <sval> tkAUXIDEF
```

```
%token tkCOMMA
%token tkCOLON
%token tkSEMICOLON
%token tkASTERISK
%token tkDOT
%token tkTO
%token tkLP
%token tkRP
%token tkLAP
%token tkRAP
%token tkLBP
%token tkRBP

%type <sval> type
%type <sval> uses
%type <sval> requires
%type <sval> modifies
%type <sval> ensures
%type <sval> predicate
%type <sval> auxilarydef
%type <sval> propertyvalue

%type <sval> ppvalue
%type <sval> restype

%type <pval> iddotpair

%%
start : sysarch { YYACCEPT; }
      | library { YYACCEPT; }
      | predefs { YYACCEPT; }
      | comp { YYACCEPT; }
      | traits {YYACCEPT;}
      ;

library : tkLIB {
     _ParsedType = 1111;
     _currLib = new LibClass();
           } comps
        | tkLIB  tkIDENTIFIER {
    _ParsedType = 1111;
    _currLib = new LibClass($2);
           } comps
        ;

sysarch : tkSYS tkIDENTIFIER {
             _ParsedType = 2222;
             _currArch = new ArchClass($2);
           }
          archtype archbody
        ;

archtype :
         | tkCOLON tkIDENTIFIER {_currArch->setType($2);}
         ;

archbody : tkLBP visables auxilarydef {
                 _currArch->insertAuxilaryDefs($3);
           } comps conns config tkRBP
         ;

visables :
         | tkVISABLE v_iddotpair restofiddotpairs tkSEMICOLON
         ;

restofiddotpairs :
                 | tkCOMMA v_iddotpair restofiddotpairs
                 ;
```

```
v_iddotpair : tkIDENTIFIER tkDOT tkIDENTIFIER {
                _currArch->insertVisable($1, $3);
              }
            ;

/* Components */
comps : comp
      | comp comps
      ;

comp  : tkCOMP tkIDENTIFIER {
          if (_ParsedType != 1111 && _ParsedType != 2222)
                _ParsedType = 3333;
            _currComp = new CompClass($2);
        }
        comptype  comprest {
            if (_ParsedType == 2222)
              _currArch->addComp(_currComp);
            else if (_ParsedType == 1111)
              _currLib->insert(_currComp);
        }
      ;

comptype :
         | tkCOLON tkIDENTIFIER { _currComp->setType($2);}
         ;

comprest : tkSEMICOLON
         | tkLBP uses {
             _currComp->setUses($2);
     /* (_currComp->traits).parse($2); */
           } ports constr impl tkRBP
         ;

ports : port
      | port ports
      ;

constr :
       | tkCONSTR tkLBP constraints tkRBP { _currComp->setCstrd();}
       ;

impl :
     | tkIMPLEMENTATION {
         _inIMPL =1;
       }
       tkLBP propertyspecs tkRBP {
         _inIMPL =0;
       }
     ;

constraints :
            | constraint constraints
            ;

constraint  : dependence
            | behaviorconstr
            ;

dependence : tkDEPENDENCE tkLBP tkIDENTIFIER {
               _thePort = _currComp->findPort($3);
               assert(_thePort != NULL);
             } tkCOLON pnlist tkRBP
           ;

pnlist : tkIDENTIFIER {
           _thePort->addDependence($1);
```

```
        }
      | tkLBP tkIDENTIFIER {
          _thePort->addDependence($2);
        } restofids tkRBP
      ;

restofids :
        | tkCOMMA tkIDENTIFIER {
            _thePort->addDependence($2);
          } restofids
        ;

behaviorconstr : tkBEHAVIORCSTR tkLBP predicate tkRBP {
                    _currComp->setBehaviorCstr($3);
                 }
               ;

port : tkPORT tkIDENTIFIER {
         _currPort = new PortClass($2);
       } porttype portrest {
         _currComp->addPort(_currPort);
       }
     ;

porttype :
        | tkCOLON tkIDENTIFIER {
            _currPort->setType($2);
          }
        ;

portrest : tkSEMICOLON
         | tkLBP {
             _inPORT = 1;
           } resspec propertyspecs tkRBP {
             _inPORT = 0;
           }
         ;

propertyspecs :
              | propertyspec
              | propertyspec tkSEMICOLON propertyspecs
              ;

propertyspec : tkIDENTIFIER tkCOLON propertyvalue {
                 if (_inIMPL)
                   _currComp->addImplProperty($1, $3);
                 else if (_inPORT)
                   _currPort->addProperty($1, $3);
               }
             ;

propertyvalue : tkIDENTIFIER {$$ = $1;}
              | tkSTRING {$$ = $1;}
              | restype {$$ = $1;}
              ;

resspecs : resspec
         | resspec resspecs
         ;

resspec : funcspec
        | dataspec
        | adtspec
        | streamspec
        | eventspec
        | rbundlespec
        ;
```

```
restype : tkSTREAM {$$ = $1;}
        | tkEVENT {$$ = $1;}
        | tkDATA {$$ = $1;}
        | tkADT {$$ = $1;}
        | tkFUNC {$$ = $1;}
        | tkRBUNDLE {$$ = $1;}
        ;

streamspec : tkSTREAM tkIDENTIFIER tkCOLON tkIDENTIFIER tkSEMICOLON {
                _currStream = new StreamCls($2, $4);
                if (_inRBundle)
                    _currRBundle->insertRes(_currStream);
                else
                    _currPort->setResource(_currStream);
              }
           ;

eventspec : tkEVENT tkIDENTIFIER {
                 _currEvent = new EventCls($2);
                if (_inRBundle)
                    _currRBundle->insertRes(_currEvent);
                else
                    _currPort->setResource(_currEvent);
             } eventrest
          ;

eventrest : tkSEMICOLON
          | tkLBP predicate tkRBP {
              _currEvent->setCondition($2);
            }
          ;

rbundlespec : tkRBUNDLE tkIDENTIFIER {
                _currRBundle = new RBundleCls($2);
                _inRBundle = 1;
              } tkLBP resspecs tkRBP {
                _currPort->setResource(_currRBundle);
                _inRBundle = 0;
              }
            ;

dataspec : tkDATA tkIDENTIFIER tkCOLON tkIDENTIFIER {
                 _currData = new DataCls($2, $4);
                if (_inRBundle)
                    _currRBundle->insertRes(_currData);
                else
                    _currPort->setResource(_currData);
           } datarest
         ;

datarest : tkSEMICOLON
         | tkWITHBEHAVIOR behaviorspec
         ;

adtspec : tkADT tkIDENTIFIER {
                _currAdt = new AdtCls($2);
                if (_inRBundle)
                    _currRBundle->insertRes(_currAdt);
                else
                    _currPort->setResource(_currAdt);
          } ptheader adtrest
        ;

adtrest : tkSEMICOLON
        | behaviorspec
        ;

behaviorspec : tkLBP behaviorbody tkRBP
```

```
            ;

behaviorbody :
            |uses {
                _currAdt->setComUses($1);
                /* (_currAdt->traits).parse($1);*/
                _inBehaviorBody=1;
              } funcspecs { _inBehaviorBody = 0; }
            ;

funcspecs : funcspec
          | funcspec funcspecs
          ;

funcspec : tkFUNC tkIDENTIFIER {
              _currFunc = new FuncCls($2);
              }
           funcrest {
              if (_inBehaviorBody)
                 if (strcmp($2, _currAdt->getName())==0)
                    _currAdt->addAConstructor(_currFunc);
                 else
                    _currAdt->addABehavior(_currFunc);
              else if (_inRBundle)
                      _currRBundle->insertRes(_currFunc);
              else
                 _currPort->setResource(_currFunc);
   }
         ;

 /* procedure */
funcrest : tkSEMICOLON
         | tkLP params tkRP retspec funcrest2
         ;
/* retspec */
retspec :
        | tkRETURN type {
            _currFunc->setRetType($2);
          }
        ;
/* procedure LARCH specification */
funcrest2 : tkSEMICOLON
          | tkLBP uses requires modifies ensures tkRBP {
                _currFunc->setUses($2);
                /* (_currFunc->traits).parse($2); */
                _currFunc->setRequires($3);
                _currFunc->setModifies($4);
                _currFunc->setEnsures($5);
             }
          ;

uses : { $$ = strdup(""); }
     | tkUSES { $$ = $1; }
     ;
requires : { $$ = strdup(""); }
         | tkREQUIRES { $$ = $1; }
         ;
modifies : { $$ = strdup(""); }
         | tkMODIFIES { $$ = $1; }
         ;
ensures  : { $$ = strdup(""); }
         | tkENSURES { $$ = $1; }
         ;

predicate : { $$ = strdup("");}
          | tkPRED { $$ = $1;}
          ;
```

```
auxilarydef : { $$ = strdup("");}
            | tkAUXIDEF { $$ = $1;}
            ;

/* paramters */
params :
       | paramlist
       ;

paramlist : param
          | param tkCOMMA paramlist
          ;

param  : type {
            _currFunc->appendParamType($1);
            _currFunc->appendParam("");
         }
       | type tkIDENTIFIER {
            _currFunc->appendParamType($1);
            _currFunc->appendParam($2);
         }
       ;

type   : tkIDENTIFIER { $$ = $1;}
       | tkIDENTIFIER tkASTERISK {
           char *s=strdup($1);
           safeStrcat(s, "*");
           $$ = s;
         }
       ;

/* paramaterized types of an ADT resource  */
ptheader :
         | tkLAP ptlist tkRAP
         ;

ptlist : tkIDENTIFIER {
               _currAdt->addParamType($1);
            }
       | tkIDENTIFIER {
               _currAdt->addParamType($1);
            }
         tkCOMMA ptlist
       ;

conns : conn
      | conn conns
      ;

conn : tkCONN tkIDENTIFIER {
         _currConn = new ConnClass($2);
       } conntype connrest {
         _currArch -> addConn(_currConn);
       }
     ;

conntype :
         | tkCOLON tkIDENTIFIER {
               _currConn->setType($2);
            }
         ;

connrest : tkSEMICOLON
         | tkLBP propertyspecs tkRBP
         ;

config : tkCONFIGURE tkLBP cfglist tkRBP
       ;
```

```
cfglist :
        | cfg
        | cfg tkSEMICOLON cfglist
        ;

cfg : iddotpair tkTO iddotpair {
        PortClass *p = _currArch->findPort($1->majorname,
   $1->minorname);
        assert(p!=NULL);
        p->addCFG(new PortCFG($3->majorname, $3->minorname));

        RoleStruct *r = _currArch->findRole($3->majorname,
                                            $3->minorname);
        assert(r!=NULL);
        strcpy(r->CFG_compName, $1->majorname);
        strcpy(r->CFG_portName, $1->minorname);
      }
    ;

iddotpair : tkIDENTIFIER tkDOT tkIDENTIFIER {
                $$ =  new namepair($1, $3);
              }
          ;

/* Syntax for predefined system information */
predefs : defelem
        | defelem predefs
        ;

defelem : defporttp
        | defcomptp
        | defconntp
        ;

defporttp : tkDEFPORTTYPE tkIDENTIFIER {
                pt = new ptTypeSt($2);
             }
            tkLBP porttpspec tkRBP {
                _currSysBase->insertPortType(pt);
     }
          ;

porttpspec : pproperty
           | pproperty tkSEMICOLON porttpspec
           ;

pproperty : tkIDENTIFIER tkCOLON ppvalue {
                (pt->properties).addPair(new pair($1, $3));
              }
          ;

ppvalue : tkIDENTIFIER {$$ = $1;}
        | restype {$$ = $1;}
        ;

defcomptp : tkDEFCOMPTYPE tkIDENTIFIER {
                cp = new cpTypeSt($2);
             }
            tkLBP comptpspec tkRBP {
                _currSysBase->insertCompType(cp);
      }
          ;

comptpspec : tkPORTS tkCOLON tkLBP cportlist tkRBP
           ;

cportlist : tkIDENTIFIER {
```

```
                (cp->supportPortList).insertAtEnd(strdup($1));
        }
            | tkIDENTIFIER {
                (cp->supportPortList).insertAtEnd(strdup($1));
        }
              tkCOMMA cportlist
            ;

defconntp : tkDEFCONNTYPE tkIDENTIFIER {
                cn = new cnTypeSt($2);
              }
              tkLBP conntpspec tkRBP {
                _currSysBase->insertConnType(cn);
        }
            ;

conntpspec : tkROLES tkCOLON  tkLBP rlist tkRBP
           ;

rlist : rspec
      | rspec tkSEMICOLON rlist
      ;

rspec : tkIDENTIFIER {
                rl = new rlTypeSt($1);
        }
        tkCOLON tkLBP rdomlist tkRBP {
                (cn->rList).insertAtEnd(rl);
}
      ;

rdomlist : tkIDENTIFIER {
                (rl->configurablePortList).insertAtEnd($1);
           }
         | tkIDENTIFIER {
                (rl->configurablePortList).insertAtEnd($1);
           }
           tkCOMMA rdomlist

/* Handle Larch Shared Language traits introduced by uses */
traits : tkTRAITS trlist
       ;

trlist :
       | trlist1 tkSEMICOLON
       ;

trlist1 : trait
        | trait tkCOMMA trlist1
        ;

trait : tkIDENTIFIER {
           _currTr = new trait;
           strcpy(_currTr->name, $1);
        }
      | tkIDENTIFIER {
           _currTr = new trait;
          strcpy(_currTr->name, $1);
        } tkLP forpairs tkRP {
            _currTraits->addTrait(_currTr);
           }
      ;

forpairs :
         | forpairs1
         ;

forpairs1 : forpair
```

```
          | forpair tkCOMMA forpairs1
          ;

forpair : tkIDENTIFIER tkFOR tkIDENTIFIER {
            (_currTr->forPairs).addPair(new pair($3, $1));
  }
        ;


%%

/* Following is test code */
/*
int yyparse();
int main(int argc, char *argv[])
 {
   if (argc > 1) {
      yyin = fopen(argv[1], "r");
      yyparse();
      fclose(yyin);
    }
   return 0;
 }
*/
int yyerror(char *s)
{
  printf("%s", s);
  return 0;
}
```

# Appendix D

# Architectural Knowledge Description File

This appendix contains a file that is loaded when ABRIE is executed, and it provides information about types of architectural elements supported by the current runtime system. By modifying this file, users can customize ABRIE environment for specific purposes.

```
#
# adl.predefs
#
# Predefined types: meta-knowledge about software architectures.
#

# Port Types
defporttype ProcDef {
            restype : FUNC;
            direction : OUT;
            connectivity : MULTIPLE
        }
defporttype ProcInvoc {
            restype : FUNC;
            direction : IN;
            connectivity : SINGLE
        }
defporttype DataDef {
            restype : DATA;
            direction : OUT;
            connectivity : MULTIPLE
        }
defporttype DataUse {
            restype : DATA;
            direction : IN;
            connectivity : SINGLE
        }
defporttype ADTDef {
            restype : ADT;
            direction : OUT;
            connectivity : MULTIPLE
        }
defporttype ADTUse {
            restype : ADT;
            direction : IN;
            connectivity : SINGLE
        }
defporttype EventAnnounce {
            restype : EVENT;
```

```
            direction : OUT;
            connectivity : MULTIPLE
        }
defporttype EventListen {
            restype : EVENT;
            direction : IN;
            connectivity : SINGLE
        }
defporttype InStream  {
            restype : STREAM;
            direction : IN;
            connectivity : SINGLE
        }
defporttype OutStream {
            restype : STREAM;
            direction : OUT;
            connectivity : SINGLE
        }
defporttype RBundleDef {
            restype : RBUNDLE;
            direction : OUT;
            connectivity : MULTIPLE
        }
defporttype RBundleUse {
            restype : RBUNDLE;
            direction : IN;
            connectivity : SINGLE
        }

$component types
defcomptype Module {
    ports : {ProcDef, ProcInvoc, DataDef, DataUse, ADTDef, ADTUse,
             RBundleDef, RBundleUse, EventAnnounce, EventListen}
}
defcomptype Filter {
    ports : {InStream, OutStream}
}
defcomptype Process {
    ports : {InStream, OutStream, EventAnnounce, EventListen}
}
defcomptype Generic {
    ports : {ProcDef, ProcInvoc, DataDef, DataUse, ADTDef,
             ADTUse,InStream, OutStream, RBundleDef, RBundleUse,
             EventAnnounce, EventListen}
}

$Connector types
defconntype CallProc {
    roles : { Definer : {ProcDef};
              Caller  : {ProcInvoc}
            }
  }
defconntype AccessData {
    roles : { Definer : {DataDef};
              User    : {DataUse}
            }
  }
defconntype UseADT {
    roles : { Definer : {ADTDef};
              User    : {ADTUse}
            }
  }
defconntype UseRBundle {
    roles : { Definer : {RBundleUse};
              User : {RBundleDef}
            }
  }
defconntype Pipe {
```

```
    roles : { Source : {InStream};
              Sink   : {OutStream}
            }
  }
defconntype EventProc {
    roles : { Announcer : {EventAnnounce};
              Listener  : {EventListen}
            }
  }
```

# Appendix E

# LSL traits for the LCB specifications of ENFORMS

The formal specifications and analysis of the Local Control Brokers (LCB) are based on a number of Larch Shared Language (LSL) traits that provide primitive sorts and operators necessary for specifying and reasoning about the LCB components. These traits were originally developed by Michele J. Morin [129] and are used in our case study. We include several important LSL traits in this appendix for references. As a historic note, the term "Local Control Broker" was not used at the beginning of the ENFORMS project. Instead, "Local Access Control" was the term originally used. As a result, instead of having *lcb.lsl*, we have *lac.lsl* in this appendix.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   itemreg.lsl
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
itemreg (IR) : trait
% This trait describes the item registry file.
% It is expected to have a series of correct
% item descriptors, otherwise it is considered
% in error.

assumes plib,
dictionary(dictionary for D, string for K, ID for V),
file(string for C, FILE for F)

includes itemdesc

% In the implementation, ItemRegistry inherits from the Dictionary
% class.  However, in this specification, dictionary was specified
% as a simple trait, not a full class in it's own right.
% Therefore, instead of inheriting from Dictionary, IR will
% have a dictionary as part of it's sort definition.

IR tuple of registry : dictionary,
    changebit : int,
      defaultSaveName : string

introduces

% The following are the operations on the entire set of
% Item Descriptors.

validRegistry : FILE -> Bool
getItemDescriptors  : FILE, string -> dictionary
getID : FILE -> ID
writeItemDescriptors : FILE, dictionary -> FILE
```

```
asserts
\forall f: FILE, s: string, d: dictionary


% validRegistry returns true if all the item descriptors in the item
% registry file are valid descriptors.
validRegistry(f) == if \not (eof(f))
    then atBegItem(f) \and validID(skipBegItem(f))
\and validRegistry(skipEndItem(skipBegItem(f)))
    else true;

% getID gets the values of an item descriptor from the item registry file
% and stores them in the internal representation of the item descriptor.
getID(f) ==    [getKey(skipBegItem(f)),
       getextType(skipBegItem(f)),
       getidescript(skipBegItem(f)),
       s,
       countManipulators(skipBegItem(f)),
       getManipulators(skipBegItem(f)) ];

% getItemDescriptors adds all item descriptors (in their internal representation)
% to the registry dictionary.
getItemDescriptors(f, s) ==
if \not(eof(f))
then add( assoc( getKey(skipBegItem(f)), getID(f)),
  getItemDescriptors(skipEndItem(skipBegItem(f)),s) )
else new;

% writeItemDescriptors writes all item descriptors stored in the registry
% dictionary to the item registry file.
writeItemDescriptors(f, d) == if isEmpty(d)
      then f
      else writeItemDescriptors(
writeID(writeBegItem(f),
value(head(d)) ),
              tail(d) ) ;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   app.lsl
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
app : trait
% This trait abstract the app manipulation of a database item.
includes character,
   String( string for C, char for E )

assumes system

includes dictionary( dictionary for D, string for K, string for V )

% This dictionary is the Parameter dictionary of the
% Manipulator Descriptor.

introduces
initialized  : -> Bool
validAppParams : dictionary -> Bool
getPath : dictionary -> string
getName : dictionary -> string
getPresets : dictionary -> string
getParms : dictionary -> string
fullname : string, string -> string
appActivated : dictionary, string -> Bool

asserts
\forall p : dictionary, s : string

% For app, initialized is always true.  It is only used for consistency
```

```
% among all manipulators.
initialized  == true;

% validAppParams returns true if the Keys in the parameter dictionary
% p are the expected strings for app.
validAppParams(p) == (capP-|(capR-|(capO-|(capG
-|(capR-|(capA-|(capM-|(space
-|(capP-|(capA-|(capT-|(capH
-| {} )))))))))))) \in p
   \and \not isEmpty( getPath(p) )

   \and (capP-|(capR-|(capO-|(capG
                              -|(capR-|(capA-|(capM-|(space
                              -|(capF-|(capI-|(capL-|(capE
-| {} )))))))))))) \in p
   \and \not isEmpty( getName(p) )

   \and (capP-|(capR-|(capE-|(capE
-|(capX-|(capE-|(capC-|(capU
-|(capT-|(capI-|(capO-|(capN
-|(space-|(capC-|(capO-|(capM
-|(capM-|(capA-|(capN-|(capD
-|(capS-| {} ))))))))))))))))))) \in p
   \and \not isEmpty( getPresets(p) )

   \and (capE-|(capX-|(capE-|(capC
-|(capU-|(capT-|(capI-|(capO
-|(capN-|(space-|(capP-|(capA
-|(capR-|(capA-|(capM-|(capE
-|(capT-|(capE-|(capR-|(capS-|{}
))))))))))))))))))) \in p
   \and \not isEmpty( getParms(p) ) ;

% getPath gives the value associated with the key "PROGRAM PATH".
getPath(p)  == value(lookup( capP-|(capR-|(capO-|(capG
                              -|(capR-|(capA-|(capM-|(space
                              -|(capP-|(capA-|(capT-|(capH
                              -| {} )))))))))), p) ) ;

% getName gives the value associated with the key "PROGRAM NAME".
getName(p) == value(lookup( capP-|(capR-|(capO-|(capG
                              -|(capR-|(capA-|(capM-|(space
                              -|(capF-|(capI-|(capL-|(capE
                              -| {} )))))))))), p) );

% getPresets gives the value associated with the key
% "PREEXECUTION COMMANDS".
getPresets(p) == value(lookup( capP-|(capR-|(capE-|(capE
                              -|(capX-|(capE-|(capC-|(capU
                              -|(capT-|(capI-|(capO-|(capN
                              -|(space-|(capC-|(capO-|(capM
                              -|(capM-|(capA-|(capN-|(capD
                              -|(capS-| {} ))))))))))))))))))), p ) );

% getParms gives the value associated with the key
% "EXECUTION PARAMETERS".
getParms(p) == value(lookup( capE-|(capX-|(capE-|(capC
                              -|(capU-|(capT-|(capI-|(capO
                              -|(capN-|(space-|(capP-|(capA
                              -|(capR-|(capA-|(capM-|(capE
                              -|(capT-|(capE-|(capR-|(capS-|{}
                              ))))))))))))))))))), p ) );


% fullname: The activation of an application is system dependent, and
% therefore is an implementation issue. So fullname will take the name
% and the path amd return the full path of the application to be run.
% This is all the information that can be given for the specification
```

```
% of fullname, without going into implementation detail.

% It is assumed that if the fullname, given the application name and
% path, exists then the application is run on the given display, s.
% Anything more is an implementation issue.
% getPresets should also be used to execute any necessary commands
% before exectuting the application.

appActivated(p,s)== access(fullname(getName(p), getPath(p)));



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   lac.lsl
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
lac : trait
% Describes the operations performed by the LAC layer interface.

assumes plib, system

includes List( sList for C, string for E),
 List( iList for C, ID for E),
 itemreg, imview, app, grass, textdisp

introduces
ListToString  : sList, IR -> string
StringToList : string -> iList
itemDescrString : string, IR -> string
activated : MD, string -> Bool

% For specifications only.  The implementation requires the full path along with the
% file name of the database.  fullname will use an implementation specific environment
% variable, along with the database file name and return the full pathname.
fullname : string -> string

asserts
\forall sl : sList, s : string, i : IR, m : MD

% ListToString takes a List of Item Descriptor keys and creates a string
% containing all the Item Descriptors for these keys as strings.
ListToString(sl, i) == if isEmpty(sl)
      then {}
      else itemDescrString(head(sl), i) ||
ListToString(tail(sl), i);

% itemDescrString takes a key s and writes the Item Descriptor corresponding
% to the key to a string.
itemDescrString(s, i) == fileString(writeID(writeBegItem([{},{}]),
    value(lookup(s, i.registry))));

% StringToList converts a string of Item Descriptors in string format to
% a List of Item Descriptors in their internal representation.
StringToList(s) == if ~isEmpty(s) /\ validID(skipBegItem([{},s]))
   then getID([{}, s])
-| StringToList(fileToEnd(
   skipEndItem(skipBegItem([{},s]))))
   else {};

% activated returns true if the given manipulator is activated correctly.
activated(m, s) == if m.manipulator = (smI-|(smM-|(smV-|(smI-|(smE
-|(smW-|{}))))))
   then imviewActivated(m.theParameters, s)
   else (if m.manipulator = (smT-|(smE-|(smX-|(smT
    -|(smD-|(smI-|(smS-|(smP
    -|(smL-|(smA-|(smY-|{}))))))))))))
 then textdispActivated(m.theParameters, s)
 else (if m.manipulator = (smG-|(smR-|(smA
 -|(smS-|(smS-|{}))))))
```

```
      then grassActivated(m.theParameters, s)
      else (if m.manipulator =
(smA-|(smP-|(smP-|{})))
   then appActivated(m.theParameters, s)
   else false)));
```