Toward Automated Component Adaptation

John Penix and Perry Alexander

Knowledge-Based Software Engineering Lab
Department of Electrical and Computer Engineering and Computer Science
University of Cincinnati
Cincinnati, Ohio 45221-0030
E-mail: jpenix@ece.uc.edu,alex@ece.uc.edu

Abstract

This paper explores the use of specification matching to discover and select component adaptation strategies. This is done within a formal framework that integrates specification-based component retrieval with a formal architecture representation to support component retrieval and adaptation. The key to integration is determining the relationship between what components are potentially reusable and how they can be properly adapted. An example is given to illustrate how the results of specification matching can be used to guide the selection and application of adaptation tactics and automate component adaptation.

1. Introduction

As software systems are becoming larger and more complex, the demand for high levels of reliability and productivity is also increasing. New techniques and tools will be necessary to enable design methodologies to meet these demands. Furthermore, successful development methodologies must support a high degree of automation to make this investment economically attractive.

This paper describes part of an effort to use formal specifications to assist and automate a system design process based on reusable components and architectures. Identification of reusable components is done using *specification matching* [17], a general framework for applying formal reasoning to verify that a logical relationship holds between two specifications. Component adaptation is supported by using *architecture theories* to declaratively specify system structure. While these provide formal mechanisms for component retrieval and adaptation, providing automated support for the entire reuse process requires heuristics for selecting adaptation strategies based on the available components and architectures. The goal of this paper is to explore the use of specification matching results to discover and select component adaptation strategies.

In the next section we present and overview of our design system. This is followed by an introduction to the specification formalisms used in the system. The next two sections provide details of specification matching to determine component reusability and the method of architecture specification. We then discuss how these two are integrated by using specification matching results to guide component adaptation and give an example of adaptation. Finally, we present related work and conclude by summarizing and discussing future research directions.

2. System Overview

Figure 1 depicts the flow of information within our framework. In the diagram, boxes represent data structures, ovals represent computations and arrows represent data flow into computations and references between data structures. In the framework, formal specifications are used to model the problem requirements and the function of the library components. The specifications are based on abstract data types defined in an algebraic domain theory [4, 15].

Component retrieval is made efficient by layering a classification scheme above the specifications. The classification scheme consists of a collection of formal definitions representing possible component features in the domain. The formalization of the scheme permits automated classification of the specifications. The retrieval mechanism is based on syntactic comparison of features sets. The details of this specific specification-based retrieval mechanism are discussed elsewhere [8]. The components returned by the retrieval mechanism are passed on to a more detailed evaluation that uses specification matching to determine the relationship that exists between each of the retrieved components and the requirements specification.

The adaptation phase uses the specification matching results to select a mechanism to adapt or combine the retrieved

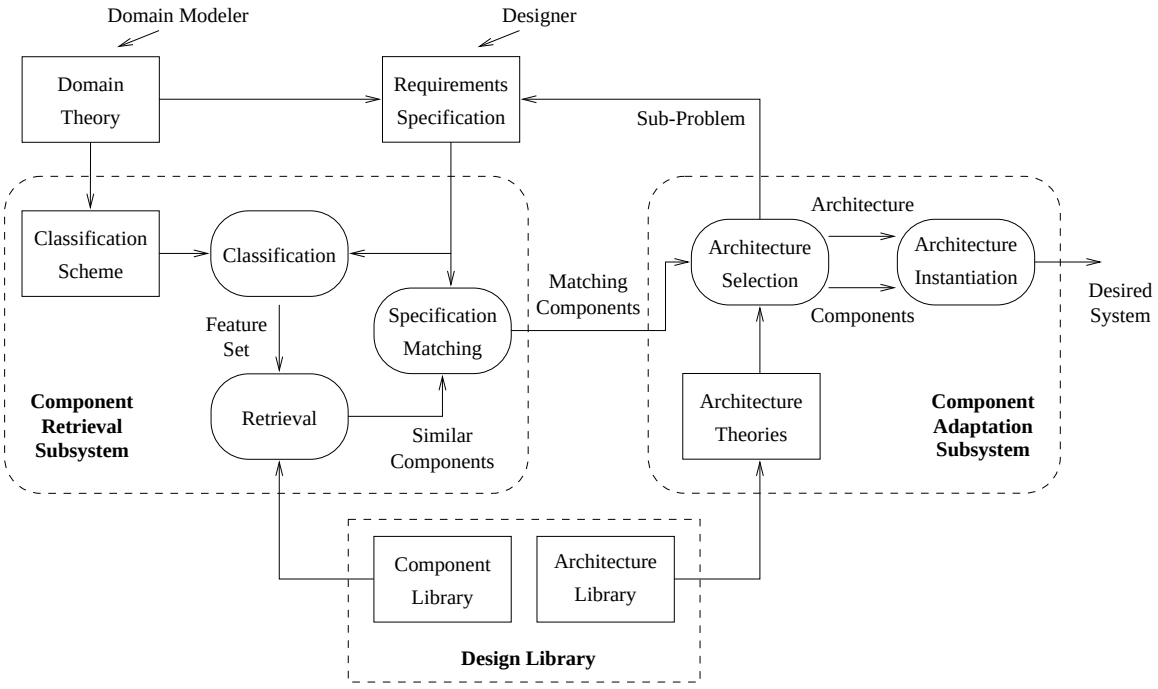


Figure 1. Overview of Component Retrieval and Adaptation Activities

components to solve the problem. During adaptation, it may be necessary to locate or construct additional components by issuing a subproblem back to the component retrieval system.

The structures used for component adaptation are modeled formally using *architecture theories*. An architecture theory specifies the behavior of a system in terms of the behavior of its subcomponents via a collection of axioms. The system decomposition specified by an architecture theory is implemented by an architecture schema that describes an architecture in an *architecture description language* [10]. Each architecture theory has one or more corresponding schemas in the architecture library. Once an architecture theory is selected that correctly combines components to solve the problem, a schema is selected from the library and instantiated with the components. The result is a system with the desired properties.

3. Formalisms

In our work, components are procedural abstractions representing subroutines, methods, or chips. Components take inputs and perform some computation that results in changes to their outputs. Component behavior and architectures are both specified declaratively using theories.

3.1. Theories

A *theory* is a unit of encapsulation for algebraic specifications [1]. Theories define *operations* over a collection of *sorts* and constrain the behavior of the operations via a set of *axioms*. A sort, like a type, is a set of values. Operations specify how to construct, modify and differentiate values of the sort. The axioms define terms with equivalent values in the sort. A variety of algebraic specification languages and tools are available to represent and debug theories. We will use the Larch Shared Language (LSL) [4] through out the paper.

Theory morphisms are the formal mechanism underlying two methods of constructing larger theories from smaller ones: *extension* and *parameterization* [13]. A theory morphism maps the sorts and operators of one theory to sorts and operators of another theory such that the axioms of first theory are theorems in the second theory [14]. Theory B is an *extension* of theory A if B contains all of the sorts, operators and axioms of A. An extension is represented by a theory morphism from A to B that maps each sort and operator to itself in the target theory.

A *parameterized theory* is a pair of theories: a *parameter theory* and a *target theory* that is an extension of the parameter theory [1]. A parameter theory is instantiated using a theory morphism that maps the parameter theory to the *actual parameter* as shown in Figure 2. The *induced passing*

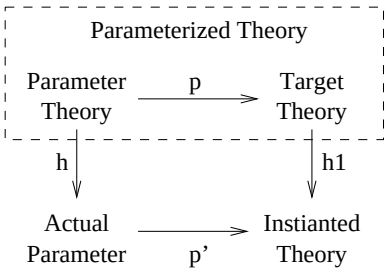


Figure 2. Parameter Passing Diagram

morphism $h1$ maps the sort and operator symbols originating from the parameter theory (via p) according to the translation defined by h. In the resulting *instantiated theory* the axioms of the target theory are expressed in the language of the actual parameter. This has the effect of replacing the parameter theory by the actual parameter. If the axioms of the actual parameter are valid theorems in the instantiated theory, then the diagram commutes ($p' \circ h = h1 \circ p$) and the parameter instantiation is correct.

In LSL, theory extension and parameterization are expressed using the `includes` directive. Arguments to the theories simplify renaming of sorts and operations.

3.2. Component Specifications

An interface specification for a component consists of two predicates, a *precondition* and *postcondition*, defined over the domain and range of the component. The precondition specifies the set of domain values that have a defined output, called the *legal inputs* to the problem. The output condition specifies the relationship that must hold between a legal input and a *feasible output*. A specification for a component C with precondition I_C and postcondition O_C has the following interpretation:

> If execution of C is begun in a state satisfying I_C, then it is guaranteed to terminate in a finite amount of time in a state satisfying O_C. [3]

A component interface specification can be represented formally as a *component theory*. A component theory is an extension of a *problem theory* [14]. A problem theory specifies the relationship between the domain and range of a specification and the precondition and postcondition, as shown in Figure 3(a). To specify a specific problem, the general problem theory is extended by adding definitions for the domain, range, precondition and postcondition.

A component theory extends problem theory by adding an axiom stating that a valid output exists for every legal input. This is a formalization of the interpretation of the relationship between the precondition and postcondition of a specification. The general component theory is shown in Figure 3(b).

(a)
```
ProblemTheory(D,R,I,O) : trait
introduces
 I : D → Bool
 O : D, R → Bool
```

(b)
```
ComponentTheory : trait
includes
 ProblemTheory(D,R,I,O)
asserts
 ∀ x:D ∃ z:R  I(x) ⇒ O(x,z)
```

Figure 3. Problem and Component Theories

4. Component Retrieval

Formal interface specifications describe the functionality that is encapsulated by a component. By comparing this description with a component requirements specification, we can determine if a component provides the correct functionality to be used within a system. First, the requirements of the desired component are modeled by an interface specification referred to as the problem or query specification. This specification is then matched against the specifications of existing components.

Using specification matching to evaluate reusability requires a formal definition of the relationship that must exist for a component to be reused to solve a new problem. A component completely solves a problem if it results in one of the problem's valid outputs for each of the problem's legal inputs. Formally, component specification C satisfies problem specification P if the following condition holds:

$$\forall x : R, z : D$$
$$(I_P(x) \Rightarrow I_C(x)) \wedge (I_P(x) \wedge O_C(x, z) \Rightarrow O_P(x, z))$$

The first conjunct states that the component will accept all legal inputs to the problem. The second conjunct requires that all valid outputs of the component for a legal problem input are valid outputs of the problem. In the standard notation for specification matches [17], universal quantification is assumed and the variable arguments for the predicates are dropped:

$$match_{satisfies}(C, P) = (I_P \Rightarrow I_C) \wedge (I_P \wedge O_C \Rightarrow O_P)$$

This definition indicates that $match_{satisfies}(C, P)$ will be true when C satisfies P.

It is not always the case that a component completely satisfying the problem exists. Therefore, it is desirable for a query to match components that can be adapted or combined to solve the problem. Zaremski and Wing [16, 17] have

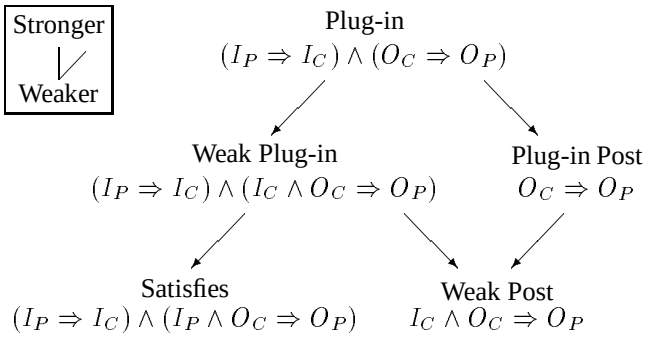


Figure 4. A Lattice of Specification Matches

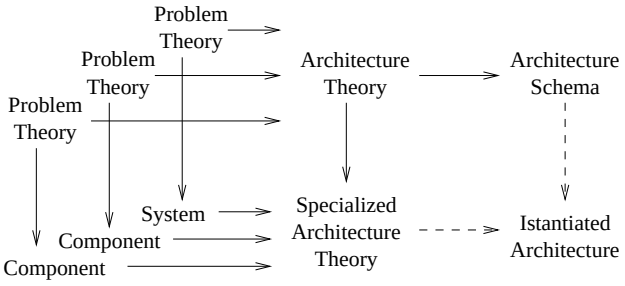


Figure 5. Overview of System Design Using Architecture Theories

identified various specification matches that are useful for evaluating forms of reusability. Figure 4 shows a lattice of specification matches that are of interest in determining reusability. An arrow between two matches indicates that the match at the base of the arrow is stronger than (logically implies) the match at the head of the arrow.

The three matches on the left-most path require the precondition of the problem be stronger than the precondition of the component. They differ in the set of inputs that require a valid output from the postcondition relation of the component. Because of the logical relationship between these matches, any components matching under Plug-in or Weak Plug-in will match under Satisfies.

The Plug-in Post and Weak Post matches differ from the others by not requiring all legal problem inputs to be legal component inputs. If a component matches a problem in one of these ways, there are problem inputs that cause unspecified behavior in the component. However, it may be possible to compose a collection of such components to provide a complete solution.

Due to the complexity of automated theorem proving for first order logic, specification matching is too computationally expensive to test a large number of components [16]. There is ongoing research directed toward finding ways to make component retrieval efficient while maintaining the semantic precision provided by the formal specifications [2, 5, 8].

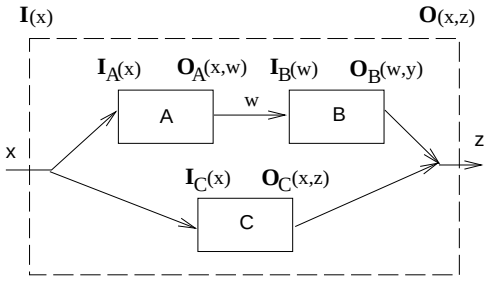
5. Architecture Specification

Existing architecture formalisms [7, 10], are target toward specific architectural styles (pipe-filter, client-server, etc) and therefore contain a high level of implementation detail. However, when specifying systems at the architectural level, the goal is to capture the precise component behavior that is required to guarantee correct system-level behavior. In addition, there should be some traceability between the system and component level functionality. Reasoning efficiently about these relationship requires an architecture representation that abstracts out the operational details and provides a declarative specification of an architecture.

An *architecture theory* constrains the behavior of a system in terms of the behavior of its subcomponents via a collection of axioms. This relationship is specified declaratively, abstracting away implementation concerns. Formally, an architecture theory is the target theory of a parameterized theory. An overview of the role of architecture theories in system design is show in the parameter passing diagram of Figure 5. The system and component specifications in an architecture theory are extensions of the generic problem theory in Figure 3 allowing the system and subcomponent interface specifications to be "plugged in" to the architecture. The axioms of an architecture theory place constraints on the relationship between the system and component interfaces.

The problem theory parameters are instantiated with actual system and component specifications by specification morphisms from problem theory. Constructing these morphisms corresponds to specifying the problem and selecting components from a library. The result of instantiation is a specialized architecture theory. If the axioms of the architecture theory are valid with the problem and component substitutions, then the architecture can be used to correctly decompose the problem into the selected components.

An architecture theory is actualized as an architecture schema implemented in an *architecture description language* [10]. The correctness of the implemented system requires that the constraints placed on the system by the architecture theory are guaranteed by the schema. This can be verified based on a formal semantics of the target architecture description language [7, 10]. The schema is instantiated by constructing a morphism from the schema to an instantiated architecture by substituting the actual components into the architecture schema.




```
ExampleArchitecture : trait
includes
  ProblemTheory(X,Z,IS,OS),
  ProblemTheory(X,W,IA,OA),
  ProblemTheory(W,Z,IB,OB),
  ProblemTheory(X,Z,IC,OC)
asserts
  ∀ w:W, x:X, z:Z
  IS(x) ⇒ IA(x)∧ IC(x);
  IS(x)∧ OA(x,w)∧ OB(w,z)∧ OC(x,z)
    ⇒ OS(x,z)
```

Figure 6. Example Architecture Theory

An example architecture with its corresponding architecture theory is shown in Figure 6. It has four problem theory parameter specifications: 1 for the system and 3 for the components. Local renamings for the domain, range, precondition and postcondition of each parameter specification are provided[1]. The theory also contains axioms that place constraints on the problem and component specifications. The first axiom states that a legal input to the system is a legal input to the subcomponents to which it is passed. The second axiom states that, for a legal system input, if the intermediate and output signals in the system are constrained by the specified behavior of the components, then the output is a valid system output.

6. Component Adaptation

To support component adaptation we apply architecture theories in an incremental and constructive manner. This is done by using the architectural constraints as a guide while selecting components to plug in. As each component is selected, it further constrains any missing components. By maintaining the validity of the constraints with each component selection, we are guaranteed to construct a correct specification morphism. Specification-based component retrieval is used to assist this process.

[1]The input and output types were not specified so capitalized versions of variable names are used

```
FindProblem(D,R,I,O) : trait
includes
  Container(List,Rec),
  ProblemTheory(D,R,I,O)
  D tuple of l:List, k:Key
  R tuple of r:Rec
asserts
  ∀ pre:D, post:R
  I(pre) == true
  O(pre,post) == (post.r).key=pre.k
                 ∧ element(post.r,pre.l)
```

Figure 7. Interface Specification for `Find`.

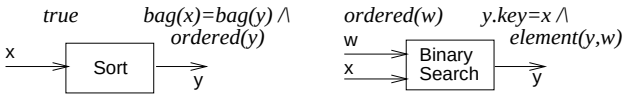


Figure 8. Sample Components

There are two ways in which architectures can be applied to component adaptation: *augmentation* and *subcomponent replacement*. In the first case, a component is treated as black box and its behavior is modified by placing it into an architecture with other components. In the second case, the component to be adapted is an architectural component. The behavior of the architecture can be modified by replacing one of its subcomponents.

6.1. Augmentation

In augmentation we modify a component's behavior by placing it into a simple architecture. For example, consider the problem specification for the `Find` component in Figure 7. A specification-based component retrieval mechanism would return the `Binary Search` component from Figure 8 as a possible candidate for reuse due to their identical output conditions. However, `Binary Search` does not satisfy `Find` because its behavior is not defined for all of the legal inputs to `Find`. Instead, the Weak Post match holds between the two specifications:

$$\text{Weak Post:}(I_{BS} \wedge O_{BS}) \Rightarrow O_F$$

This information can be used to guide component adaptation. Specifically, if we can adapt `Binary Search` to cover the rest of the legal inputs, then we will have a complete solution to the problem.

The component can be properly adapted by placing it in an architecture with another component that provides the missing functionality required to solve the problem. We have identified two adaptation tactics that will work, shown in Figure 9: add a component that (a) solves the problem

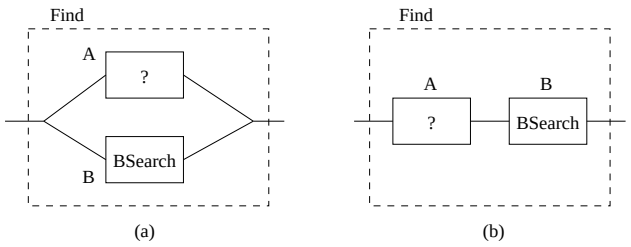


Figure 9. Adaptation Tactics for Weak Post

in the situations where `Binary Search` does not or (b) maps all legal problem inputs to legal `Binary Search` inputs in such a way that a valid solution is generated. We are currently investigating alternate architectures and heuristics for selecting adaptation tactics. Tradeoffs that must be considered while selecting an adaptation tactic may include performance requirements, limits on parallelism and constraints on the target language. For illustrative purposes, we will use the second method in this example.

By plugging `Find` and `Binary Search` into the proper architecture theory, we can generate a specification for the missing component. This process is shown in Figure 10. The renamings associated with a specification morphism are associated with the morphism arrows. The `Find Problem` specification is plugged in for the system specification in the `Arch Sequential` architecture theory. This theory defines a simple sequential architecture like that in Figure 9b. It contains axioms constraining the correctness of the architecture and the connection between components A and B. By plugging in the `Find Problem` specification, the system specification predicates are replaced resulting in the `Arch Find Sequential` theory. Next, the `Binary Search` component is plugged in for component `B`, further defining the constraints in the architecture `Arch Find BSearch Sequential`.

The only remaining undefined predicates are the predicates for the `A` component. Based on the axioms, we can derive definitions for the missing predicates. The missing precondition A_I can be derived from the first axiom:

$$\forall x : \langle List, Key \rangle \cdot true \Rightarrow A\_I(x)$$

This indicates that A_I must be $true$ to satisfy the architectural constraints. The missing postcondition A_O is constrained by the last two axioms:

$$\begin{aligned}\forall x : \langle List, Key \rangle, y : \langle List, Key \rangle, z : \langle Rec \rangle \\ A\_O(x, y) \wedge (z.r).key = y.k \wedge element(z.r, y.l) \\ \Rightarrow (z.r).key = x.k \wedge element(z.r, x.l) \\ A\_O(x, y) \Rightarrow ordered(y.a)\end{aligned}$$

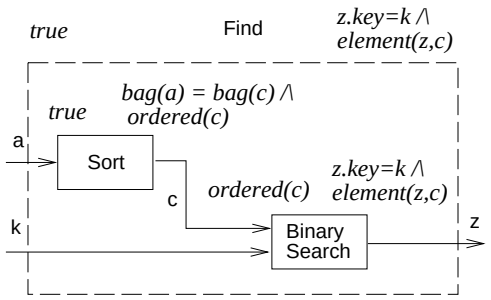


Figure 11. An Architecture for `Find`

A definition for A_O can be found by removing it from each axiom and deriving a $\{x, y\}$-antecedent [11] for each of the remaining statements. This basically entails rewriting and simplifying the equation in terms of x and y. For the first axiom this gives:

$$\begin{aligned}\forall x : \langle List, Key \rangle, y : \langle List, Key \rangle \\ y.k = x.k \wedge bag(y.l) = bag(x.l)\end{aligned}$$

where $bag(l)$ is the multi-set containing the elements in list l. For the second axiom we get:

$$\forall x : \langle List, Key \rangle, y : \langle List, Key \rangle \cdot ordered(y.a)$$

The antecedent that ensures both axioms are true is the conjunction of these two antecedents. Therefore, if A_O ensures this relationship, the architectural constraints will be met.

To find an appropriate component, the constraints are issued as a query specification to the component retrieval system. In this case, the `Sort` component is returned. This component can be placed in the architecture as shown in Figure 11. Because the sort component satisfies the constraints required by the architecture theory, the architecture is correct by construction.

6.2. Subcomponent Replacement

To adapt a component by subcomponent replacement, we assume that the component is implemented by an architecture. The behavior of the architecture can be modified by replacing one of its subcomponents. The mechanics of subcomponent replacement are the same as augmentation. However, the original system-level specification is replaced with the target problem specification. Then all of the component specifications are plugged in except for the one being replaced. Based on the architecture constraints, a specification for the new subcomponent is derived. Further investigation is underway to develop tactics for choosing which subcomponent to replace based on the specification matching results.

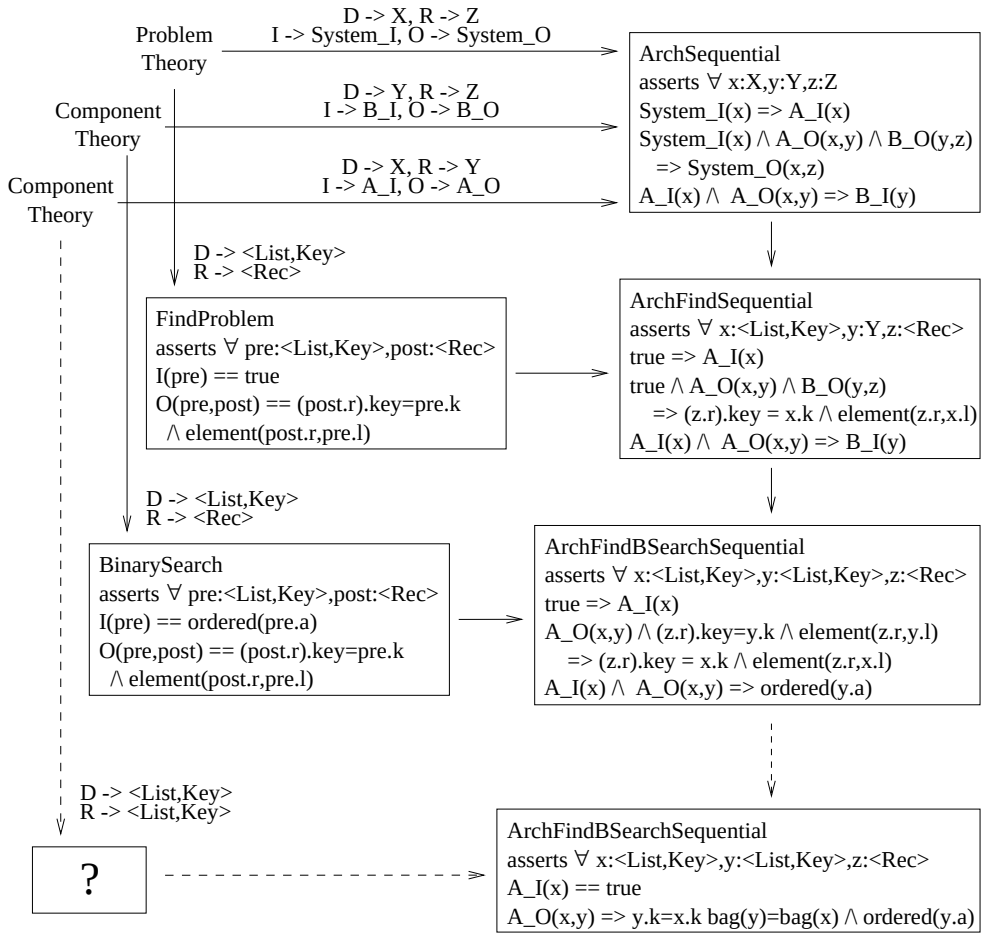


Figure 10. Component Adaptation Using an Architecture Theory

7. Related Work

Our work is an extension of the ideas used in Kestrel's Interactive Development System (KIDS) [12]. In KIDS, the structure of specific algorithms such as global search or divide and conquer are represented algorithm theories. Architecture theories generalize algorithm theories by specifying structure in terms of subcomponent problem theories rather than operators. This allows the construction of hierarchical systems and scales the KIDS methodology beyond the construction of single subroutines.

There is a large body of work dedicated to using formal specifications to aid component retrieval and reuse [2, 5, 6, 9, 16]. A theoretical foundation for this work was recently provided by Zaremski and Wing [16, 17] by defining the general activity of specification matching. Most of the approaches to specification-based component retrieval provide a mechanism to limit the application of specification matching by arranging the library in a hierarchy or a lattice based on specification generality. However, a major concern in all of these systems is scalability because they make extensive use of theorem proving during retrieval.

Many formalisms have recently been introduced to make software architecture a more rigorous activity. Most efforts [10] are targeted at formalizing specific architectural styles (pipe-filer, client-server, etc.) and not with the problem decomposition aspects of architecture. Therefore, these representations are too operational to represent the types of relationships that we are interested in. However, formal models of architectural styles do provide an important link between a general architecture theory and a specific architecture schema which implements the theory.

8. Conclusion

This paper describes a formal framework for component adaptation based on interface specifications and architec-

tures. This framework allows us to capture the relationship between what components are potentially reusable and how they can be properly adapted. Identification of reusable components is done using formal interface specifications and specification matching. Component adaptation is supported by using architecture theories to declaratively specify system structure. An example illustrated how architecture theories can be used constructively to guide component adaptation. Based the success of algorithm theories [12] and some initial studies such as those described in this paper, we believe that the flexibility and abstract nature of architecture theories will enable them to support component adaptation.

We are currently investigating control mechanisms for automating component adaptation using architecture theories. This involves finding adaptation strategies and heuristics for selecting adaptation strategies based on the available components and architectures. In the adaptation example, two adaptation tactics with associated architectures were given for the Weak-Post match. We believe that associating various tactics with the specification match hierarchy can provide a framework in which to selectively apply automated reasoning. We are investigating the matches and architectures that will work together to construct a complete solution to a problem specification.

9. Acknowledgments

We would like to thank Amy Moormann Zaremski, Karen Davis, Phillip Wilsey, Michael Lowry, and several anonymous referees for helpful suggestions. Support for this work was provided in part by the Advanced Research Projects Agency and monitored by Wright Labs under contract F33615-93-C-1316 and F33615-93-C-4304 and NASA Ames Research Center Contract NAS2-13605.

References

[1] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specifications 1: Equations and Initial Semantics*. EATCS Mongraphs on Theoretical Computer Science. Springer–Verlag, Berlin, 1985.

[2] B. Fischer, M. Kievernagel, and W. Struckmann. VCR: A VDM-based software component retrieval tool. In *Proc. ICSE-17 Workshop on Formal Methods Application in Software Engineering Practice*, 1995.

[3] D. Gries. *The Science of Programming*. Texts and Monographs in Computer Science. Springer-Verlag, New York, NY, 1981.

[4] John V. Guttag and James J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, New York, NY, 1993.

[5] Jun-Jang Jeng and Betty H. C. Cheng. A formal approach to using more general components. In *Proceedings of the 9th Knowledge-Based Software Engineering Conference*, pages 90–97, September 1994.

[6] A. Mili, R. Mili, and R.Mittermeir. Storing and retrieving software components: A refinement based system. In *Proc. 16th Int'l Conf. on Software Engineering*, pages 91–100, Sorrento, Italy, May 1994.

[7] Mark Moriconi, Xiaolei Qian, and Bob Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering*, 21(4):356–372, April 1995.

[8] John Penix, Phillip Baraona, and Perry Alexander. Classification and retrieval of reusable components using semantic features. In *Proceedings of the 10th Knowledge-Based Software Engineering Conference*, pages 131–138, November 1995.

[9] Dewayne E. Perry and Steven S. Popovitch. Inquire: Predicate-based use and reuse. In *Proceedings of the 8th Knowledge-Based Software Engineering Conference*, pages 144–151, September 1993.

[10] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[11] Douglas R. Smith. Top-Down Synthesis of Divide-and-Conquer Algorithms. *Artificial Intelligence*, 27(1):43–96, 1985.

[12] Douglas R. Smith. KIDS: A Semiautomatic Program Development System. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, 1990.

[13] Douglas R. Smith. Toward a classification approach to design. In *Proceedings of the Fifth International Conference on Algebraic Methodology and Software Technology, AMAST'96*, LCNS. Springer Verlag, 1996.

[14] Douglas R. Smith and Micheal R. Lowry. Algorithm Theories and Design Tactics. *Science of Computer Programming*, 14:305–321, 1990.

[15] I. Van Horebeek and J. Lewi. *Algebraic Specifications in Software Engineering: An Introduction*. Springer-Verlag, Berlin, 1989.

[16] Amy Moormann Zaremski. *Signature and Specification Matching*. PhD thesis, Carnegie Mellon University, January 1996.

[17] Amy Moormann Zaremski and Jeannette M. Wing. Specification matching of software components. In *3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, October 1995.