SPARTACAS - Automating Component Adaptation for Reuse

by

Brandon W. Morel

BSCoE, University of Kansas, 2001

Thesis submitted to the Department of Electrical Engineering and Computer Science and the Faculty of the Graduate School of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science.

Chairman, Dr. Perrry Alexander

Committee Member, Dr. Susan Gauch

Committee Memeber, Dr. Costas Tsatsoulis

Abstract

A mounting challenge for software designers is to find efficient and cost-effective implementations for large and complex software problems. Many see software reuse as an intuitive approach, however the cost of reuse tends to outweigh the potential benefits. The costs of software reuse include establishing and maintaining a library of reusable components, searching for applicable components to be reused in a design, as well as adapting components toward a solution. This thesis introduces SPARTACAS, a framework for automating specification-based component retrieval and adaptation. Using specifications, instead of implementations, allows automated theorem-provers to formally verify logical relationships between components and problems. Logical relationships are used to evaluate the feasibility of reusing the implementations of components to implement a problem. Retrieving a component that is a complete match to a problem is rare, it is more common to retrieve a component that partially satisfies the requirements of a problem. Such components have to be adapted. Rather than adapting components at the code level, SPARTACAS adapts the behavior of partial matches by imposing interactions with other components in an architecture. Based on the unsatisfied constraints of the problem, a sub-problem is synthesized that specifies the missing functionality required to complete the problem; the sub-problem is used to query the library for components to adapt the partial match. The framework was implemented and evaluated empirically, the results suggest that automated adaptation using architectures successfully promotes software reuse, and hierarchically organizes a solution to a design problem.

Acknowledgments

I regret that this section only returns modest acknowledgments to those who have contributed encouragement, ideas, and opinions to my research and study over the last two years.

I can not overstate my gratitude to my advisor, Dr. Perry Alexander, for his guidance, support, and sound advice. It has been an extraordinary experience to work for him on a project that provided me with such challenge, stress, intrigue, and reward. I also wish to thank Dr. Susan Gauch and Dr. Costas Tsatsoulis for participating on my thesis committee and their educational teachings on related issues.

I am indebted to the brilliant people who researched specification-based component retrieval before me, specifically John Penix and Bernd Fischer, for my achievements are a result of standing on their shoulders. Thanks to EDAptive Computing and NASA for their sponsorship and effort on the research project.

I owe a great deal to my many student colleagues for providing a environment in which to learn and grow. Thanks to the members and alumni of the Systems Level Design Group: Srinivas Akkipeddi, Satyanarayana Kakarlamudi, Garrin Kimmell, Ed Komp, Cindy Kong, David Schonberger, Jesse Stanley, Zhongjun Wang, Justin Ward, and Kalpesh Zinjuwadia for the enjoyable research experience. Thanks to Shyang Tan who offered little or no technical assistance, but was entertaining and made me laugh everyday. I am grateful to Adam Gossman who has been there for me since day one, and to Samata Pokharel who has taught me the most about myself. My special thanks to my biggest supporters, my cheer-leading section, my family. They have given me so much love and support to which I'm overwhelmingly thankful. I must also thank the professors, secretaries, librarians, staff, and students at the University of Kansas for providing a wonderful experience.

“Thou shalt study thy libraries and strive not to reinvent them without cause, that thy code may be short and readable and thy days pleasant and productive.“

- Henry Spencer

“Considering the current sad state of our computer programs, software development is clearly still a black art, and cannot yet be called an engineering discipline.“

- President Bill Clinton

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Reuse is an established practice and considered a sound engineering principle in many design fields. Some engineering disciplines, such as hardware engineering, have seen the proliferation of commercial-off-the-shelf components that can be used to successfully construct progressively complex systems. As software systems continue to increase in complexity and size, there is a greater demand to design and deploy these systems as safely and quickly as possible while keeping costs down. Reuse potentially offers many attractive benefits to the software design cycle, including the ability to: reduce errors early in system design, increase the productivity of software engineers, and increase the quality and reliability of the software produced. However, the benefits of software reuse must outweigh its costs for it to become widespread. These costs include the effort to create and maintain a library of reusable components, and the costs associated with retrieving, adapting, and integrating reusable components into an implementation to a design problem.

Software reuse [19, 35] requires an initial investment to collect a library of reusable software components. These components may come from a variety of sources such as individuals, organizations, and the World-Wide-Web. Software reuse can also be applied to most objects in the software life-cycle: concept designs, requirements, specifications, code, and even test plans. Several works have focused on solving the management, maintenance, organization, and representation of software libraries and repositories in which these objects are stored and retrieved [36, 28, 7].

In order for a component to be reused, the component has to be located. Once a library of components has been established, a user must spend the time browsing through the library until a component that matches, or closely approximates, the requirements of the problem. Even if the components in the library are somehow indexed, browsing through thousands of components is not feasible. It must be more efficient to locate a matching component than it is to design it. Automated component retrieval has long been an area of research, generating a range of retrieval systems and frameworks at various levels of software abstraction [27, 37, 12, 24, 49, 56].

Prior work has shown to considerable improve the efficiency in retrieving components that match a particular problem. While successful experiments are becoming more common, the practice of software reuse has been slow to realize the potentials claimed by software reuse advocates. Several works [20, 33] attribute the unfulfilled promise to various technical and non-technical reasons. While non-technical issues are unavoidable, one technical obstacle still remains to be resolved, namely automating the adaptation of components. One would not expect to always find a component that is a perfect match to a design problem. Generally there is a higher probability of finding a component that *almost* matches the object the user is searching for, rather than finding a complete match. These "partial matches" require modification or adaptation. Although automating adaptation has been an area of research in case-based reasoning and knowledge-based systems [58, 34, 52], few experiments have attempted to address the issue of software adaptation [46, 25].

Safely and correctly modifying software code is not a trivial task [38], in some instances adapting complex software code may outweigh the cost of producing the software from scratch. Penix [46, 45] proposed adapting software components using adaptation architectures. Adaptation architectures modify the behavior of a software component by imposing interactions with other components. The components in the architecture adjust the input and output values of the partial match such that for all legal inputs of the problem, the architecture generates the valid outputs. Not only do adaptation architectures increase the prospects of reuse, they also provide an organizational hierarchy in the implementation to the design problem.

1.2 Problem Statement

Most software reuse frameworks that have been developed retrieve components from a library that satisfy the constraints specified by a design problem. One can not always expect a library to contain a component that satisfies a large or complex problem. It is more feasible to expect that a component exists in the library that partially satisfies the constraints of a design problem. This work addresses this issue by adapting partially matching components to satisfy a design problem. To decrease the costs to the user, the methodology must be fully automated and ideally produce accurate and precise solutions in a timely manner. It also must be effective across multiple component libraries to be considered a general purpose approach.

1.3 Proposed Solution

Adapting software to meet the needs of a software programmer occurs in the present-day software engineering design cycle, however, the methods used are often based on ad-hoc techniques that have no formal basis. Using these methods has a profound affect on the correctness and security of the component and its role within a larger system. Current adaptation techniques depend on the software engineer's knowledge of the component and its overall interaction in a system, limiting its use.

We address the problem of adaptation in this work by developing a framework for reuse called SPecification-based Architecture and Retrieval Techniques for Automating Component Adaptation and Synthesis. SPARTACAS uses a layered component retrieval engine and formal architectural adaptation tactics for software component reuse. Based on the unsatisfied constraints of a partially matching component, the missing functionality required to solve the problem is synthesized in a sub-problem. Using formal methods allows for a rigorous technique to mathematically verify that the adaptation conforms to the requirements of the problem, thus preventing errors and ambiguities in the adaptation process.

SPARTACAS uses an automated theorem-prover to automate the retrieval of exact and partial matches. An adaptation architecture theory is a specification that formally specifies the interaction of sub-components and the relationship between the functionality of the sub-components and the functionality of the system. Using this relationship, the functionality of the system and the functionality of a partially matched component can be used to formally define a sub-problem. The sub-problem represents the required functionality to adapt the partially matched component. The sub-problem is automatically synthesized and used to re-search the library for components for adaptation. If a component is found for adaptation, the adaptation architecture theory is instantiated with the partial match and the adapting component, resulting in an architectural solution to the design problem.

The objective of this thesis is to define and evaluate the SPARTACAS reuse and adaptation framework. Components are represented at the specification-level using Rosetta [2]. Using specifications over implementation allows for a formal representation that allows automation of the retrieval, synthesis, and adaptation processes. The goal will be to show that the automated adaptation process in SPARTACAS leads to a reliable design using existing components, thereby reducing the time to implement and test a solution.

1.4 Thesis Outline

Chapter 2 gives a brief introduction on the background theory on formal specifications, component retrieval, component adaptation, and system architectures. The chapter motivates the use of formal specifications as a representation of software components. It outlines some of the automated retrieval techniques used at the specification level, such as feature-based, signature-based, and specification-based retrieval engines. The chapter goes on to describe architecture specifications and how they can be used for adaptation. Chapter 3 presents the SPARTACAS framework. Each module in the framework is introduced and the module's role within the reuse framework is detailed. Chapter 4 describes three adaptation architecture theories: sequential, parallel, and alternative. These theories are formally specified and their similarities and differences are described. To adapt the behavior of a partial match, the input and output values must be adjusted to satisfy the requirements of the problem. A partially matching component and components to adapt its behavior are interconnected using port connections. The methods for port connection and its impact on retrieval is the theme of chapter 5. Chapter 6 elaborates on the adaptation of components toward a solution to a problem. Specifically, the tactics used and the definitions for sub-problem synthesis will be presented. The chapter shows that the behavior of the problem being solved is properly maintained during each adaptation step, implying that the architectural solution generated is correct by composition. Two adaptation examples are explained in chapter 7 and the SPARTACAS framework is evaluated in chapter 8. The chapter draws on empirical results to show that the framework increases the prospects of reuse. Chapter 9 summarizes the future work and limitations of the current framework. Related work is compared in chapter 10, followed by a summary of the results and contributions of this work in chapter 11.

Chapter 2

Background

2.1 Software Component Representation for Reuse

Numerous reuse frameworks have been developed at almost every level of software design, ranging from the high-level requirements level [24] to the low-level execution level [27]. Frakes and Gandel [18] have outlined some of the important issues in selecting a reuse representation. The representation should be consistent in the way all designers interpret, create, and understand a software design component in the reuse framework. The representation is also dependent on the s-cope (i.e. domain-specific, level-specific) of the reuse framework. Issues, such as the expressiveness, granularity, integrity, complexity, and stability of the representation also needs to be considered.

Although each abstraction level certainly has its advantages, using formal specifications over code to represent software allows a designer to precisely model, verify, and analyze software components. A formal specification [61] states the behavior of a component without stating the implementation details. The mathematical foundations of formal specifications are beneficial when designing complex systems. Formal specification tools, e.g. theorem-provers and model-checkers, can be used to detect errors and verify properties of high-level systems quickly and efficiently.

Modeling software components at the specification level offers greater integrity, stability, consistency, and understand-ability. Formal specifications also offers greater flexibility in the complexity, expressiveness, and granularity of the systems that can be represented.

2.2 Formal Specifications

Each component is described by a formal specification [61] that states the behavior of a component without stating the implementation details. Using formal specifications over implementations allow automated theorem-provers to verify match conditions between two components. The formal component specifications use a simple axiomatic structure [26, 29, 57]:

$$\forall d \in D, \exists r \in R | I(d) \Rightarrow O(d, r)$$

D and R are the domain and range respectively. The domain represents the input values to the component and the range represents the output values of the component. I is a set of pre-conditions that define the *legal inputs* to the component. The pre-conditions constrain the domain to the values that have a defined output. O is a set of post-conditions that define the *feasible outputs* for each legal input based on a D×R relation. If the pre-conditions hold then the component will end in a state such that the post-conditions are true. If the pre-conditions do not hold then there are no guarantees that the post-conditions will hold, however, termination is assumed in all cases.

Component specifications are written in Rosetta [1], a systems level design language for modeling heterogeneous systems. A Rosetta *facet* describes the requirements or behavior of a particular aspect of a system or component. Facet parameters declare the domain (*input* typed variable declarations) and range (*output* typed variable declarations) of the component. A facet operates in a declared domain, which defines the vocabulary of semantics available to the facet. Facet term labels starting with *pre* and *post* are used to define the pre- and post-conditions over the domain and range respectively. Facet term labels that start with *arch* will be used to define structural component specifications and structural solutions to problems. The structure of the specifications in Rosetta is shown in figure 2.1.

```
package componentName() :: domainName is
  export all;
begin

  facet componentName(parameterList) :: domainName is
     export all;
  begin
     termLabel: term;
     ...
  end facet componentName
end package componentName
```

Figure 2.1: Specification structure in Rosetta

2.3 Component Retrieval

A number of works have applied formal specifications to automated reuse [9, 31, 32, 13, 53, 47] with attractive results. Formal specifications allow formal verification tools to verify logical relationships between specifications. The relationships are used to determine the degree to which a component can be reused to implement the problem.

2.3.1 Feature-based Retrieval

One of the simplest and most common approaches to component retrieval is to *classify* a component by assigning it keywords. Components are retrieved for problem queries using a Boolean relationship between the keywords of the component and the keywords of the problem. The components that are retrieved are in the same "class" as the problem. The feature-based retrieval scheme in figure 2.2 represents the framework to automate the classification and retrieval of specifications using *domain-specific features*. The framework is analogous to classifying components with keywords, and retrieval based on keyword similarity between a component and the problem query.

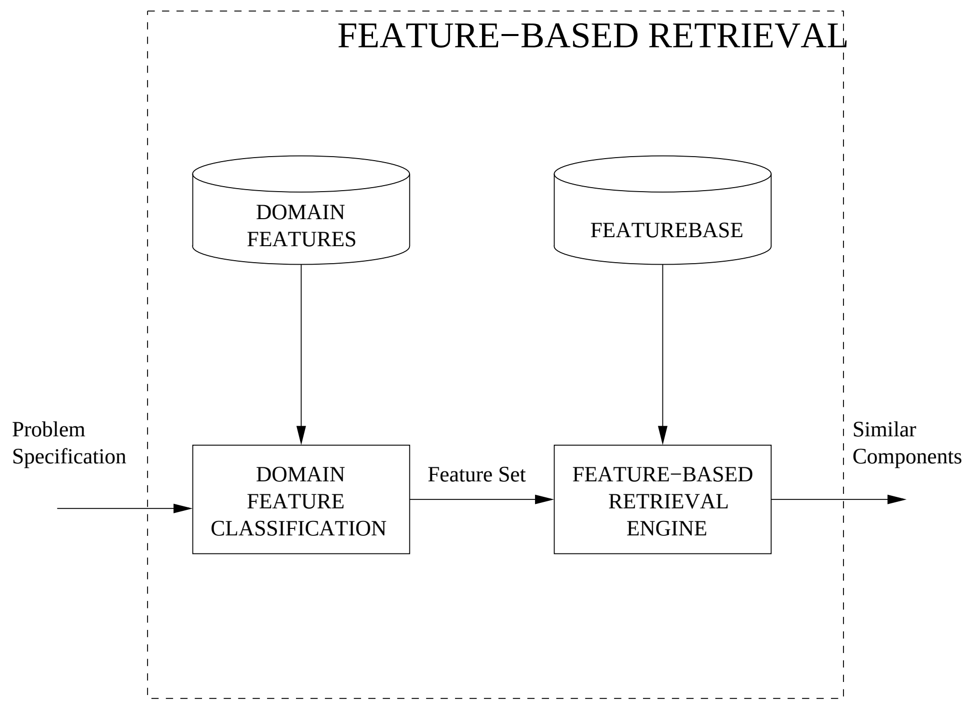


Figure 2.2: Feature-based retrieval framework

Feature Classification

The feature-based retrieval framework contains a collection of domain-dependent features [48], which are a set of theorems or predicates that capture some characteristic or trait within a specific domain. The following is an example of a feature that filters elements from a list:

$$FILTER(list, element) \equiv \exists x, y : list | \forall z : element | (z \in y \Rightarrow z \in x) \wedge (x \in D) \wedge (y \in R)$$

A specification is assigned a domain feature if the feature can be logically derived from the specification. Although a domain expert is required to develop and specify the set of domain-specific feature definitions, a theorem-prover can be used to automate the process of classifying [23, 50, 48] components using domain-specific features. A *feature set*, see definition 2.3.1, is assigned to a component in the library and stored in a database called a *featurebase*. Classification of component specifications is performed off-line, only classification of problem specifications is perform during retrieval.

Definition 2.3.1 *A feature set FS is a collection of features f_i with predicates ϕ_i that can be assigned to a component C*

$$FS(C) = \{f_i || I_C \wedge O_C \Rightarrow \phi_i\}$$

Feature-based Filter

A feature-based retrieval engine [48] filters out all of the components that do not have the same feature classification. Components that are retrieved through feature-based matching is based on a similarity threshold. Similarity between a component C and problem P is based on the number of features they have in common:

$$Similarity_{feature}\ \alpha\ (C, P) = \frac{size(C_{featureset} \cap P_{featureset})}{size(C_{featureset} \cup P_{featureset})}$$

Once the library of components has been classified with features, the benefit of feature matching is quick and efficient retrieval of components to problems with the same classification. Feature matching is a necessary, but not sufficient, match condition: matching features are necessary for a component to satisfy a problem, but the components retrieved are not guaranteed to satisfy the problem. For instance, a problem may specify low-pass filtering of digital signals, which would be classified as a FILTER. The problem query may retrieve a component that performs high-pass filtering of digital signals. The component has the same classification, but does not completely satisfy the problem. Feature-based matching is generally used to restrict the search of a large library of components to a few components that belong in the same class.

2.3.2 Signature-based Retrieval

A signature represents the collection of domain and range types of a unit of software; it does not contain any semantic information. A signature-based retrieval engine [62] filters out components that do not have compatible signatures. The signature-matching process is simply described as type matching, where the input and output types of a component must match the input and output types of a problem respectively. A type is defined as is either a type variable in the set of simple types or a type constructor over the simple types. The generic signature matching is described in definition 2.3.3.

Definition 2.3.2 *Type equality is defined as $\tau =_\tau \tau'$ iff they are lexically identical simple type variables or for type constructors* tc *and* tc', $\tau = tc(\tau_1, ..., \tau_n)$, $\tau' = tc'(\tau'_1, ..., \tau'_n)$, $tc = tc'$, *and* $\forall 1 \geq i \geq n$, $\tau_i = \tau\tau'_i$

Definition 2.3.3 *Signature match condition M over the component signature of types τ_C and a problem signature of types τ_C is defined as $M(\tau_C, \tau_P) = \exists$ a transformation function T and match relation R such that $T(\tau_C)$ R $T(\tau_P)$*

The transformation functions include a variety of operations over the sequence of types, such as reordering, substitution, and currying [62]. The relation R can include equality or relaxed matches, such as generalization or specialization [62].

2.3.3 Specification-based Retrieval

A specification-based retrieval engine performs specification matching [31]. The set of pre-conditions (I) and the set of post-conditions (O) are used to verify that a logical relationship holds using the semantics of a component (C) and problem (P). The specification matching condition for a component to satisfy a problem is given in the following two conditions:

$$\forall d : D_P \parallel I_P(d) \Rightarrow I_C(d)$$

$$\forall d : D_P,\ r : R_P \parallel I_P(d) \land O_C(d, r) \Rightarrow O_P(d, r)$$

The first condition states that any legal input to the problem must be a legal input to the component. The component specification (i.e. $I_C \Rightarrow O_C$) assures that, given a legal input, an output will be produced. The second condition states that all the feasible outputs of the component for legal problem inputs are valid outputs of the problem. In case of an illegal input, the behavior of the component is unpredictable. Given a problem specification and a component specification, reuse can be demonstrated by proving that the two satisfaction conditions hold. Often the components retrieved are not exact matches and they need to be adapted to satisfy the problem. Zaremski and Wing [64] established a number of match conditions (sometimes referred to as the *degree of a match* or *degree of satisfaction*) for assessing specification reuse. Figure 2.3 shows a portion of these match conditions.

If a component C formally *Satisfies* a problem P, then the implementation of C can be reused to implement P. C satisfies P if C accepts all legal inputs to P, and the valid outputs of C are valid outputs of P when given legal inputs. *Weak Plug-in* and *Plug-in* are stronger match conditions of Satisfies. The *Plug-in Pre* match condition implies that a component meets only the pre-condition requirements. The *Plug-in Post* and *Weak Post* match conditions imply that a component meets the post-condition requirements, but operates in a more restrictive environment. Plug-in Pre, Plug-in Post, and Weak Post are referred to as *partial match conditions.*

Several specification-based retrieval engines [15, 48] have been developed using automated theorem-provers to logically verify that a component specification matches a problem specifica-

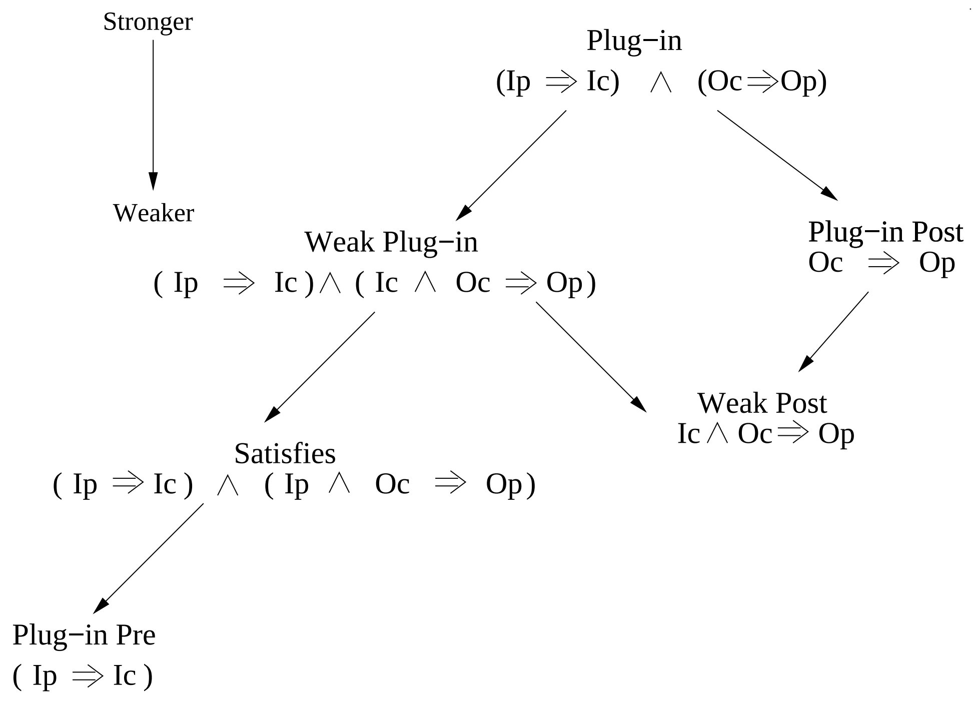


Figure 2.3: Specification match lattice

tion. Although theorem proving is sound and precise, it can not be practically applied when using a large library of components. Typically, a less formal syntactic matching process is used to reduce the number of components used during specification matching.

2.4 Architecture Specification

An *architecture theory* is a collection of axioms that specify the relationship between the behavior of a system and the behavior of the interconnected sub-components. An architecture theory is a parameterized theory specification that can be instantiated with component and system constraints. Figure 2.4 shows the architecture theory instantiated with constraints from system and component specifications. The result is an architecture morphism [46] that allows the problem to be decomposed into a system of sub-components.

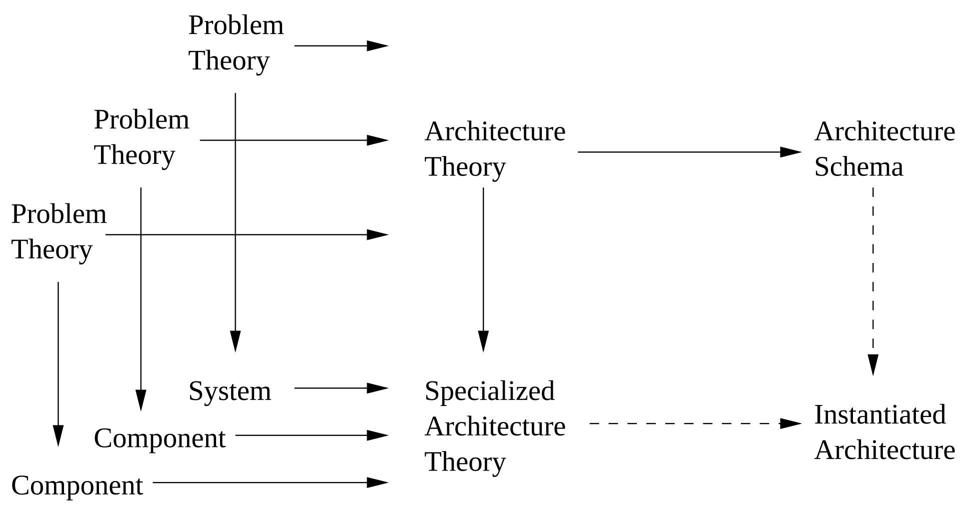


Figure 2.4: Architecture theory instantiation

Chapter 3

Component Reuse Framework

The primary goal of SPARTACAS is to retrieve solutions to a design problem from a library of components at the specification-level. Several specification-based retrieval frameworks [14, 15, 48, 9] have already been developed with success. Although these frameworks efficiently retrieve components that are exact or partial matches to a problem query, few deal with adapting partial matches to completely satisfy the problem. The SPARTACAS framework, shown in figure 3.1, differs from other frameworks by including an automated adaptation capability.

In the SPARTACAS framework, a formal specification that specifies a design problem is used to query the retrieval engine. The retrieval engine returns component specifications that are either total and partial matches from a library of components (the component library contains a collection of existing component specifications that have been created, tested, and shown to correctly specify the component's functionality). For partial matches, the adaptation engine selects a tactic for adapting the component. The tactic synthesizes sub-problems that specifies the missing functionality required to solve the rest of the problem. The sub-problem is used to search for components to adapt the behavior of the partially matching component. The components are instantiated in an architecture. The potential architectural solutions are verified and added to the component library, further increasing the potential of reuse.

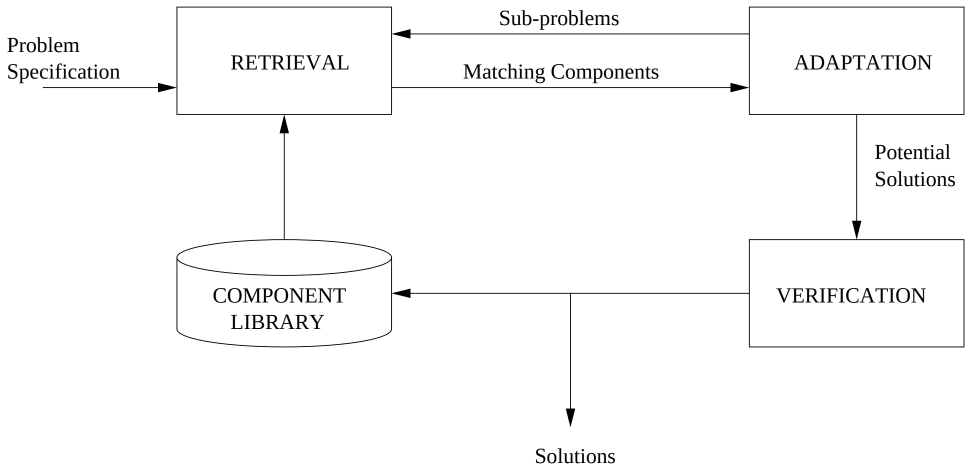


Figure 3.1: General component reuse framework

3.1 Component Retrieval Framework

The retrieval framework, figure 3.2, uses a layered architecture of retrieval engines, where each layer progressively filters out irrelevant components. The components that pass through all the filters have the highest probability of matching the problem. This layered approach to retrieval is similar to work by Fischer [13].

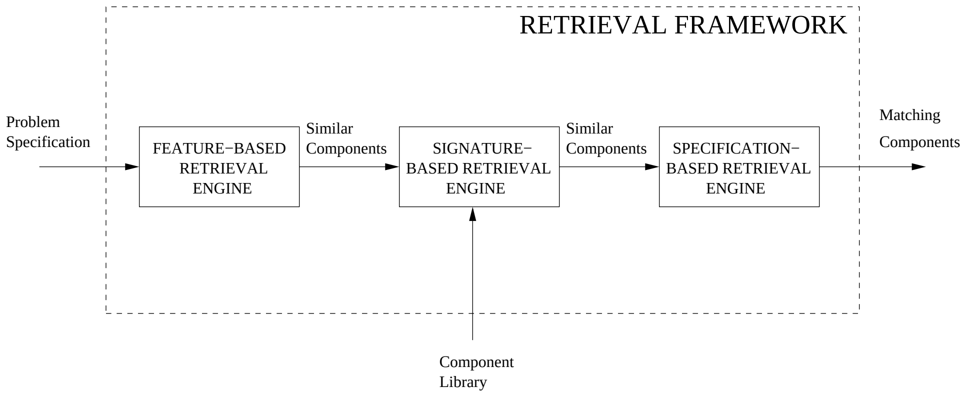


Figure 3.2: Component retrieval framework

In the first layer, a feature-based retrieval engine classifies the problem specification by assigning it domain-specific features. The feature-based retrieval engine retrieves components that have similar features, thereby filtering out components that are not in the same class as the problem. The component search space is reduced by the feature-based retrieval engine and given to the signature-based retrieval engine. In the second layer, the signature-based retrieval engine filters out components that do not have compatible signatures. A signature is the collection of input and output variable declarations declared in the specification's interface. Components that do have compatible (exact or relaxed mappings from component to problem ports) signatures can be instantiated to possibly solve the problem. The information used by the signature-based retrieval engine is also used in the instantiation of components in an architecture. The last layer is the specification-based retrieval engine, which performs specification matching using an automated theorem-prover to logically verify that a component specification matches a problem specification. Figure 2.3 shows a portion of these match conditions, proposed by Zaremski and Wing [64], that are used to evaluate reuse.

Proving logical relationships for the entire library using an a theorem-prover is not feasible since formal verification is a time consuming task [44]. The feature-based and signature-based retrieval engines perform fast and efficient matching of feature keywords and signatures respectively, therefore filtering out, with reasonable certainty, components that do not match the problem. The layered architecture approach to retrieval allows a subset of promising components to pass to the next level of computationally intensive retrieval. The layered approach to retrieval using feature-based and specification-based retrieval engines has been implemented in other works [46, 42].

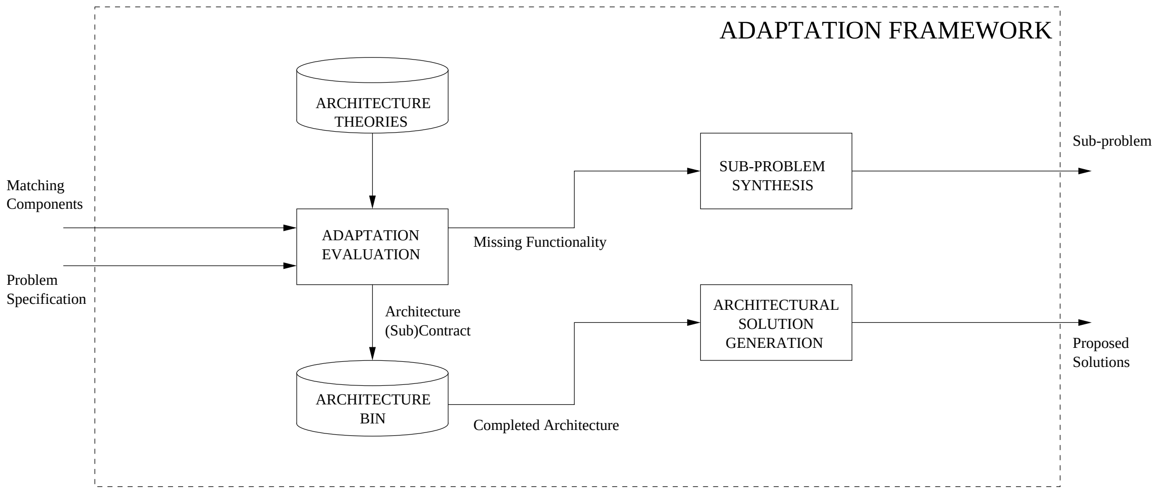


Figure 3.3: Component adaptation framework

3.2 Component Adaptation Framework

It is naive to assume that a component will exist in a library that satisfies a large and complex problem. It is more feasible to retrieve a component that has a subset of the properties of the problem. The behavior of the component can be adapted to obtain the properties of the problem by placing the component in an architecture with other components, where in this work an architecture is simply defined as a collection of interconnected components.

The adaptation framework, shown in figure 3.3, contains a collection of adaptation architecture theories. They specify the constraints on the interconnection of sub-components and the behavioral relationship between the sub-components and the overall system. Given a partially matching component to a problem, the *adaptation evaluation* module determines which adaptation architecture should be applied. It is possible that several adaptation architectures are applicable. The adaptation tactic and the component is added to the *architecture bin* as a "contract". The architecture bin acts as a tree of architectural blueprints, which is used to plan the execution of contracts toward a solution.

The missing functionality required to adapt a partial match is synthesized into a sub-problem specification. The synthesizer generates a sub-problem specification that is re-submitted to the retrieval engine. Complete matches to the sub-problems may not exist in the library. If a component is a partial match to a sub-problem, then the adaptation process can be repeated. Adaptation required for sub-problems results in sub-architectures within architectures. A component that completely satisfies a problem (or sub-problem) signifies that an architectural contract has been completed. Adaptation continues until (1) an architecture of interconnected components satisfying the problem has been constructed, (2) a solution to the design problem is known not to exist given the current component library, or (3) a solution to the design problem can not be realized using the heuristic limitations of the retrieval and adaptation process, e.g. a ceiling on the number of components used in an architecture.

3.2.1 Verification Framework

As is the case with physical architectures, inconsistencies between the constructed product and the conceptual blueprints may exist. In a design of a critical system it is important that flaws are identified and removed from the product. Ideally, the causes of inconsistencies or flaws should be avoided during construction. Figure 3.4 shows the framework for verifying and validating architectural solutions to a problem. Given a possible solution to the problem, the domain features are assigned to a solution, which are compared to the features of the problem specification. The *specification-based verification* module proves that the solution logically satisfies the problem. Verification of an architecture requires a considerable amount of computation, increasing the time to retrieve a solution set. If component adaptation is performed correctly, then verification can be avoided.

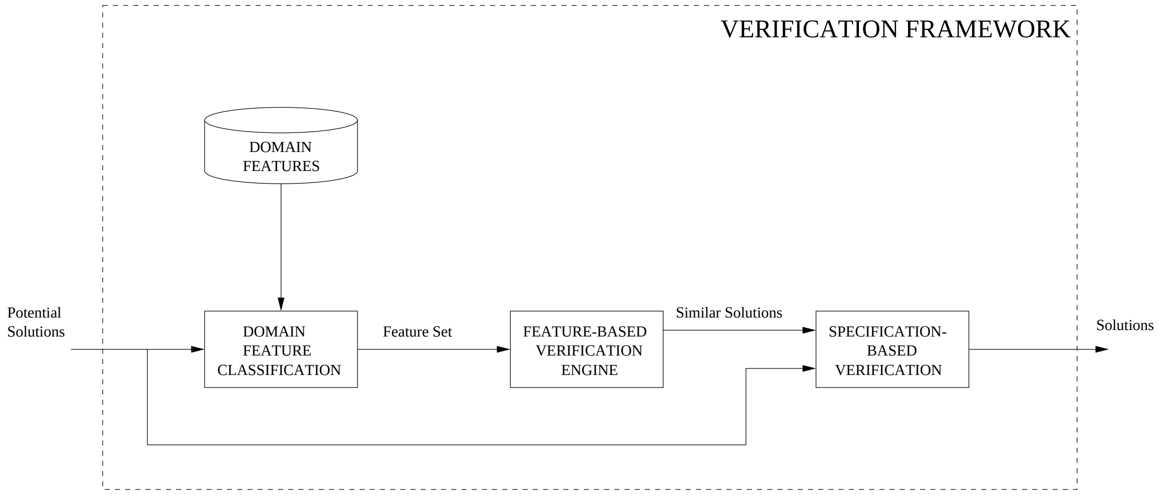

VERIFICATION FRAMEWORK
DOMAIN FEATURES
Potential Solutions
DOMAIN FEATURE CLASSIFICATION
Feature Set
FEATURE-BASED VERIFICATION ENGINE
Similar Solutions
SPECIFICATION-BASED VERIFICATION
Solutions

Figure 3.4: Component verification framework

Chapter 4

Adaptation Architecture Theories

Black-box reuse [55] involves reusing a software component without modifying the internal implementation, however, modification of the component interface may be required for reuse. Components in black-box reuse can typically be reused "as is" or adapted using a simple interface wrapper [51] to modify its interface (i.e. reordering the parameters). *White-box reuse* [55] involves reusing a software component after modification of the internal implementation to meet the requirements of a problem. Performing white-box reuse safely and correctly can be very difficult, but can be applied to solve many problems. Black-box reuse is often easy to perform, yet limited in the problems that can be solved. *Behavioral adaptation* [44] is the process of altering the functionality of a component by imposing interactions with other components in an architecture. The components in the architecture adjust the input and output values of the component to be adapted, resulting in valid outputs for all legal inputs of the problem. The components used in the architecture can be used "as is" or modified with interface wrappers.

An adaptation architecture theory [44] is a formal specification of an architecture that specifies the interaction and configuration of sub-components in the composition of a system, as well as the relationship between the functionality of the sub-components and the functionality of the system.

There are several advantages to using formal specifications to represent adaptation architectures: formal specifications specify the abstract relationships between sub-components without specifying the implementation details of the architecture, they allow for a precise definition of the component adaptation process, and the architecture solutions generated from the architecture theories will be in a representation that is consistent with components in the library, which can be added to the component library.

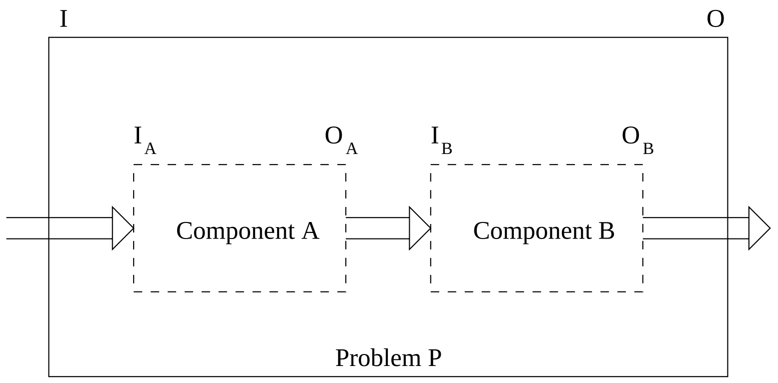
Architecture	*Match Condition*	*Instantiation*
Sequential	Plug-in Post Weak Post	Plug component into tail position, derive sub-problem to satisfy head position
	Plug-in Pre	Plug into head position, derive sub-problem to satisfy tail position
Alternative	Weak Post	Plug into either position, derive sub-problem to satisfy missing functionality
Parallel	N/A	Problem decomposition, independent instantiation

Table 4.1: Available adaptation architecture tactics

Penix [47] proposed three adaptation architecture theories: sequential, alternative, and parallel. One or more of these adaptation architectures can be applied to adapt the behavior of a partially matched component. The degree to which a component satisfies a problem determines which adaptation architecture tactic can be applied. Table 4.1 shows the adaptation tactics associated with each match condition. The three adaptation architectures can be used to compose increasingly complex architecture structures. A component declared in an adaptation architecture theory specification can abstractly represent other architectures. Moreover, components in the library can also represent user-defined or domain-specific architectures which can be retrieved and instantiated within other architectures.

4.1 Sequential Architecture

The sequential architecture, figure 4.1, is the interconnection of two components where the output of one component is the input to another component through some type of communication medium. The head component adapts the input values to the tail component such that the tail component generates feasible outputs. Conversely, the tail component adapts results generated by the head component for all legal inputs.



Sequential Architecture Theory
BEGIN

// Problem and component definitions
Problem(D, R, I, O)
$\text{Component}_A(D_A, R_A, I_A, O_A)$
$\text{Component}_B(D_B, R_B, I_B, O_B)$

// Domain and range constraints
drConstraint1: $D \subseteq D_A$
drConstraint2: $R_A \subseteq D_B$
drConstraint3: $R_B \subseteq R$

// Pre and post-condition constraints
behConstraint1: $\forall\, d : D \mid I(d) \Rightarrow I_A(d)$
behConstraint2: $\forall\, d : D,\, x : D_B \mid I(d) \wedge O_A(d, x) \Rightarrow I_B(x)$
behConstraint3: $\forall\, d : D,\, y : R_A,\, r : R \mid I(d) \wedge O_A(d, y) \wedge O_B(y, r) \Rightarrow O(d, r)$

END Sequential Architecture Theory

Figure 4.1: Sequential architecture theory

Double arrows represent the collection of interconnected ports. A *port* is an input or output typed variable that is constrained by the domain or range. During retrieval, the signature-matching engine attempts to find components with matching signatures. The signature is used to connect ports of a component to the ports of the problem or other components. Input ports of a component can be connected to the input ports of a problem or to output ports of another component, while output ports of a component can be connected to the output ports of a problem or to input ports of another component. Figure 4.1 also specifies constraints over the types of interconnected ports. The bottom of figure 4.1 lists several constraints that specify the relationships between the functionality of the problem and the functionalities of the sub-components.

4.2 Alternative Architecture

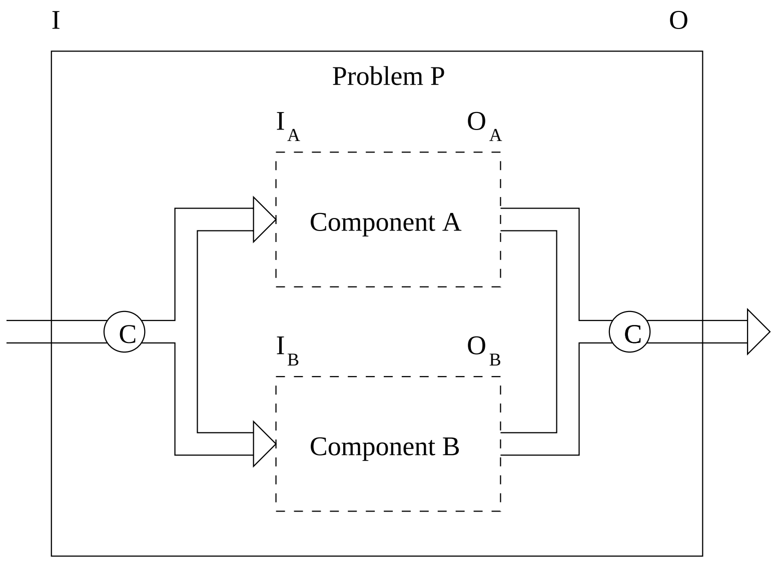
The alternative architecture is the interconnection of two independent components that work simultaneously whose outputs are combined to satisfy the problem requirements (shown in figure 4.2). In this architecture, one component satisfies the problem for some subset of legal input values to the problem. This component is adapted with another component which correctly covers the rest of the input values.

This architecture requires a control structure to achieve correct cooperation to reach the solution to a problem. Without a control structure, the two components could possibly diverge on a given input and drive contradictory results on the output. A forward control structure routes inputs to the appropriate component based on some control function, forcing one and only one component to drive the output. Although two variables drive the problem output port, the forward control structure acts as a resolution function since, for any give input, the variable the drives the the output port can be determined. The reverse control structure demultiplexes the two outputs values of the components to drive a single output based on the same control function as the forward control structure. The alternative architecture may have one or both control structures.

4.3 Parallel Architecture

The parallel architecture, figure 4.3, is similar to the alternative architecture. Independent components work simultaneously and their outputs collectively satisfy the problem requirements. The difference is that the output values generated by the components in the parallel architecture are not combined to affect a single output. The components in the parallel architecture compute on disjoint sub-ranges, which collectively form the range of the problem. Each component computes results for some (not necessarily disjoint) sub-domain of the problem.

Figure 4.3 uses the || notation to represent the composition of some disjoint sub-ranges into the range. For instance, a problem may have the following output ports: {x::integer, y::boolean, z::real}. The range could possibly be represented as: {y::boolean}||{x::integer, z::real}. The parallel adaptation architecture decomposes a problem into independent sub-problems. Components that satisfy these sub-problems never interact with each other, they merely solve an isolated aspect of the problem.



Alternative Architecture Theory
BEGIN

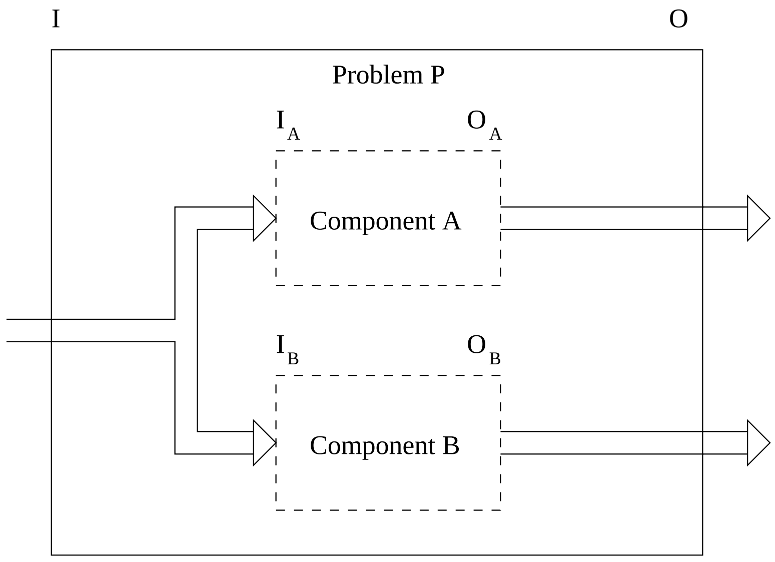
// Problem and component definitions
Problem(D, R, I, O)
Component$_A$(D$_A$, R$_A$, I$_A$, O$_A$)
Component$_B$(D$_B$, R$_B$, I$_B$, O$_B$)

// Domain and range constraints
drConstraint1: D $\subseteq$ D$_A$
drConstraint2: D $\subseteq$ D$_B$
drConstraint3: R$_A$ $\subseteq$ R
drConstraint4: R$_B$ $\subseteq$ R

// Pre and post-condition constraints
behConstraint1: $\forall d : D|(I(d) \Rightarrow I_A(d)) \vee (I(d) \Rightarrow I_B(d))$
behConstraint2: $\forall d : D, r : R|(I_A(d) \wedge O_A(d,r) \Rightarrow O(d,r)) \vee (I_B(d) \wedge O_B(d,r) \Rightarrow O(d,r))$

END Alternative Architecture Theory

Figure 4.2: Alternative architecture theory



Parallel Architecture Theory
BEGIN

// Problem and component definitions
Problem(D, R, I, O)
Component$_A$(D$_A$, R$_A$, I$_A$, O$_A$)
Component$_B$(D$_B$, R$_B$, I$_B$, O$_B$)

// Domain and range constraints
drConstraint1: D $\subseteq$ D$_A$ $\cup$ D$_B$
drConstraint2: R$_A$ || R$_B$ $\subseteq$ R

// Pre and post-condition constraints
behConstraint1: $\forall d_1 \cup d_2 : D | (I(d_1 \cup d_2) \Rightarrow I_A(d_1) \wedge I_B(d_2))$
behConstraint2: $\forall d_1 \cup d_2 : D, r_1 || r_2 : R | I(d_1 \cup d_2) \wedge O_A(d_1, r_1) \wedge O_B(d_2, r_2) \Rightarrow O(d_1 \cup d_2, r_1 || r_2)$

END Parallel Architecture Theory

Figure 4.3: Parallel architecture theory

Chapter 5

Port Connection Methods

Component and problem specifications define a domain and range through port declarations. A port is an input/output typed variable that is constrained in the pre- and post-conditions. Much of the early work in software retrieval dealt with signature matching [62], where a signature is comprised of input parameters and return types of functions, procedures, and other such software artifacts. In specification-based matching, the problem interface acts as a foundation: the input and output ports are defined to interface with the world. Components are constructed within the foundation to implement the functionality required by the problem. The ports of a component are connected with the ports of a problem, which requires that the component and problem have compatible input and output ports. The instantiation of ports, or port connection, is given in definition 5.0.1. Operations such as currying, generalization, and specialization [62] of types can be applied to find a proper mapping.

Definition 5.0.1 *A port connection ρ is a function mapping the ports of component $C(D_C, R_C, I_C, O_C)$ to the ports of problem $P(D, R, I, O)$ such that:*

- $\forall$ *input port* $c_i : T_c \in \mathbf{D_C} | \exists$ *input port* $p_i : T_p \in \mathbf{D} | (\rho(c_i) \rightarrow p_i) \wedge (T_p \subseteq T_c)$

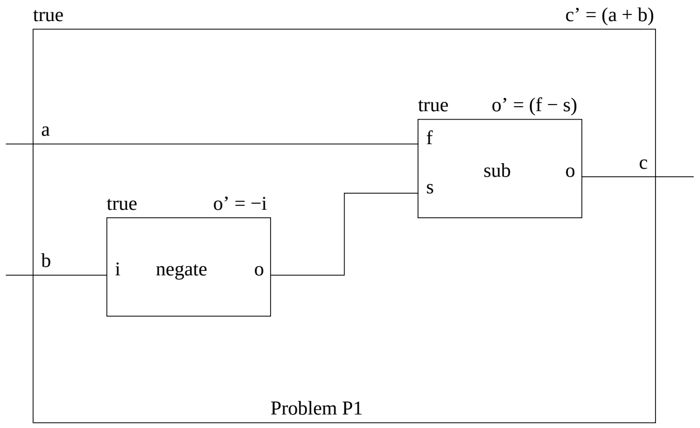
- $\forall$ *output port* $c_o : T_c \in \mathbf{R_C} | \exists$ *output port* $p_o : T_p \in \mathbf{R} | (\rho(c_o) \rightarrow p_o) \wedge (T_c \subseteq T_p)$

Bijective port connection retrieves components to problem for which there is a one-to-one and onto mapping from input ports of the component to the input ports of the problem, and similarly a mapping of component output ports to problem output ports. In traditional signature matching, the component and problem must have an equal number of compatible input ports and output ports. This can hinder potential applications for adaptation. Figure 5.1 specifies a problem to perform simple addition on two real numbers. Figure 5.2 shows a successfully instantiated solution using the simple mathematical operations library in figure 6.3. Following retrieval, it is determined that the numerical subtraction (*sub*) component is a (bijective) Plug-in Pre partial match to the problem. The numerical negation (*negate*) component is not retrieved since it does not have the proper number of ports for instantiation. Clearly a solution can not be found using bijective signature matching for even the simplest problems, thus motivating the need for less restrictive port connection methods.

```
// Simple addition problem
package P1() :: null is
  export all;
begin

  facet P1(a :: input real; b :: output real;
           c :: output real) :: state_based_semantics is
    export all;
  begin

    pre:   true;
    post:  c' = (a + b);

  end facet P1;
end package P1;
```

Figure 5.1: Problem specification of a simple addition problem

A less restrictive port connection method is the *one-to-one port connection*. The one-to-one port connection requires that all component input ports be driven by one and only one problem input



```
// Sequential architecture solution to problem P1
problem P1() :: null is
  export all;
begin

  use negate;
  use sub;
  facet P1(a :: input real; b :: input real;
           c :: output real) ::
    state_based_semantics is
    export all;
    x__0 :: M__Type(sub.s);
  begin

    arch0: negate(b, x__0);
    arch1: sub(a, x__0, c);

    pre:   true;
    post:  c’ = (a + b);

  end facet P1;
end package P1;
```

Figure 5.2: Solution to problem P1 using the sequential architecture

port, and all component output ports drive one and only one problem output port. This allows potential components to have fewer number of input/output ports than the problem. However, since not all the ports of the problem can be instantiated, components that are retrieved will not completely satisfy the problem (assuming the uninstantiated ports have meaningful constraints on them). A sub-problem is required to search for other components to instantiate and satisfy the functionality of the unconnected ports.

The *onto port connection* method simply requires that, for all ports in the component, there is a connection to some problem port with respect to port direction. Potentially a single problem input port can drive multiple component input ports, and a single problem output port can be driven by multiple component output ports (in such a case the final output value needs to be resolved). This allows the component to have more ports than the problem.

5.1 Connection Trade-offs

Using a less restrictive port connection method to increase recall may result in expensive overheads. Table 5.1 shows the maximum number of signature instantiations per component for each of the port connection methods. The number of instantiations per component may drown the specification-based retrieval engine from making timely progress.

Port Connection Method	*Maximum Combinations*
Bijective	$(\sum ports_{C_i})! * (\sum ports_{C_o})!$
One-to-one	$\frac{(\sum ports_{P_i})!}{(\sum ports_{C_i})! * ((\sum ports_{P_i}) - (\sum ports_{C_i}))!} * \frac{(\sum ports_{P_o})!}{(\sum ports_{C_o})! * ((\sum ports_{P_o}) - (\sum ports_{C_o}))!}$
Onto	$(\sum ports_{C_i})^{(\sum ports_{C_i})} * (\sum ports_{C_o})^{(\sum ports_{C_o})}$

Table 5.1: Maximum instantiation combinations per port connection method

It can also be argued that feature-matching is not a necessary condition if the signature-matching engine uses a less restrictive port connection method. Assume the domain features for the math library consists of: *ADDITION*, *SUBTRACTION*, and *NEGATION* over real numbers. In such an example, the problem would clearly be classified with the following feature set: {ADDITION}. However, none of the components in the library can be classified with the ADDITION feature, therefore none of the components would be retrieved when queried with Problem P1. As a result, the threshold of the feature-based retrieval engine may have to be relaxed in order to find a solution. Relaxing the retrieval filters, in order to increase recall, increases the probability of resorting to a complete (and often very expensive) search of the component library.

Chapter 6

Automated Component Adaptation

Using the system/sub-component functionality relationships specified in an adaptation architecture theory, the functionality required to adapt a partially matching component toward a solution can be derived and synthesized into a sub-problem. The adaptation theory used is contingent on the match condition between the problem and the partially matched component. The sub-problem represents the missing functionality required to fulfill the problem. The sub-problem is used to re-search the library to identify applicable components for adaptation (if an exact match can not be found, the process repeats resulting in sub-architectures). The partially matched component, and the component (sub-architecture) to adapt its behavior, are instantiated in an architecture. Once an architecture has been constructed, the potential solution still needs to be verified that it correctly solves the problem. However, if the sub-problem constraints are strictly and properly maintained during each step of adaptation, then a correct solution to the problem is guaranteed by composition.

6.1 Sequential Adaptation

Recall from table 4.1 that Plug-in Post, Weak Post, and Plug-in Pre matched components can be adapted using the sequential adaptation architecture. The missing functionality synthesized in a sub-problem for Plug-in Post and Weak Post adaptation is referred to as *post-match driven synthesis* since the post-conditions of the problem have been met. *Pre-match driven synthesis* refers to synthesizing sub-problems for adaptation using Plug-in Pre matched components since the pre-conditions have been met.

6.1.1 Post-match Driven Synthesis

A Weak Post or Plug-in Post matched component is abstractly represented by Component$_B$ in the sequential architecture in figure 4.1. A sub-problem specification must be synthesized in order to find a component, i.e. Component$_A$, to adapt Component$_B$. Component$_A$ must change the environment to allow Component$_B$ to execute and satisfy the behavior of the problem for all legal inputs. Using the relationship between the functionality of the system and the functionalities of the sub-components in the sequential architecture, the missing functionality required to adapt a partial match can be derived by instantiating the architecture theory with the problem as the system and the partial match as one of the sub-components. The unknown sub-component pre- and post-conditions can be solved in terms of the system and partial match pre- and post-conditions. The sub-problem synthesis is specified in definition 6.1.1.

Definition 6.1.1 *Given a problem P(D, R, I, O) and a Weak Post/Plug-in Post matched component B(D_B, R_B, I_B, O_B), the synthesized sub-problem for the missing functionality in the sequential architecture is:*

- *Domain: D*

- *Range:* $D_B \cup \{r \in R | \neg\exists x \in R_B | \rho(x) \rightarrow r\}$
- *Pre-conditions:* $\forall d : D | I(d)$
- *Post-conditions:* $\forall d : D, x : D_B, y : \{r \in R | \exists x \in R_B | \rho(x) \rightarrow r\}, r : R|$

  $I_B(x) \wedge (\neg O_B(x, y) \vee O(d, r))$

In figure 6.1, the behavioral relationships specified in the sequential architecture theory are inferred from the match conditions. The inference starts from the Weak-Post (Plug-in Post can also be inferred) match condition between the partially matched component and the problem, and the Satisfies match condition between the component for adaptation and the synthesized sub-problem. In step 1, the conjunction is split for simplification. The synthesized pre- and post-conditions are replaced with definition 6.1.1 in step 2, and the conjunction is split in step 3. The first behavioral constraint of the sequential architecture follows immediately in step 4. The conjunction in the consequent is split in step 5, and the second behavioral constraint is satisfied in step 6. Lastly, the negated term is moved to the antecedent in step 7, which leads to the third behavioral constraint.

$$\dfrac{I_B \wedge O_B \Rightarrow O_P \qquad \dfrac{\dfrac{\dfrac{behConstraint1}{I_P \Rightarrow I_A}\,(4) \qquad \dfrac{\dfrac{behConstraint2}{I_P \wedge O_A \Rightarrow I_B}\,(6) \qquad \dfrac{\dfrac{behConstraint3}{I_P \wedge O_A \wedge O_B \Rightarrow O_P}\,(8)}{I_P \wedge O_A \Rightarrow \neg O_B \vee O_P}\,(7)}{I_P \wedge O_A \Rightarrow I_B \wedge (\neg O_B \vee O_P)}\,(5)}{(I_P \Rightarrow I_A) \wedge (I_P \wedge O_A \Rightarrow I_B \wedge (\neg O_B \vee O_P))}\,(3)}{(I_{synth} \Rightarrow I_A) \wedge (I_{synth} \wedge O_A \Rightarrow O_{synth})}\,(2)}{(I_B \wedge O_B \Rightarrow O_P) \wedge ((I_{synth} \Rightarrow I_A) \wedge (I_{synth} \wedge O_A \Rightarrow O_{synth}))}\,(1)$$

Figure 6.1: Post-match driven sequential synthesis inference tree

For illustration purposes, the simple math problem in figure 6.2 is used to query the small component library of simple math functions in figure 6.3. The library also shows the degree of match between a component and a problem as well as the port connections that were used to obtain the match condition. The library shows that the *pInc* component is a Weak Post match to problem *P2* using a bijective port connection. The synthesized sub-problem using definition 6.1.1 is specified in figure 6.4. The range of the sub-problem consists of the input types to the component as well as any output types of the problem that were not instantiated during signature matching. Since all output ports of the problem were instantiated in this example, the range of the sub-problem reduces to the input types of the component.

```
// Simple math problem
package P2() :: null is
  export all;
begin

  facet P2(x :: input real;
           z :: output real) :: state_based_semantics is
    export all;
  begin

    pre:   true;
    post:  if (x < 0)
           then (z' = ((-1 * x) + 1))
           else (z' = (x + 1)) end if;

  end facet P2;
end package P2;
```

Figure 6.2: Problem specification of a simple math problem

The sub-problem in this example thus specifies:

$$\forall d : D, d_B : D_B, r : R | I(d) \Rightarrow I_B(x) \wedge (\neg O_B(x, r) \vee O(d, r))$$

or more specifically:

$$x : real, a : real, z : real | (true \Rightarrow (a' >= 0)) \wedge (\neg(z = (a' + 1)) \vee$$

$$(if(x < 0)\ then(z' = ((-1 * x) + 1))\ else(z' = (a + 1))\ endif))$$

The variables `x` and `a` are the domain and range of the synthesized sub-problem respectively, however the variable `z` is being referenced. This variable represents a quantified variable over the pre- and post-conditions. In order to avoid variable name conflicts, the variable names have been mapped to unique names, i.e. *P2.x : real* $\rightarrow$ *P3.i__0 : real*, *pInc.a : real* $\rightarrow$ *P3.o__0 : real*, and *P2.z : real* $\rightarrow$ *P3.q__0 : real*. The *absVal* component is the only component that completely satisfies the requirements of sub-problem *P3*. Figure 6.12 shows a solution to problem P2 using the sequential architecture.

Pre-match Driven Synthesis

A component retrieved using the Plug-in Pre match condition can be abstractly represented as Component$_A$ in the sequential architecture in figure 4.1. A sub-problem specification has to be synthesized in order to find components, i.e. Component$_B$, to adapt Component$_A$. Component$_B$ must change the results of Component$_A$ to satisfy the behavior of the problem in all cases. The sub-problem synthesis is defined in definition 6.1.2. The behavioral relationships specified in the sequential architecture theory are inferred from the match conditions in figure 6.5.

Definition 6.1.2 *Given a problem P(D, R, I, O) and a Plug-in Pre matched component A(D_A, R_A, I_A, O_A), the synthesized sub-problem for the missing functionality in the sequential architecture is:*

- *Domain:* $R_A \cup \{d \in D | \neg \exists x \in D_A | \rho(x) \rightarrow d\}$
- *Range:* R
- *Pre-conditions:* $\forall d : \{d \in D | \exists x \in D_A | \rho(x) \rightarrow d\}, x : R_A | I(d) \wedge O_A(d, x)$
- *Post-conditions:* $\forall d : D, r : R | O(d, r)$

Consider starting with the absVal component to solve problem P2. The synthesized sub-problem using constraints from this component and problem P2 using the definition in 6.1.2 is specified in figure 6.6. The pInc component is retrieved to generate the same sequential adaptation architecture solution in figure 6.12.

6.2 Alternative Adaptation

In the alternative architecture theory, Component$_A$ can perform the same functionality as the problem, but only on a subset of the legal inputs. The problem would be solved if another component, namely Component$_B$, performs the same functionality but covers the rest of the legal inputs. Since the pInc component satisfies problem P2 when the input is positive, the alternative adaptation tactic can be applied to find a component (or another architecture) that satisfies the rest of the problem when the input is (at least) not positive. The synthesis is defined in definition 6.2.1. Using this definition, the synthesized sub-problem specifying the missing functionality is shown in figure 6.7.

Definition 6.2.1 *Given a problem P(D, R, I, O) and a Weak Post/Plug-in Post matched component A(D_A, R_A, I_A, O_A), the synthesized sub-problem for the missing functionality in the alternative architecture is:*

- *Domain: D*
- *Range: R*
- *Pre-conditions:* $\forall d : D | I(d) \wedge \neg I_A(d)$
- *Post-conditions:* $\forall d : D, r : R | O(d, r)$

The behavioral constraints are inferred from the match conditions in figure 6.8. In step 1, the synthesized pre- and post-conditions are replaced with definition 6.2.1. Given that $I_P \wedge \neg I_A \Rightarrow I_B$, the equation is rewritten in step 2. Step 3 involves the joining of the antecedents of sequents with O_P in the consequence. By splitting the conjunction in step 4, the behavioral constraints become apparent.

The solution to the sub-problem in figure 6.7 requires another adaptation architecture. It is clear to see that a sequential architecture using the negate and pInc components provides a solution. The limitation of the alternative architecture is that a control sub-problem also needs to be synthesized. Assuming two components are found for the alternative architecture, both components will drive the output of the problem, therefore a control mechanism needs to select which component will drive the output in the appropriate situation. For instance, pInc will generate an output even when the input is not positive, driving a nonsensical value on the output of the problem. The synthesis process for the forward and reverse control structures are shown in definitions 6.2.2 and 6.2.3. The control structures define mappings from input ports to output ports based on some control function.

Definition 6.2.2 *Given a problem P(D, R, I, O) and a Weak Post/Plug-in Post matched component A(D_A, R_A, I_A, O_A) which is adapted with component B(D_B, R_B, I_B, O_B), the synthesized forward control structure sub-problem for the alternative architecture is:*

- *Domain:* D
- *Range:* $D_A \cup D_B$
- *Pre-conditions: true*
- *Post-conditions:* $\forall d : D | (I_A(d) \Rightarrow \{\forall x : D | \exists y : D_A | \rho(x) \rightarrow y\}) \wedge (\neg I_A(d) \Rightarrow \{\forall x : D | \exists y : D_B | \rho(x) \rightarrow y\})$

Definition 6.2.3 *Given a problem P(D, R, I, O) and a Weak Post/Plug-in Post matched component A(D_A, R_A, I_A, O_A) which is adapted with component B(D_B, R_B, I_B, O_B), the synthesized reverse control structure sub-problem for the alternative architecture is:*

- *Domain:* $R_A \cup R_B \cup D$
- *Range:* R
- *Pre-conditions: true*

- *Post-conditions:* $\forall d : D | (I_A(d) \Rightarrow \{\forall x : R_A | \exists y : R | \rho(x) \rightarrow y\}) \wedge (\neg I_A(d) \Rightarrow \{\forall x : R_B | \exists y : R | \rho(x) \rightarrow y\})$

Expecting control structures to exist in the library is a significant drawback to the alternative architecture. Frequently it is sufficient to synthesize the language's control structures for an architectural solution since the control structures are simple (i.e. if-then-else statement). The solution to problem P2 using the alternative architecture is shown in figure 6.12.

6.3 Parallel Adaptation

In the previous adaptation architectures, a partially matching component was retrieved and a sub-problem was synthesized to search for other components to satisfy the missing properties. The sub-problem is dynamically defined based on the partially matching component. This type of behavioral adaptation is referred to as *bottom-up behavioral adaptation*. In *top-down behavioral adaptation*, the design problem is decomposed into sub-problems and each sub-problem is used to search for components for adaptation. In this type of adaptation (albeit loosely defined as adaptation), all possible decompositions must be applied until components in the library are found to match some combination of sub-problems.

In the parallel architecture, the goal is to adapt components that compute values on independent output variables. The specification-based match lattice describes match conditions over the set of range variables and sets of pre- and post-conditions, therefore bottom-up behavioral adaptation is not suited for the parallel adaptation tactic. Moreover, using the theorem-prover to prove a logical relationship between components using some subset of range variables and behavioral relationships is a very expensive and tedious task. Rather, SPARTACAS uses the top-down approach to parallel adaptation. Using specification slicing, a problem can be decomposed into independent sub-problems quickly and efficiently.

Program slicing [59] is a decomposition process used to isolate a subset of program behavior. A program slice is a sub-program that contains only those statements and variables that affect or are affected by a slicing criterion. A slice criterion is a set of variables that are of interest at some point in the program. Program slicing has generated a breadth of applications at the implementation level of software design, including debugging [60], maintenance [21], and reuse [8].

Program slicing is applied at the specification level by Oda and Araki [41]. They define a technique for slicing Z specifications. A slice contains a portion of the statements in the specification that constrain the value of a variable. Specification slicing is applied to the specifications written in Rosetta. The goal will be to use specification slicing to decompose a problem specification by isolating the independent behaviors; a retrieval engine will be used to locate components that satisfy the slices.

If a term `p` potentially affects term `q`, then term `q` is data dependent on `p`. Post-condition terms are data dependent on other post-conditions if they both constrain a range variable. Similarly, pre-condition terms are data dependent on other pre-conditions if they both constrain a domain variable. The functions for data dependency are defined:

$$dDepend(q,p:O):bool = \exists r \in R | constrains(q,r) \wedge constrains(p,r)$$

$$dDepend(q,p:I):bool = \exists d \in D | constrains(q,d) \wedge constrains(p,d)$$

If a term `p` potentially determines if term `q` is applied, then term `q` is control dependent on `p`. Pre-conditions control the application of the post-conditions, therefore a post-condition is control dependent on a pre-condition if the pre-condition constrains the legal inputs required to compute the feasible outputs. The function for control dependency is defined:

$$cDepend(o:O,i:I):bool = \exists d \in D | constrains(i,d) \wedge requires(o,d)$$

A specification slice is represented by a tuple (D_s, R_s, I_s, O_s). The slice criterion is a set of range variables. A specification slice is computed with the algorithm in figure 6.10. Initially, the post-conditions that affect or are affected by the criterion are assigned to O_s, then all post-conditions that are data dependent on post-conditions in O_s are added in steps 1 and 2. Next all the pre-conditions that potentially determine if the post-conditions in O_s are applied are added to I_s in step 3. Then all pre-conditions that affect or are affected by the pre-conditions in I_s are added to I_s in steps 4 and 5. Finally, the domain and range variables involved in O_s and I_s are selected in steps 6 through 9.

R_s represents all the range variables that affect or are affected by the criterion. If the specification was re-sliced with $criterion' \leftarrow R_s$, the same slice would be obtained. A slice may contain the original specification or an empty specification. An empty specification has no pre-conditions or post-conditions, i.e. pre: true and post: true.

Definition 6.3.1 *A "criterion partition" is the disjoint subsets of the range variables such that*

1) the union of the subsets equals the range

2) a subset is not empty

3) a subset does not affect or is not affected by another subset, meaning the subsets are all independent

A criterion partition is the same as a traditional partition of a set if all the range variables (R) are independent. A criterion partition generates a slice partition.

Definition 6.3.2 *A "slice partition" is the partition of slices generated by a criterion partition. A slice partition must also be disjoint since a criterion partition is disjoint, meaning each slice is independent of all other slices in the set (one slice does not influence and is not influenced by another slice).*

A criterion may contain several range variables. If the variables are all interdependent, then the criterion is referred to as the smallest criterion.

Definition 6.3.3 *The "smallest criterion partition" contains the disjoint subsets of smallest criterion.*

In terms of the *Stirling number of the second kind* [11], the smallest criterion partition corresponds to S(n, n), where S(n, k) states the number of partitions of an n-set into k blocks. Here "n" is the number of disjoint smallest criterion.

The algorithm in figure 6.11 is used to generate the smallest criterion partition. C_F represents the range variables. A single range variable is selected from C_F and is used as a slicing criterion. The slicing algorithm is used to generate R_s, which represents the smallest criterion that generates that particular slice. By removing R_s from C_F and repeating, the smallest criterion partition, C_{scp}, can be obtained. The algorithm is also used to generate the smallest slice partition, S_{ssp}.

Definition 6.3.4 *The "smallest slice partition" contains the disjoint subsets of smallest specification slices generated from the smallest criterion partition.*

Each slice in the smallest slice partition represents the smallest independent behavior of the specification, referred to as the smallest independent slice.

Definition 6.3.5 *The "smallest independent slice" is a specification slice that can not be further sliced. The criteria variables for generating the smallest independent slice are interdependent.*

Each slice specification in a partition is submitted to a retrieval engine. If a matching component is found for each slice, then the partition represents the collection of components to connect in parallel. Based on set union and composition, it is intuitive to infer the properties of the parallel architecture theory from the Satisfies match conditions between slices and matching components.

Component specifications in the library are not guaranteed to be in the smallest independent form, thus the smallest slice partition does not guarantee a complete solution. Therefore every possible slice partition must be generated (generated by every possible combination of the smallest criterion). The number of partitions of an n-set follow the *Bell numbers,* b_n [5, 54], which increase exponentially. If none of the slice partitions generate a complete solution, then there does not exist a parallel composition architecture to solve the problem.

It is clear to see the problem specification in figure 6.9 can be decomposed into the following independent sub-problems: ({a, b}, {x}, {pre}, {post1}), ({a, b}, {y}, {pre}, {post2}), and ({a, b}, {z}, {pre}, {post3}). Figure 6.13 shows one of many possible solutions to the problem.

6.4 Partial Connection Adaptation

As described in section 5, the bijective port connection can be too restrictive. To see the potential benefits of the one-to-one port connection method, it is applied to the solution search of problem P1 in figure 5.1. Following retrieval, the numerical negation component (one-to-one Plug-in Pre) is retrieved. Note the 1-1 port mappings from the negate component to problem P1: $[(\rho_{1-1}(negate.i) \rightarrow P1.b)(\rho_{1-1}(negate.o) \rightarrow P1.c)]$). Using the sequential adaptation architecture synthesis and the partially matched negate component, the sub-problem to solve the missing functionality is derived, see figure 6.14. Synthesis definition 6.1.2 states that the domain of the synthesized sub-problem contains output ports from the partially matched component (i.e. negate.o, renamed as P7.i_0) and the collection of problem input ports that are not connected to any of the input ports of the partially matched component (i.e. P1.a, renamed as P7.i_1). The sub-problem query results in locating the subtraction component for adaptation (see figure 5.2).

```
// Positive constrained increment
package pInc() :: null is
  export all;
begin
  facet pInc(a :: input real;
           b :: output real) :: state_based_semantics is
    export all;
  begin
    pre:  a >= 0;
    post: b’ = (a + 1);
  end facet pInc;
end package pInc;

(Bijective) Match on P1: (nil)
(1−1) Match on P1: (nil)
(Bijective) Match on P2: (Weak Post [(a −> x)(b −> z)])
(Bijective) Match on P3: (nil)
(Bijective) Match on P4: (Satisfies [(a −> i__0)(b −> o__0)])
(Bijective) Match on P5: (Weak Post [(a −> i__0)(b −> o__0)])
(Bijective) Match on P6: (nil)
(Bijective) Match on P7: (nil)
```

```
// Greater than component
package gt() :: null is
  export all;
begin
  facet gt(m :: input real; n :: input real;
             o :: output boolean) :: state_based_semantics is
    export all;
  begin
    pre:  true;
    post: o’ = (m > n);
  end facet gt;
end package gt;

(Bijective) Match on P1: (nil)
(1−1) Match on P1: (nil)
(Bijective) Match on P2: (nil)
(Bijective) Match on P3: (nil)
(Bijective) Match on P4: (nil)
(Bijective) Match on P5: (nil)
(Bijective) Match on P6: (nil)
(Bijective) Match on P7: (nil)
```

```
// Real number subtraction component
package sub() :: null is
  export all;
begin
  facet sub(f :: input real;
          s :: input real;
          o :: output real) :: state_based_semantics is
    export all;
  begin
    pre:  true;
    post: o’ = (f − s);
  end facet sub;
end package sub;

(Bijective) Match on P1: (Plug−in Pre [(f−>a)(s−>b)(o−>c)])
(1−1) Match on P1: (Plug−in Pre [(f−>a)(s−>b)(o−>c)])
(Bijective) Match on P2: (nil)
(Bijective) Match on P3: (nil)
(Bijective) Match on P4: (nil)
(Bijective) Match on P5: (nil)
(Bijective) Match on P6: (nil)
(Bijective) Match on P7: (Satisfies [(f−>i__1)(s−>i__0)(o−>o__0)])
```

```
// Great than equal to component
package geq() :: null is
  export all;
begin
  facet geq(m :: input real; n :: input real;
             o :: output real) :: state_based_semantics is
    export all;
  begin
    pre:   true;
    post: o’ = (m >= n);
  end facet geq;
end package geq;

(Bijective) Match on P1: (nil)
(1−1) Match on P1: (nil)
(Bijective) Match on P2: (nil)
(Bijective) Match on P3: (nil)
(Bijective) Match on P4: (nil)
(Bijective) Match on P5: (nil)
(Bijective) Match on P6: (nil)
(Bijective) Match on P7: (nil)
```

```
// Absolute value component
package absVal() :: null is
  export all;
begin
  facet absVal(i :: input real;
             o :: output real) :: state_based_semantics is
    export all;
  begin
    pre:  true;
    post: o’ = abs(i);
  end facet absVal;
end package absVal;

(Bijective) Match on P1: (nil)
(1−1) Match on P1: (Plug−in Pre [(i −> b)(o −> c)])
(Bijective) Match on P2: (Plug−in Pre [(i −> x)(o −> z)])
(Bijective) Match on P3: (Satisfies [(i −> i__0)(o −> o__0)])
(Bijective) Match on P4: (nil)
(Bijective) Match on P5: (nil)
(Bijective) Match on P6: (nil)
(Bijective) Match on P7: (nil)
```

```
// Numerical negation component
package negate() :: null is
  export all;
begin
  facet negate(i :: input real;
             o :: output real) :: state_based_semantics is
    export all;
  begin
    pre:  true;
    post: o’ = −i;
  end facet negate;
end package negate;

(Bijective) Match on P1: (nil)
(1−1) Match on P1: (Plug−in Pre [(i −> b)(o −> c)])
(Bijective) Match on P2: (Plug−in Pre [(i −> x)(o −> z)])
(Bijective) Match on P3: (Plug−in Pre [(i −> i__0)(o −> o__0)])
(Bijective) Match on P4: (nil)
(Bijective) Match on P5: (nil)
(Bijective) Match on P6: (nil)
(Bijective) Match on P7: (nil)
```

Figure 6.3: Small library of math components

```
// Post-match driven synthesis to problem P2 using
// component pInc and the sequential architecture
package P3() :: null is
  export all;
begin

  facet P3(i__0 :: input real;
           o__0 :: output real) :: state_based_semantics is
    export all;
    q__0 :: real;
  begin

  pre:  true;
  post: (o__0' >= 0) and
        ((not(q__0 = (o__0' + 1))) or
         (if (i__0 < 0)
          then (q__0' = ((-1 * i__0) + 1))
          else (q__0' = (i__0 + 1)) end if));

  end facet P3;
end package P3;
```

Figure 6.4: Post-match driven synthesis to problem P2 using component pInc and the sequential architecture

$$\dfrac{\dfrac{behConstraint1}{I_P \Rightarrow I_A}\,(2) \qquad \dfrac{\dfrac{\dfrac{behConstraint2}{I_P \wedge O_A \Rightarrow I_B}\,(5) \quad \dfrac{behConstraint3}{I_P \wedge O_A \wedge O_B \Rightarrow O_P}\,(6)}{(I_P \wedge O_A \Rightarrow I_B) \wedge (I_P \wedge O_A \wedge O_B \Rightarrow O_P)}\,(4)}{(I_{synth} \Rightarrow I_B) \wedge (I_{synth} \wedge O_B \Rightarrow O_{synth})}\,(3)}{(I_P \Rightarrow I_A) \wedge ((I_{synth} \Rightarrow I_B) \wedge (I_{synth} \wedge O_B \Rightarrow O_{synth}))}\,(1)$$

Figure 6.5: Pre-match driven sequential synthesis inference tree

```
// Pre-match driven synthesis to problem P2 using
// component absVal and the sequential architecture
package P4() :: null is
  export all;
begin

  facet P4(i__0 :: input real;
           o__0 :: output real) :: state_based_semantics is
    export all;
    q__0 :: real;
  begin

    pre:  true and (i__0 = abs(q__0));
    post: if (q__0 < 0)
          then (o__0' = ((-1 * q__0) + 1))
          else (o__0' = (i + q__0)) end if;

  end facet P4;
end package P4;
```

Figure 6.6: Pre-match driven synthesis to problem P2 using component absVal and the sequential architecture

```
// Synthesis to problem P2 and component pInc using the
// alternative architecture
package P5() :: null is
  export all;
begin

  facet P5(i__0 :: input real; o__0 :: output real)
    :: state_based_semantics is
    export all;
  begin

    pre:  true and not(i__0 >= 0);
    post: if (i__0 < 0)
          then (o__0' = ((-1 * i__0) + 1))
          else (o__0' = (i__0 + 1)) end if;

  end facet P5;
end package P5;
```

Figure 6.7: Synthesis to problem P2 and component pInc using the alternative architecture

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{behConstraint2}{(I_A \wedge O_A) \vee (I_B \wedge O_B) \Rightarrow O_P}\,(5) \quad \dfrac{\dfrac{behConstraint1}{I_P \Rightarrow I_B \vee I_A}\,(7)}{I_P \wedge \neg I_A \Rightarrow I_B}\,(6)}{((I_A \wedge O_A) \vee (I_B \wedge O_B) \Rightarrow O_P) \wedge (I_P \wedge \neg I_A \Rightarrow I_B)}\,(4)}{(I_A \wedge O_A \Rightarrow O_P) \wedge (I_P \wedge \neg I_A \Rightarrow I_B) \wedge (I_B \wedge O_B \Rightarrow O_P)}\,(3)}{(I_A \wedge O_A \Rightarrow O_P) \wedge ((I_P \wedge \neg I_A \Rightarrow I_B) \wedge (I_P \wedge \neg I_A \wedge O_B \Rightarrow O_P))}\,(2)}{(I_A \wedge O_A \Rightarrow O_P) \wedge ((I_{synth} \Rightarrow I_B) \wedge (I_{synth} \wedge O_B \Rightarrow O_{synth}))}\,(1)$$

Figure 6.8: Alternative synthesis inference tree

```
// Comparison block math problem
package P6() :: null is
  export all;
begin

  facet P6(a :: input real;
           b :: input real;
           x :: output real;
           y :: output real;
           z :: output real) :: state_based_semantics is
    export all;
  begin

    pre:   true;
    post1: x' = (a > b);
    post2: y' = (a = b);
    post3: z' = (a < b);

  end facet P6;

end package P6;
```

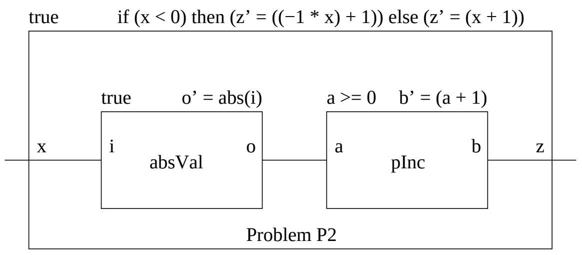
Figure 6.9: Problem specification of comparison block math problem

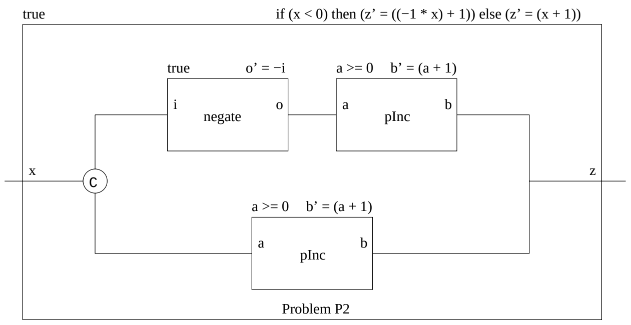
```
$INIT : R_s \leftarrow criterion,$
$O_s \leftarrow \{o | \forall o \in O, \exists r \in R_s, constrains(o, r)\}$
$D_s \leftarrow \emptyset, I_s \leftarrow \emptyset,$
$1. O'_s \leftarrow O_s \cup \{o | \forall o \in O, \exists x \in O_s, dDepend(o, x)\}$
$2. Repeat\ step\ 1\ until\ O'_s = O_s$
$3. I'_s \leftarrow I_s \cup \{i | \forall i \in I, \exists x \in O_s, cDepend(x, i)\}$
$4. I'_s \leftarrow I_s \cup \{i | \forall i \in I, \exists y \in I_s, dDepend(i, y)\}$
$5. Repeat\ step\ 4\ until\ I'_s = I_s$
$6. D'_s \leftarrow D_s \cup \{d | \forall d \in D, \exists o \in O_s, requires(o, d)\}$
$7. D'_s \leftarrow D_s \cup \{d | \forall d \in D, \exists i \in I_s, constrains(i, d)\}$
$8. D'_s \leftarrow D_s \cup \{r | \forall r \in R, \exists o \in O_s, requires(o, r)\}$
$9. R'_s \leftarrow R_s \cup \{r | \forall r \in R, \exists o \in O_s, constrains(o, r)\}$
```

Figure 6.10: Rosetta specification slicing algorithm

```
$INIT : C_F \leftarrow R,$
$criterion \leftarrow \emptyset$
$C_{scp} \leftarrow \emptyset, S_{ssp} \leftarrow \emptyset,$
$1.criterion \leftarrow \{r | \exists r \in C_F\}$
$2.(D_s, R_s, I_s, O_s) \leftarrow slice(criterion)$
$3.S'_{ssp} \leftarrow S_{ssp} \cup \{(D_s, R_s, I_s, O_s)\}$
$4.C'_{scp} \leftarrow C_{scp} \cup \{R_s\}$
$5.C'_F \leftarrow C_F - R_s$
$6.Repeat\ step\ 1\ until\ C_F = \emptyset$
```

Figure 6.11: Smallest criterion partition/smallest slice partition algorithm





```
// Sequential architecture solution
package P2() :: null is
  export all;
begin
  use absVal;
  use pInc;
  facet P2(x :: input real;
           z :: output real) ::
      state_based_semantics is
    export all;
    x__0 :: M__Type(pInc.a);
  begin
    arch0: absVal(x, x__0);
    arch1: pInc(x__0, z);
    pre:   true
    post:  if (x < 0)
           then (z' = ((-1 * x) + 1))
           else (z' = (x + 1)) end if;
  end facet P2;
end package P2;
```

```
// Alternative architecture solution
package P2() :: null is
  export all;
begin
  use negate;
  use pInc;
  facet P2(x :: input real;
           z :: output real) ::
      state_based_semantics is
    export all;
    x__0 :: M__Type(pInc.a);
    x__1 :: M__Type(pInc.a);
    x__2 :: M__Type(negate.i);
  begin
    arch0: if (x >= 0) then (x__1' = x)
           else (x__2' = x) end if;
    arch1: pInc(x__1, z);
    arch2: negate(x__2, x__0);
    arch3: pInc(x__0, z);
    pre:   true
    post:  if (x < 0) then (z' = ((-1 * x) + 1))
           else (z' = (x + 1)) end if;
  end facet P2;
end package P2;
```

Figure 6.12: Solutions to problem P2 using the sequential architecture (left) and the alternative architecture (right)

```
// Parallel architecture solution to problem P6
package P6() :: null is
  export all;
begin
  use gt;
  use geq;
  use and;
  facet P6(a :: input real; b :: input real;
           x :: output real; y :: output real;
           z :: output real)
    :: state_based_semantics is
    export all;
    x__0 :: M__Type(and.m);
    x__1 :: M__Type(and.n);
  begin
    arch0: gt(a, b, x);
    arch1: geq(a, b, x__0);
    arch2: geq(b, a, x__1);
    arch3: and(x__0, x__1, y);
    arch4: gt(b, a, z);

    pre:   true;
    post1: x' = (a > b);
    post2: y' = (a = b);
    post3: z' = (a < b);
  end facet P6;
end package P6;
```

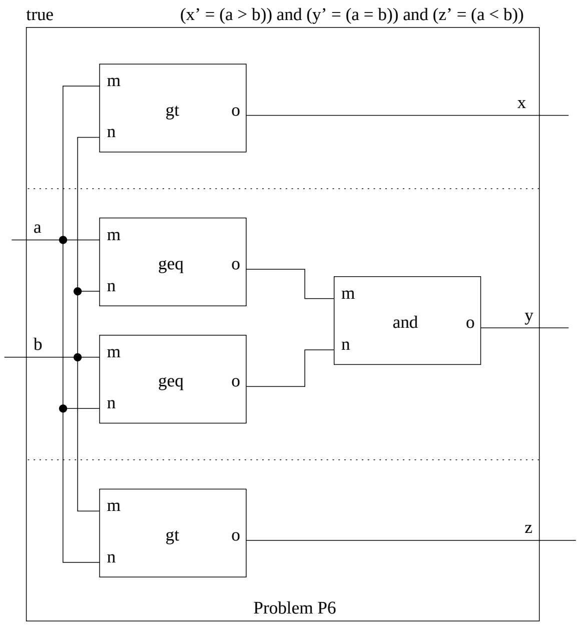


Figure 6.13: Solution to problem P6 using the parallel architecture

```
// Pre-match driven synthesis to problem P1 using
// component negate and the sequential architecture
package P7() :: null is
  export all;
begin

  facet P7(i__0 :: input real;
           i__1 :: input real;
           o__0 :: output real) :: state_based_semantics is
    export all;
    q__0 :: real;
  begin

    pre:  true and (i__0 = (-1 * q__0));
    post: o__0' = (i__1 + q__0);

  end facet P7;
end package P7;
```

Figure 6.14: Pre-match driven synthesis to problem P1 using component negate and the sequential architecture

Chapter 7

Examples

7.1 Record Find Example

In the classic *find* example, first introduced by Penix [44], the problem (in figure 7.1) spcifies the retrieval of a record (defined as a key-value pair) from a list of records given a key. The key type is defined as the set of all unique key names used to construct records in the list, therefore a record must exist in the list of records for all legal keys to the problem.

```
// Find record given a unique key name
package find() :: null is
  export all;
begin

  facet find(a :: input recordList;
             k :: input key;
             z :: output record) :: recorddomain is
    export all;
  begin

    pre:  true;
    post: (getKey(z') = k) and (isMember(z', a));

  end facet find;
end package find;
```

Figure 7.1: Classical record find problem

Following retrieval, the *binary search* component is determined to be a (Bijective) Plug-in Post match. As step 1 in figure 7.3 shows, the binary search component is a partial match because of the additional pre-condition that the record list must be ordered. Intuitively, the input list of records should be sorted to adapt the binary search component and then the problem would be complete, as shown in step 1a. The sub-problem can not be sliced into the parallel architecture as suggested in step 1a for two reasons: 1) the top sub-problem only specifies that the output must be ordered, thus a component that generates an empty list for all inputs would be a match, 2) the semantics of the key varaible can not be rigorously maintained if the key and record list ports are seperated during parallel adaptation. As figure 7.2 shows, parallel adaptation can not be applied since the semantics of the operation of the binary search, with respect to the key value, must be maintained as a partial solution to the find problem. Otherwise the key could be safely modifed in the currently adaptation step, but violate a constraint in a previous adaptation step. Using step 1 fails to locate a solution to the problem.

An appropriate solution follows step 2 using the (1-1) Plug-in Pre matched *sort* component with sequential adaptation results in locating the binary search component to successfully adapt the sort component towards the solution in step 2a.

7.2 Flip Flop Example

The following example illustrates using architectural components to construct larger and more complex architectures to solve new problems. The flip flop problem in figure 7.4 specifies feedback of the output signals into the input signals.

The problem is submitted to SPARTACAS, the first step in the retrieval of the *flipFlop* component. The component retrieved, see figure 7.5, can not be instantiated because the component does

```
// Post-match driven synthesis to find using
// component binary search and the sequential architecture
package P8() :: null is
  export all;
begin

  facet P8(i__0 :: input recordList;
           i__1 :: input key;
           o__0 :: output recordList;
           o__1 :: output key) :: recorddomain is
    export all;
    q__0 :: recordList;
  begin

    pre:  true;
    post: (ordered(o__0')) and
          (not((getKey(q__0) = o__1') and (isMember(q__0, o__0'))) or
           (getKey(q__0) = i__1) and (isMember(q__0, i__0)))

  end facet P8;
end package P8;
```

Figure 7.2: Post-match driven synthesis to find using component binary search and the sequential architecture

not specify the implementation of a flip-flop, rather its specifies an architecture that implements the functionality. The sub-components of the architecture are submitted to SPARTACAS, resulting in component population of the architecture in step (1a).

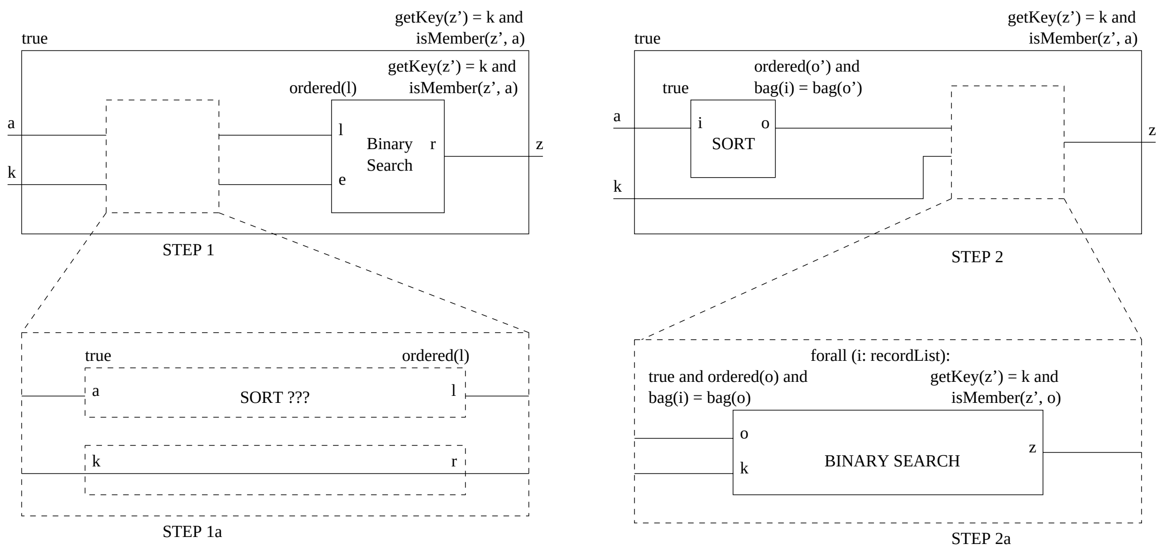


Figure 7.3: Search space for the find problem

```
// Flip flop problem
package ff() :: null is
  export all;
begin

  facet ff(i0 :: input boolean;
           i1 :: input boolean
           o0 :: output boolean;
           o1 :: output boolean) :: state_based_semantics is
    export all;
  begin

    pre:   true;
    post0: o0' = (i0 and o1);
    post1: o1' = (i1 and o0);

  end facet ff;
end package ff;
```

Figure 7.4: Flip flop problem

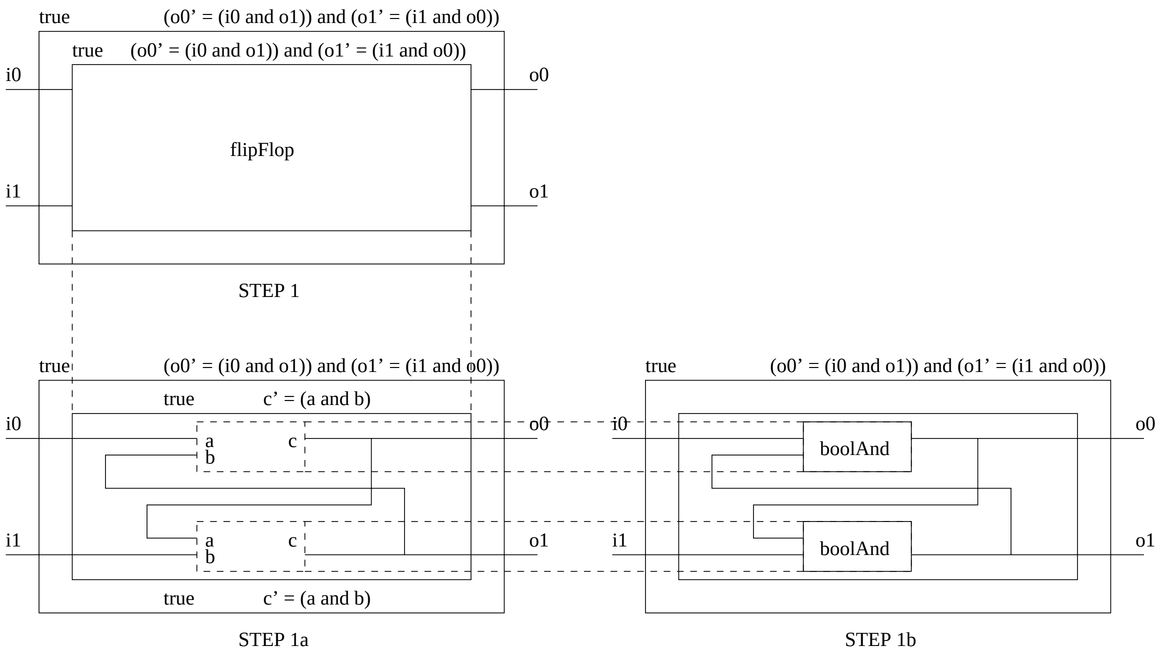

true
(o0' = (i0 and o1)) and (o1' = (i1 and o0))
true
(o0' = (i0 and o1)) and (o1' = (i1 and o0))
i0
o0
flipFlop
i1
o1
STEP 1
true
(o0' = (i0 and o1)) and (o1' = (i1 and o0))
true
c' = (a and b)
i0
a
b
c
o0
i1
a
b
c
o1
true
c' = (a and b)
STEP 1a
true
(o0' = (i0 and o1)) and (o1' = (i1 and o0))
i0
boolAnd
o0
i1
boolAnd
o1
STEP 1b

Figure 7.5: Search space for the flip-flop problem

Chapter 8

Evaluation

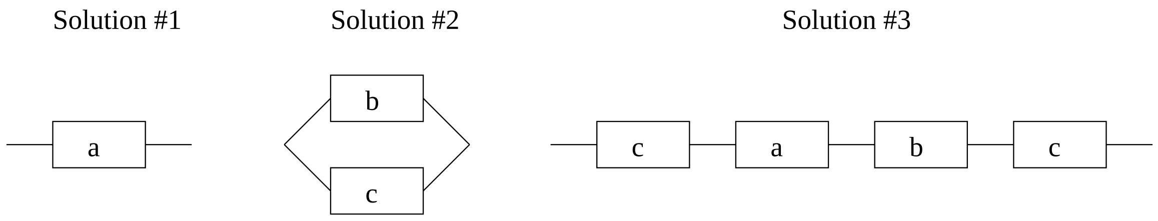
Precision and *recall* metrics are traditionally used to evaluate the performance of component retrieval. Precision is defined as the ratio of correct solutions retrieved to the total number of results retrieved. High precision is the result of retrieving few irrelevant or invalid results. Recall is defined as the ratio of the number of correct solutions retrieved to the number of correct solutions that exist in the library. Ideally recall should be high, meaning correct solutions should not be missed in the library. Generally, the trade-off between recall and precision is inversely proportional since increasing recall increases the probability of retrieving irrelevant results.

The equation for recall must be modified since a library may contain an infinite number of possible architectural solutions. Some of these solutions may contain nonsensical or redundant configurations of components, yet nevertheless solve the problem. For instance, if the goal is to find a solution to a problem that increments an input value, an infinite number of solutions can be constructed by using N decrement components and N+1 increment components. The traditional equation for recall will always evaluate to zero when an infinite number of solutions exist. Three different methods for calculating recall are proposed in definitions 8.0.1, 8.0.2, and 8.0.3, which use different finite grouping relations. Figure 8.1 illustrates the solution grouping relationships for calculating recall. In addition to precision and recall, the average number of components per solution and the average number of proof obligations will also be calculated.

Definition 8.0.1 *$Recall_1$ is defined as the ratio of the number of relevant component groups retrieved to the number of relevant component groups in the library. The grouping relation is defined as the containment of some combination (without replacement) of components such that a solution exists.*

Definition 8.0.2 *$Recall_2$ is defined as the ratio of the number of relevant component groups retrieved to the number of relevant component groups in the library. The grouping relation is defined as the containment of the smallest combination (without replacement) of components such that a solution exists.*

Definition 8.0.3 *$Recall_3$ is defined as the ratio of the number of relevant solutions retrieved to the number of relevant solutions in the library, where a solution contains a threshold of N components, where N is some integer greater than 0.*



Equation	*Solution Groups*	*Number of Solutions*
Recall_1	Solutions in group {a}: #1 Solutions in group {b, c}: #2 Solutions in group {a, b, c}: #3	3
Recall_2	Solutions in group {a}: #1 and #3 Solutions in group {b, c}: #2 and #3	2
Recall_3	Solutions with N=2: #1 and #2	2

Figure 8.1: Calculating recall example using components a, b, and c

8.1 Evaluation Library and Query Set

Evaluation is performed over a library containing 46 complex mathematical specifications, a library of 106 list manipulation specifications, a library of 30 record manipulation specifications, and a library consisting of 42 digital signal processing specifications. These libraries have also been used in evaluating other works [47, 17, 42, 4].

The query set contains 103 queries, which can be classified into six types: queries for solutions that are solved (1) by one and only one component, (2) by a single component, but can also be solved by an architecture of components, (3) only by an architecture of components, (4) by an infinite number of architecture configurations, (5) by multiple sub-architectures, or (6) by components from multiple libraries. The experiments are designed to test the automated adaptation of partial solutions using adaptation architectures, hence the majority of queries are designed to be solved by an architecture. The experiments in other works using the same libraries focused on retrieval of single component solutions to queries, therefore a direct comparison of results can not be performed.

8.2 Empirical Results

The SPARTACAS framework can be tuned using the port connection and depth variables. The port connection method, ρ, can be one of the methods described in chapter 5. As outlined, the less restrictive port connection methods generate a greater number of signature combinations than the bijective port connection method. The signature combinations must be checked by a theorem-prover, resulting in a time and computation expense. A trade-off in recall and proof obligations is expected when comparing the port connection methods.

The retrieval framework uses a depth-first architecture construction strategy. Since it is possible to get trapped in the construction of an architecture without an end, there has to be a depth, or a limit on the number of components used in the construction of an architecture. The depth, denoted as δ, is equivalent to the number N in the definition of Recall$_3$. Clearly a trade-off in recall and proof obligations is expected since increasing the depth allows SPARTACAS to search longer. The experiment will be performed with ($\sigma = 1$) and without ($\sigma = 0$) adaptation capabilities.

Expr	(σ, ρ, δ)	*Recall*$_1$	*Recall*$_2$	*Recall*$_3$	*Precision*	*Proof Obligations*	$\frac{Components}{Solution}$
1.	(0, -, 2)	.04(0-1)	.08(0-1)	.15(0-1)	1(1-1)	2.5(1-4)	1(1-1)
2.	(0, -, 5)	.04(0-1)	.08(0-1)	.15(0-1)	1(1-1)	2.5(1-4)	1(1-1)
3.	(0, -, 8)	.04(0-1)	.08(0-1)	.14(0-1)	1(1-1)	2.5(1-4)	1(1-1)
4.	(1, Bi, 2)	.29(0-1)	.35(0-1)	1(1-1)	1(1-1)	24.5(1-41)	1.3(1-2)
5.	(1, 1-1, 2)	.33(0-1)	.52(0-1)	1(1-1)	1(1-1)	33.8(1-50)	1.4(1-2)
6.	(1, Onto, 2)	.31(0-1)	.5(0-1)	1(1-1)	1(1-1)	51.0(1-71)	1.4(1-2)
7.	(1, Bi, 5)	.4(0-1)	.58(0-1)	1(1-1)	.99(.8-1)	57.3(1-80)	2.4(1-5)
8.	(1, 1-1, 5)	.64(0-1)	.79(0-1)	.99(.8-1)	.99(.8-1)	88.2(1-102)	2.8(1-5)
9.	(1, Onto, 5)	.59(0-1)	.64(0-1)	.99(.75-1)	.99(.8-1)	118.0(1-145)	2.6(1-5)
10.	(1, Bi, 8)	.61(.15-1)	.70(.4-1)	.97(.75-1)	.98(.8-1)	150.0(1-188)	4.0(1-8)
11.	(1, 1-1, 8)	.82(.4-1)	.91(.54-1)	.96(.75-1)	.98(.8-1)	188.9(1-201)	4.5(1-8)
12.	(1, Onto, 8)	.78(.35-1)	.81(.5-1)	.96(.75-1)	.98(.8-1)	258.4(1-289)	4.3(1-8)

Table 8.1: Empirical results

The framework was implemented and evaluation results were obtained over the query set. Table 8.1 shows a portion of the results obtained, minimum and maximum values are shown in parentheses. Experiments 1 trough 3 show the results of running SPARTACAS without the adaptation features. Although SPARTACAS was able to retrieve results quickly with high precision, the recall was around 10%. Low recall in these experiments were the result of queries that could not be solved by a single component in the library. Experiments 4 through 12 used the adaptation features in SPARTACAS. The results generally show that high recall was gained without giving up precision. High precision was the result of using formal methods to verify and maintain the constraints of the problem during solution construction. Variable renaming in one example caused an architectural resolution conflict, thus adversely affecting precision.

Figure 8.2 shows the relationships between recall and the number of proof obligations versus the

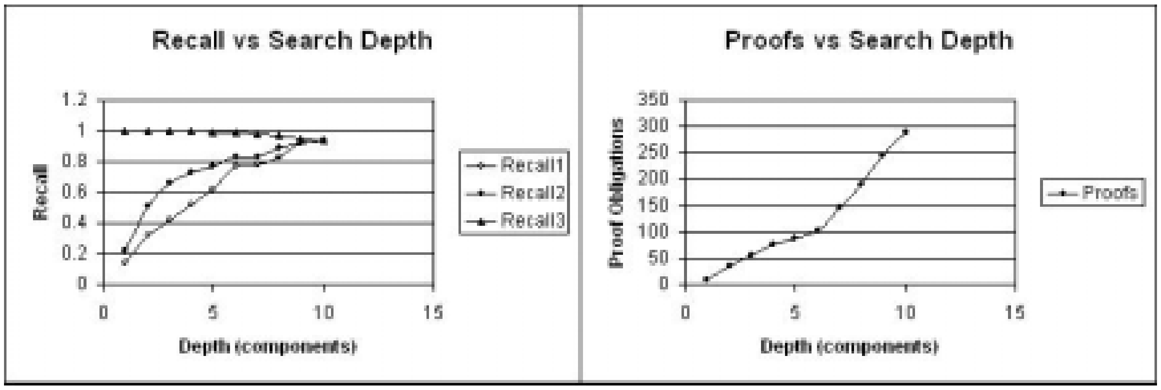


Figure 8.2: Search depth effects on performance (ρ = 1-1)

search depth. Since the number of components per solution as well distributed for small depths, Recall_1 had a steady increase versus the search depth before leveling off. Recall_2 initially had large gains since many solutions groups were found with a small search depth, however as the search depth was increased there was diminishing returns since SPARTACAS is less likely to find a solution to an undiscovered solution group. Recall_3 was high for small search depths, but slowly decreased as the search depth increased. This result is accredited to the theorem-prover's strategies inability to prove match conditions on some of the large and complex synthesized sub-problems. The results also show an exponential increase in the number of proof obligations per solution. This result was expected since as the search depth is increased SPARTACAS is likely to generate more proof obligations when adapting partial matches.

Figure 8.3 shows the impact that the port connection methods have on recall (using Recall_2) and the number of proof obligations. The graph clearly shows an increase in recall at the expense of proof obligations when using a less restrictive port connection. The onto port connection method had early gains in recall but did not provide much increase when the depth was increased, whereas the one-to-one port connection provided better recall results. The functionality of components using one-to-one port connection were more applicable to the problems than using onto port connection.

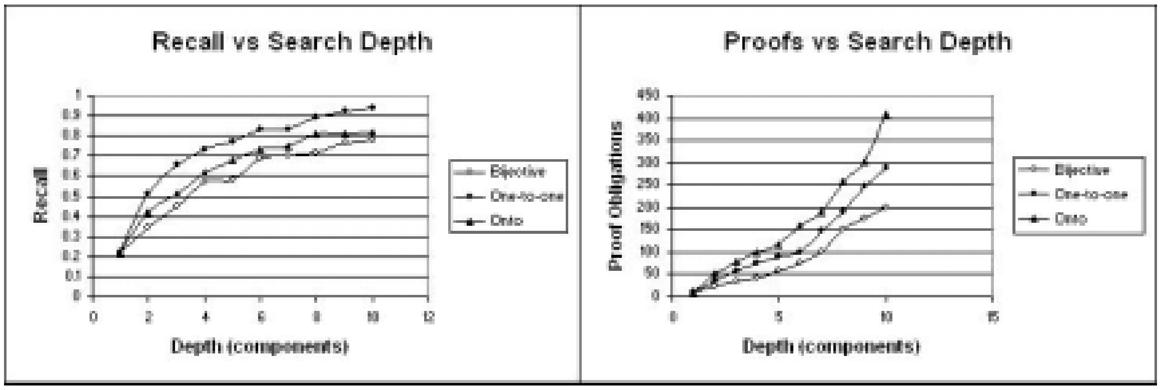


Figure 8.3: Port connection effects on performance

Using onto port connection also had higher proof obligation count since onto port connection requires an exponential number of possible port instantiantions to verify.

8.3 Implementation Platform

The Rosetta specifications are initially parsed by the Rosetta ANTLR parser, which generates a data structure called the Rosetta Object Model (ROM). The ROM is then converted to an intermediate notation (XML) using a simple multi-visitor pattern written in Java. The benefit of using an intermediate notation is that it allows SPARTACAS to be theorem-prover independent. Various retrieval engines can be plugged into the layered architecture, as long as the engine can translate the XML to its native language. For this experiment, SPARTACAS uses SOCCER2 (an updated version of SOCCER [42]) as the retrieval engine. SOCCER2 uses PVS to perform automated theorem-proving for feature classification in the feature-based retrieval engine and logical relational proofs in the specification-based retrieval engine. An XSLT script converts the XML into a PVS form that can be interpreted by SOCCER2. SOCCER2 generates a list of components that match or partially match the problem as well as the signature used to instantiate the component. For partial matches, a Common Lisp program synthesizes sub-problems which are fed back to the retrieval engine. A small Java program is used to coordinate the process and keeps a record of the architectural blueprints that have been constructed. Users can also run SPARTACAS interactively at each search/adaptation step.

Several search and strategy variables can be used to tune SPARTACAS. In addition to setting the search depth and port connection method, the number of solutions retrieved can be specified. The retrieval strategies can also be modified, including the feature-match threshold and the specification-match strategy. The user can specify any match condition listed in figure 2.3, or use *order-by-strength*, *satisfies-first*, or *complete-search* strategies. The order-by-strength strategy will first return components that are a complete match to the problem. If no matches exist, then post-matches are returned. If no matches yet exist, pre-matches are attempted and returned. The satisfies-first strategy will first return complete matches, else will return post- and pre-matches if no complete matches exist. Lastly, the complete search will return components that match under any of the match conditions. During this experiment, SPARTACAS was set to search for all possible solutions while varying the search depths and port connection methods. The feature-match threshold was set to 50% and the complete-search strategy was used in the specification-based retrieval engine.

Chapter 9

Future Work and Limitations

The architectures synthesized by SPARTACAS use shared-variable communication. SPARTACAS needs to address various communication protocols. The framework can resolve this issue by: 1) including communication protocols as a search criteria when selecting components during retrieval, or 2) populating the component library with communication connector specifications [3] that can be retrieved and instantiated. The latter approach would maintain the generality of the component specifications and the SPARTACAS framework, however it would increase the overhead in the retrieval engine due to the additional connector searches.

Work needs to be invested in applying reuse to different domains. The reuse framework was evaluated on components in domains that used state-based semantics, where results are computed and placed on the output in the next state after the arrival of input. Domains may also determine the type of architectures that are applicable. Other semantic issues also need to be explored when constructing solutions. For instance, the user may consider the propagation delay of values in an architecture as a factor in a solution.

SPARTACAS is limited in the type of adaptation architectures that it can synthesize. SPARTACAS is only capable of synthesizing three types of adaptation architectures (sequential, alternative, and parallel). Future work may include other architectures as tactics for adaptation. The currently framework allows architectures, such as feedback architectures, to be stored in the library which can be retrieved and populated in a solution.

SPARTACAS is limited by the proving power of theorem-provers. There are proofs that are incapable of being solved, even with today's theorem-provers and processing power. The implementation is also limited by the depth that can be searched, SPARTACAS currently uses a depth-first search strategy in the architectural search tree. Other search strategies that estimate the cost (e.g. number of proof obligations) to reach a solution could also be applied. SPARTACAS could also be parallelized and distributed among processors, which might reduce the cost or time to reach a solution. Other methods to decrease the cost to reach a solution set also need to be investigated.

SPARTACAS currently does not rank solutions since the ranking criteria is dependent on the features that the user wishes the solutions to possess. Allowing the user to specify such criteria (e.g. number of components used, propagation delay, extra technical information [22]) before retrieval can be added. In the event that a complete solution can not be found, the framework optionally returns all partially completed architectures and the sub-problem specification required to satisfy the rest of the problem. The sub-problem may or may not be easier to implement from scratch than the original design problem, although estimating the complexity of implementing one of the sub-problem specifications is difficult.

Chapter 10

Related Work

10.1 Specification-based Retrieval

SPARTACAS builds largely upon previous work on specification-based retrieval techniques. Zaremski and Wing [64, 63] provided the precise logical relationships for comparing and organizing specifications. They constructed a tool to retrieve Larch/ML specifications and to evaluate issues such as subtyping and interoperability. The lattice of logical relationships is not only used to evaluate retrieval in the SPARTACAS framework, but also to evaluate the tactics for adaptation. The match conditions are used to determine which adaptation architecture tactic can be applied and to derive the functionality required for successful adaptation.

John Penix [48, 47] presented a methodology to automate the classification and retrieval of VSPEC components using the Larch theorem-prover. Penix motivates the use of semantic feature classification of components to improve the performance of specification-based retrieval. Feature-based retrieval is used to reduce the search space by eliminating components that do not have features in common with the problem. The reduced search space is given to the theorem-prover where match conditions are proven and used to evaluate reuse. Empirical experiments were carried out to evaluate the performance of the retrieval mechanism. Makarand Patil [42] later extended this methodology to Rosetta specifications and the PVS theorem-prover in a system called SOCCER. SPARTACAS uses an improved version of SOCCER (the new version includes stronger proof tactics and a layered retrieval engine) as the main retrieval engine.

Bernd Fischer and others [13, 14, 17] designed a deductive retrieval tool called NORA/HAMMR to progressively filter components using a series of filters, theorem-provers, and model checkers. They also used techniques to reduce the axiom and proof sets when proving a match between specifications. SPARTACAS also uses the layered filter approach to retrieval, making use of feature- and signature-based retrieval engines to filter out components. The last layer is a specification-based retrieval engine that proves a match condition between a component and a problem.

10.2 Component Adaptation

Penix [43, 45] presents a framework called REBOUND for component retrieval and adaptation. He introduces behavioral adaptation using adaptation architectures. SPARTACAS uses the REBOUND framework to automate the adaptation of components that partially match a problem. The framework and implemented and experimental results were presented in this work and elsewhere [39, 40]. The adaptation tactics are used to select an adaptation architecture theory, which is used to precisely deduce and synthesize a sub-problem to solve the adaptation requirements. Penix suggested that the parallel adaptation tactic uses the bottom-up approach to adaptation, where components are adapted in parallel such that union of all the features of the components matched those of the problem. SPARTACAS uses the top-down approach to decompose problems using specification slicing. Components that match the sub-problems are composed in parallel to satisfy the problem.

Purtilo and Atlee [51] have developed a system, called NIMBLE, that aides software designers by automating the adaptation of module interfaces. Adaptation of a module interface involves reordering, type coercion, and/or initializing or masking parameters. NIMBLE allows programmers to declare mappings and type adaptations of a program's interface, which gets transformed into a module that encapsulates the desired adaptation. SPARTACAS uses a signature-based retrieval engine that automates the necessary reordering and type coercion when matching the interface of the problem to the interface of the component. The information is stored and later used to properly instantiate the parameter configurations in an architecture specification.

Jeng and Cheng [30] describe an approach to reusing general components and identifying modifications required to satisfy a query specification. The modifications include solving type inconsistencies and undefined operations. Although SPARTACAS does not modify a component specification, SPARTACAS uses a similar notion to identify the missing functionality which is synthesized into a sub-problem. The necessary behavioral modifications occurs during component interactions in the architecture.

10.3 Synthesizing, Slicing, and Architecting for Reuse

Chen and Cheng [10] developed an architecture-based reuse environment called ABRIE. ABRIE provides a graphical representation of an architecture where each architectural element (e.g. components and connectors) contain a description of its properties. ABRIE also provides a semi-automated capability to evaluate reuse of existing components to be reused and instantiated in the architecture. SPARTACAS automates the construction and instantiation of architectures to satisfy design problems without user assistence.

Zhao [65] developed a slicing approach for architecture description specifications for reuse-of-the-large. A large description of a software system is described in an ADL using a collection of elements (components and connectors); slicing is used to extract and reuse elements or collections of elements in other system designs. SPARTACAS uses slicing for reuse-of-the-small at the specification level. Problems are decomposed into smaller sub-problems such that matching components can be reused to solve the independent sub-problems.

Fischer and Whittle [16] have investigated the integration of deductive retrieval and deductive synthesis into one framework. Meta-variables are used to represent program fragments yet to be synthesized. They discuss that the meta-variables can be used to represent the synthesis of a specification required for component adaptation based on pre-defined tactics. Their framework was only able to achieve partial automation.

Bhansali [6] uses a hybrid approach to obtain a cost-effective reuse strategy at the code level to solve configurations of geometric bodies. The approach uses a combination of architecture-driven reuse, code component retrieval, as well as program synthesis. In their work they recognize that a component is more likely to be reused in an architectural design. The synthesizer semi-automatically synthesizes partial code fragments from high-level specifications of domain knowledge. SPARTACAS uses an automated synthesizer to synthesize sub-problems, which are recursively used to search for solutions in the library.

Chapter 11

Conclusions

Reuse can potentially contribute many benefits to the software design cycle, but the costs associated with reuse must be reduced for reuse to become more common. A significant cost is the search and adaptation of components to satisfy a design problem. Most works have achieved efficient and effective component retrieval, but few works have concentrated on adapting partial matches. For adaptation to be feasible, the process needs to be reliable, scalable, error-free, and automated.

In this thesis, the SPARTACAS framework for automating specification-based component retrieval and adaptation for software reuse was presented. The thesis motivated using formal specifications to represent reusable software; which gives a formal foundation to the retrieval and adaptation process. SPARTACAS uses a layered retrieval architecture using feature-based, signature-based, and specification-based retrieval engines to retrieve components that, completely or partially, match a problem. The layered approach to retrieval reduces the cost of retrieving components from large and diverse libraries. Three adaptation architecture theories for adapting the behavior of partially matching components were specified. These theories (sequential, alternative, and parallel) specify the configuration of sub-components and specify the relationship between the functionality of the architecture and the functionality of the sub-components. Using this relationship, a sub-problem can be synthesized and used to search for components for adaptation. The components are instantiated in the adaptation architecture. The adaptation architecture effectively adapts the behavior of partially-matched components by imposing interactions with other components. The resulting architecture is a solution to the problem that can be added to the library of components, further increasing the prospects of reuse.

This work makes the following contributions:

- Presents SPARTACAS, a fully automated specification-based component retrieval and adaptation tool.
- Illustrate behavioral adaptation can be implemented using adaptation architectures.
- Demonstrates specification slicing as a tactic for problem decomposition for parallel adaptation.
- Provides a sound and formal methodology for synthesizing sub-problems for adapting partial matches.
- Supports the retrieval and adaptation framework with empirical results.
- Discusses the trade-offs in recall and the number of proof obligations to prove.

The framework was implemented and evaluated. The results showed SPARTACAS was able to recall approximately 94% of possible solutions while maintaining high precision with adaptation, compared to 4-15% recall without adaptation. The adaptation process was time consuming on large and complex examples, however, in general, correct solutions were retrieved if such solutions exist. Other limitations and future work on the framework were also discussed.

Bibliography

[1] P. Alexander, D. Barton, and C. Kong. *Rosetta Usage Guide.* The University of Kansas / ITTC, 2335 Irving Hill Rd, Lawrence, KS, 2000.

[2] Perry Alexander and Cindy Kong. Rosetta: Semantic support for model-centered systems-level design. *IEEE Computer*, 34(11):64–70, November 2001.

[3] R. Allen and D. Garlan. Formalizing Architectural Connection. In *Proc. Sixteenth International Conference on Software Engineering*, pages 71–80, May 1994.

[4] Thomas Baar and Bernd Fischer. Solving software reuse problems with theorem provers. In *CADE-15 Workshop Problem-solving Methodologies with Automated Deduction*, Lindau, Germany, 1998.

[5] E. T. Bell. Exponential numbers. *American Math Monthly*, 41:411–419, 1934.

[6] Sanjay Bhansali. A hybrid approach to software reuse. In *ACM-SIGSOFT Symposium on Software Reusability*, Seattle, WA, April 1995.

[7] Bruce B. Burton, Rhonda Wienk Aragon, Stephen A. Bailey, Kenneth D. Koehler, and Lauren A. Mayes. The reusable software library. *IEEE Software*, 4:25–33, July 1987.

[8] G. Canfora, A. Cimitile, A. De Lucia, and G.A. Di Lucca. Software salvaging based on conditions. In *Proceedings of the International Conference on Software Maintenance*, pages 424–433, Victoria, Canada, 1994.

[9] Patrick S. Chen, Rolf Hennicker, and Matthias Jarke. On the retrieval of reusable software components. In *Advances in Software Reuse - Second International Workshop on Software Reusability*, Lucca, Italy, 1993. IEEE Computer Society.

[10] Yonghao Chen and Betty H. C. Cheng. Facilitating an automated approach to architecture-based software reuse. In *Proceedings of 12th IEEE International Conference on Automated Software Engineering*, pages 238–245, Incline Village, NV, November 1997.

[11] J. H. Conway and R. K. Guy. *The Book of Numbers*. Springer-Verlag, New York, 1996.

[12] David Eichmann and Kankanahalli Srinivas. Neural network-based retrieval from software reuse repositories. In R. Beale and J. Findlay, editors, *Neural Networks and Pattern Recognition in Human Computer Interaction*, pages 215–228. Eillis Horwood Ltd., West Sussex, UK, March 1992.

[13] B. Fischer, M. Kievernagel, and G. Snelting. Deduction-based software component retrieval. In *IJCAI95 Workshop on formal Approaches to the Reuse of Plans, Proofs, and Programs*, 1995.

[14] 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.

[15] B. Fischer and J. Schumann. SETHEO goes software engineering: Application of ATP to software reuse. In *Proc. CADE-14*, July 1997.

[16] B. Fischer and J. Whittle. An integration of deductive retrieval into deductive synthesis. In *Automated Software Engineering*, pages 53–61, Cocoa Beach, FL, October 1999.

[17] Bernd Fischer and Johann Schumann. NORA/HAMMR: Making deduction-based software component retrieval practical. In *Proc. CADE-14 Workshop on Automated Theorem Proving in Software Engineering*, July 1997.

[18] W. B. Frakes and P. B. Gandel. Representation methods for software reuse. In *Proceedings of Tri-Ada '89*, pages 302–314. Association for Computing Machinery, October 1989.

[19] William Frakes and Carol Terry. Software reuse: Metrics and models. *ACM Computing Surveys*, 28(2):415–435, June 1996.

[20] William B. Frakes and Christopher J. Fox. Sixteen questions about software reuse. *Communications of the ACM*, 38(6):75–87, June 1995.

[21] K.B. Gallagher and J.R. Lyle. Using program slicing in software maintenance. *IEEE Transaction on Software Engineering*, 17(8):751–761, 1991.

[22] John H. Gennari and Mark Ackerman. Extra-technical information for method libraries. In *Proceedings of 12th Workshop on Knowledge Acquisition, Modeling and Management (KAW '99)*, Banff, Alberta, Canada, October 1999.

[23] M. R. Girardi and B. Ibrahim. Automatic indexing of software artifacts. In *Proceedings of 3rd. International Conference on Software Reuse*, pages 24–32, Rio de Janeiro, Brazil, November 1994.

[24] M. R. Girardi and B. Ibrahim. Using english to retrieve software. *The Journal of System and Software*, 30(3):249–270, September 1995. Special Issue on Software Reuse.

[25] Allen Goldberg. Reusing software developments. In R. N. Taylor, editor, *Proceedings of the 4th ACM SIGSOFT Symposium on Software Development Environments*, pages 107–119, Irvine, California, USA, December 1990. ACM Press.

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

[27] Robert J. Hall. Generalized behavior-based retrieval. In *Proceedings of the 15th International Conference on Software Engineering*, pages 371–380, Baltimore, MD, May 1993. ACM Press.

[28] Scott Henninger. Supporting the construction and evolution of component repositories. In *Proceedings of the 18th International Conference on Software Engineering*, pages 279–288, Berlin, Germany, March 1996. IEEE Computer Society Press.

[29] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12:576–580,583, 1969.

[30] 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.

[31] Jun-Jang Jeng and Betty H. C. Cheng. Specification matching for software reuse: A foundation. In *Proceedings of the ACM SIGSOFT Symposium on Software Reuse*, pages 97–105, Seattle, Washington, April 1995.

[32] L. Jilani, J. Desharnais, M. Frappier, R. Mili, and A. Mili. Retrieving software components that minimize adaptation effort. In *Automated Software Engineering*, pages 255–, 1997.

[33] Capers Jones. Economics of software reuse. *Computer*, 27:106–7, July 1994.

[34] David B. Leake Andrew Kinley and David Wilson. *Case-Based Reasoning: Experiences, Lessons, and Future Directions*, chapter Learning to Improve Case Adaptation by Introspective Reasoning and CBR. AAAI Press/MIT Press, 1996.

[35] Charles W Krueger. Software reuse. *Computing Surveys*, 24:131–83, June 1992.

[36] Yoelle S. Maarek, Daniel M. Berry, and Gail E. Kaiser. An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on Software Engineering*, 17:800–13, August 1991.

[37] 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.

[38] Hafdeh Mili, Fatma Mili, and Ali Mili. Reusing software: Issues and research directions. *IEEE Transactions on Software Engineering*, 21(6):528–562, June 1995.

[39] Brandon Morel and Perry Alexander. A slicing approach for parallel component adaptation. In *Proceedings of the Tenth IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, pages 108–114, Huntsville, AL, April 2003. IEEE Computer Society Press.

[40] Brandon Morel and Perry Alexander. Spartacas: Automating component adaptation and reuse. Technical report, Information and Telecommunication Technology Center, University of Kansas, August 2003.

[41] Tomohiro Oda and Keijiro Araki. Specification slicing in formal methods of software development. In *Proceedings of the 17th Annual International Computer Software and Applications Conference*, pages 313–319. IEEE Computer Society Press, November 1993.

[42] Makarand Patil. Soccer - a specification matching-based component retrieval system. Master's thesis, University of Kansas, 2000.

[43] J. Penix and P. Alexander. Component reuse and adaptation at the specification level. In *8th Annual Workshop on Institutionalizing Software Reuse*, Ohio State University, Columbus, March 1997.

[44] John Penix. *Automated Component Retrieval and Adaptation Using Formal Specifications*. PhD thesis, University of Cincinnati, April 1998. In preparation.

[45] John Penix. Rebound: A framework for automated component adaptation. In *Proceedings of the 9th Annual Workshop on Software Reuse*, January 1999.

[46] John Penix and Perry Alexander. Toward automated component adaptation. In *Proceedings of the Ninth International Conference on Software Engineering and Knowledge Engineering*, pages 535–542. Knowledge Systems Institute, June 1997.

[47] John Penix and Perry Alexander. Efficient specification-based component retrieval. In *Automated Software Engineering*, volume 6, pages 139–170. Kluwer Academic Publishers, 1999.

[48] 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.

[49] 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.

[50] Rubén Prieto-Díaz. Implementing faceted classification for software reuse. *Communications of the ACM*, 34(5):88–97, May 1991.

[51] James M. Purtilo and Joanne M. Atlee. Module reuse by interface adaptation. *Software: Practice & Experience*, 21(6):539–556, June 1991.

[52] Ashwin Ram and Jr. Anthony G.Francis. *Case-Based Reasoning: Experiences, Lessons, and Future Directions*, chapter Multi-plan Retrieval and Adaptation in an Experience-Based Agent. AAAI Press/MIT Press, 1996.

[53] Eugene J. Rollins and Jeannette Wing. Specifications as search keys for software libraries. In *Proceedings of the Eight International Conference on Logic Programming.* 1991.

[54] G. C. Rota. The number of partitions of a set. *American Math Monthly*, 71:498–504, 1964.

[55] Andreas Schlapbach. Enabling white-box reuse in a pure composition language. Master's thesis, University of Bern, January 2003.

[56] Guttorm Sindre, Reidar Conradi, and Even-Andre Karlsson. The REBOOT approach to software reuse. *The Journal of Systems and Software*, 30(3):201–212, September 1995.

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

[58] Barry Smyth and Mark T. Keane. Experiments on adaptation-guided retrieval in case-based design. Technical Report TCD-CS-94-17, Trinity College, Dublin, dec 1994.

[59] Mark Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method.* PhD thesis, University of Michigan, Ann Arbor, MI, 1979.

[60] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.

[61] J. Wing. A Specifier's Introduction to Formal Methods. *IEEE Computer*, 23(9):8–24, sep 1990.

[62] A.M. Zaremski and J.M. Wing. Signature matching: A key to reuse. In *ACM SIGSOFT Symp. on the Foundations of Software Engineering*, December 1993.

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

[64] 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.

[65] Jianjun Zhao. A slicing-based approach to extracting reusable software architectures. In *Proceedings of the 4th European Conference on Software Maintenance and Reengineering*, pages 215–223, 2000.