

A Specification Matching Based Approach to Reverse Engineering *

Gerald C. Gannod[†]

Computer Science and Engineering
Arizona State University
Box 875406
Tempe, AZ 85287-5406
(602) 727-4475
gannod@asu.edu

Betty H.C. Cheng

Computer Science and Engineering
Michigan State University
3115 Engineering Building
East Lansing, MI 48824-1226
(517) 355-8344
chengb@cse.msu.edu

ABSTRACT

Specification Matching is a technique that has been used to retrieve reusable components from reuse libraries. The relationship between a query specification and a library specification is typically based on *refinement*, where a library specification matches a query specification if the library specification is more detailed than the query specification. Reverse engineering is a process of analyzing components and component interrelationships in order to construct descriptions of a system at a higher level of abstraction. In this paper, we define the concept of an *abstraction* match as a basis for reverse engineering and show how the abstraction match can be used to facilitate a process for *generalizing* specifications. Finally, we apply the specification generalization technique to a portion of a NASA JPL ground-based mission control system for unmanned flight systems.

Keywords

Reverse engineering, Software Maintenance, Formal Methods

1 INTRODUCTION

Many well-documented system failures have been attributed to software. Some of the most notable incidents include the catastrophic failures of the Therac-25 radiation therapy system [13] and the Ariane 5 spacecraft [18]. A commonly overlooked aspect of these failures has been the fact that both were the result of an improper reengineering of software from one version to another. That is, the Therac-20 software was “reused” to produce the Therac-25 software, and the Ariane 4 software was reused to produce the Ariane

5 software. In both cases, critical requirements of the original software were not preserved, and the modified software failed to perform as expected.

Software reverse engineering is the process of analyzing the components and component interrelationships of a software system in order to describe that system at a level of abstraction above that of the original system [5]. Our previous investigations into reverse engineering have focused on the use of the *strongest postcondition* for deriving formal specifications from imperative program code [8]. By defining the formal semantics of each of the constructs of a programming language, a formal specification of the behavior of a program written using the given programming language can be constructed. These specifications, however, are typically considered low-level, as-built specifications since the degree of implementation bias is rather high. As such, the introduction of abstraction into the as-built specifications is desired. That is, high-level descriptions based on the low-level, as-built specifications are required in order to perform high-level reasoning about the abstract functional behavior of the original subject software system. In addition, the abstract specifications can be used to support the population of reusable component libraries [7].

In this paper, we describe a formal approach for deriving abstract functional specifications from low-level, as-built specifications that is based on preserving a *match* relationship [11, 23] between two specifications. The remainder of this paper is organized as follows. Section 2 gives background information regarding the use of formal methods and the issues related to software maintenance and software reuse. The use of specification matching to facilitate reverse engineering is discussed in Section 3. Section 4 describes an abstraction or “*specification generalization*” technique for deriving abstract functional behavior from as-built specifications. Section 5 describes a detailed example that applies the specification generalization technique to a NASA JPL ground-based mission control application. Related work is summarized in Section 6, and Section 7 draws conclusions and suggests further investigations.

2 BACKGROUND

This section gives background information in the areas of software maintenance and formal methods for software de-

*This work is supported in part by the NASA Graduate Student Researcher Fellowship NGT-70376 and the National Science Foundation grants CISE/EIA-9617310, CDA/EIA-9700732, CCR-9633391, CCR-9407318, CCR-9209873, and CDA-9312389.

[†]This author was supported in part by a NASA Graduate Student Researchers Program Fellowship. A portion of this research was performed while this author was at the NASA Jet Propulsion Laboratory.

velopment.

Software Maintenance

Figure 1 contains a graphical depiction of a process model for reverse engineering and reengineering [3]. The process model is captured by two sectioned triangles, where each section in a triangle represents a different level of abstraction. The higher levels in the model are *concepts* and *requirements*. The lower levels include *designs* and *implementations*. Entry into this reengineering process model begins with system *A*, where *Abstraction* (or reverse engineering) is performed to a level of detail appropriate to the task being performed. For instance, if a system is to be reengineered in response to efficiency constraints, then abstraction to the design level may be appropriate. The next step is *Alteration*, where the system is configured into a new form at a different level of abstraction. Finally, *Refinement* of the new form into an implementation can be performed to create system *B*.

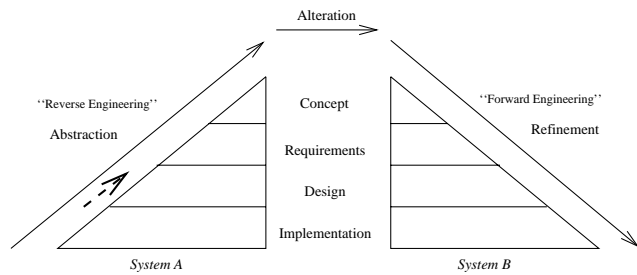


Figure 1: Reengineering Process Model

This paper describes an approach to reverse engineering that is applicable to the *design* level. In Figure 1, the context for this paper is represented by the dashed arrow. That is, we address the introduction of abstraction into as-built design specifications in order to derive higher-level descriptions of software.

Formal Methods

Although the waterfall development life-cycle provides a structured process for developing software, the design methodologies that support the life-cycle make use of informal techniques, thus increasing the potential for introducing ambiguity, inconsistency, and incompleteness in designs and implementations. In contrast, formal methods used in software development are rigorous techniques for specifying, developing, and verifying computer software [22]. A formal method consists of a well-defined specification language with a set of well-defined inference rules that can be used to reason about a specification using automated techniques [22].

Strongest Postcondition

In our previous investigations we described the use of strongest postcondition as the formal basis for reverse engineering [8]. In addition, we have applied strongest postcondition to the definition of the semantics of the C pro-

gramming language in order to analyze software used by mission control technicians for NASA’s deep space program. *Strongest postcondition*, denoted $sp(S, Q)$ is defined as follows: given that Q holds, execution of S results in $sp(S, Q)$ true, if S terminates [6]. Given a precondition Q and a program S , sp derives a predicate $sp(S, Q)$ describing the behavior of program S . Note that the precondition Q can take many different values and is not limited to the assignment “true”.

Table 1 summarizes the strongest postcondition semantics of the Dijkstra guarded command language [6], where IF represents the n alternative conditional statement

```

if
  B1 → S1;
  ...
  || Bn → Sn;
fi;

```

$B_i \rightarrow S_i$ represents a guarded command such that S_i is only executed if logical expression (guard) B_i is true. DO represents the loop statement “do $B \rightarrow S$ od” where S is executed iteratively until guard B is false.

Construct	sp Semantics
$sp(x := e, Q)$	$\equiv (\exists v :: Q_v^x \wedge x = e_v^x)$
$sp(\text{IF}, Q)$	$\equiv sp(S_1, B_1 \wedge Q) \vee \dots \vee sp(S_n, B_n \wedge Q)$
$sp(\text{DO}, Q)$	$\equiv \neg B \wedge (\exists i : 0 \leq i : sp(\text{IF}^i, Q))$
$sp(S_1; S_2, Q)$	$\equiv sp(S_2, sp(S_1, Q))$

Table 1: Strongest Postcondition Semantics for the Dijkstra Guarded Command Language

In the table, the semantics for $sp(x := e, Q)$ states that after the execution of “ $x := e$ ” there exists some value v such that every free occurrence of x in Q is replaced with v and $x = e_v^x$. The semantics for $sp(\text{IF}, Q)$ states that after execution of the if-fi statement, at least one of $sp(S_i, B_i \wedge Q)$ is true. In the case of iteration, denoted $sp(\text{DO}, Q)$, the semantics are that after execution of the loop, the loop guard is false ($\neg B$), and a disjunctive expression describing the effects of iterating the loop some number of times (approximated by the notation IF^k) is true, where $k \geq 0$. Finally, for sequences, $sp(S_1; S_2, Q)$ means that the postcondition for statement S_1 is the precondition for some subsequent statement S_2 .

In this paper we assume that sp is used to derive formal specifications from program code. That is, we use the results of direct code analysis using sp as the entry point to specification abstraction based on specification generalization (Section 4).

Specification Matching

Software reuse is the process of constructing a software system using existing software components. Jeng and Cheng [11] describe the use of a generality operator as the formal basis for identifying reusable components via specification matching. Zaremski and Wing [23] describe several

operators for matching queries to components for software reuse.

In this paper, we describe a technique for identifying abstract behavior in specifications that is based on specification matching. Specifically, we define an approach for *deriving* abstract specifications from as-built specifications by requiring that the derived abstraction and the as-built specification satisfy a matching relation.

3 ABSTRACTION MATCHING

In this section we describe an approach for software reverse engineering that is based on the concept of specification matching. Specifically, this section provides a high-level overview of the approach and describes details about specification libraries that facilitate abstraction.

Approach

Many approaches for reusing software components based on reusable component libraries have been suggested [11, 14, 16, 23]. Given a library of axiomatic (pre- and postcondition) specifications describing software components, these approaches use a *plug-in* or *generality* criteria [11, 23] to identify components in the library that match a *query* specification. The plug-in match is defined as follows:

Definition 1 (Generality (Plug-in) Match [12]) *Let q be a query specification with precondition q_{pre} and postcondition q_{post} and l be a library specification with precondition l_{pre} and postcondition l_{post} . Specifications q and l **match** (denoted by $l \preceq q$) if*

$$(q_{pre} \rightarrow l_{pre}) \wedge (l_{post} \rightarrow q_{post}).$$

Informally, this definition means that the library component l is a refinement (i.e., more specific) than q , or conversely, that q is an abstraction of l . In both interpretations, any program whose behavior is described by q will be satisfied by l and as such, l can be used as an implementation for the query given by q . Many different criteria for matching query specifications with library specifications of software components have been identified [16, 23] and all vary in the degree of component modification required to use a library component as an implementation for a given query.

As stated earlier, our previous investigations have focused on the construction of formal specifications from program code using *sp*. The specifications constructed using *sp* are considered to be at the as-built level of abstraction since the specifications represent a design of the system as it was implemented. Although as-built specifications facilitate traceability between code and specifications, they may be difficult to use for high-level reasoning since they contain an implementation bias. Therefore, a rigorous technique for deriving a more abstract functional specification is desired.

Let \mathcal{I} be a program with specification i such that the precondition is i_{pre} . The corresponding postcondition (denoted i_{post}) can be derived using the strongest postcondition (e.g.,

by using $sp(\mathcal{I}, i_{pre})$) [8]. Let l be a specification in a *specification library* with precondition l_{pre} and postcondition l_{post} . Suppose $i \preceq l$, then l is a generalization or abstraction of i . Conversely, i is a refinement of l . This means that any behavior described by l is satisfied by i and as such, program \mathcal{I} can be used as an implementation for the specification given by l . The following definition summarizes this idea.

Definition 2 (Abstraction Match) *Let \mathcal{I} be a program with specification i such that the corresponding precondition and postcondition are i_{pre} and i_{post} , respectively, and let l be an axiomatic specification with precondition l_{pre} and postcondition l_{post} . A match is an **abstraction match** if $i \preceq l$, so that*

$$(l_{pre} \rightarrow i_{pre}) \wedge (i_{post} \rightarrow l_{post}).$$

As defined, the abstraction match is a dual of the generality (plug-in) match. Accordingly, the abstraction match defines criterion for identifying abstract behavior in pre and postcondition specifications.

Specification Libraries

Specification libraries have been used in the area of automated program construction to describe theories about specific problem domains. In addition, specification libraries have been used as a means for collecting components into reuse libraries. This section describes the use of partial order relations to organize specification libraries and describes several properties that facilitate analysis of specifications based on semantic commonality and difference. In this paper, we assume the validity of certain matching relationships as partial-order relations. For further discussion and proofs of this assumption, see [9].

Library Structure

The convention used in this paper for library specifications, given in Figure 2, is based on the Larch interface language [10] syntax. In this convention, *domainsort* and *rangesort* are the input and output types of a given function, respectively. The **locals** keyword lists the variables defined within the scope of the specification, if applicable. The **requires** keyword is used to indicate the precondition of the given function. The **ensures** keyword describes the postcondition of a given function. Finally, the **modifies** keyword lists the variables that are modified by the function.

```

spec name ( (var: domainsort)* )  $\longrightarrow$  var: rangesort
  locals (var: domainsort)*
  requires precondition
  modifies variables
  ensures postcondition

```

Figure 2: Syntax of Library Specifications

Figure 3 shows a set of specifications, collectively known as “Sqr”, that describe the square root function. The specification *Sqr0* allows negative roots as output whereas *Sqr1* en-

sure that the positive roots are returned. The specifications *Sqr2* and *Sqr3* return undefined values when the input value is less than zero. These two specifications differ in that they allow (*Sqr2*) or disallow (*Sqr3*) negative roots. The specification *Sqr4* returns *root* = 0 when the input is a negative number, and a positive root for positive inputs.

<p>spec <i>Sqr0</i> ($x : \text{real}$) $\rightarrow r : \text{real}$ requires $x \geq 0$ ensures $r^2 = x$</p>	<p>spec <i>Sqr3</i> ($x : \text{real}$) $\rightarrow r : \text{real}$ requires <i>true</i> ensures $(x \geq 0 \wedge (r \geq 0 \wedge r^2 = x)) \vee$ $(x < 0 \wedge r = \text{undefined})$</p>
<p>spec <i>Sqr1</i> ($x : \text{real}$) $\rightarrow r : \text{real}$ requires $x \geq 0$ ensures $r \geq 0 \wedge r^2 = x$</p>	<p>spec <i>Sqr4</i> ($x : \text{real}$) $\rightarrow r : \text{real}$ requires <i>true</i> ensures $(x \geq 0 \wedge r^2 = x) \vee$ $(x < 0 \wedge r = 0)$</p>
<p>spec <i>Sqr2</i> ($x : \text{real}$) $\rightarrow r : \text{real}$ requires <i>true</i> ensures $(x \geq 0 \wedge r^2 = x) \vee$ $(x < 0 \wedge r = \text{undefined})$</p>	<p>spec <i>SqrPos</i> ($x : \text{real}$) $\rightarrow r : \text{real}$ requires $x \geq 0$ ensures $r \geq 0$</p>

Figure 3: Square Root Specification Library “Sqr”

As a partially ordered set on the plug-in relation, the library in Figure 3 has the structure given by the Hasse diagram of Figure 4, where the specification at the head of the arc is more general than the specification at the tail of the arc. As such, *Sqr0* is more general than *Sqr1*, and *Sqr2* is more general than *Sqr3*. The structure of this library suggests that there are three different ways to construct a square root function. The first way requires that the inputs to the function be a positive real number. The second way to construct a square root function is to produce an undefined value when the input is a negative real number. The final way to construct a square root function is to return the value zero when a negative real is used as input.

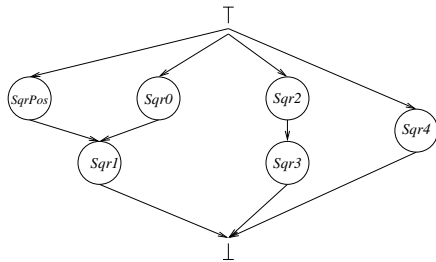


Figure 4: Square Root Library as a partial order

Structuring a library as a partially ordered set has many applications including the fact that it provides a means for partitioning libraries based on behavioral differences as in the example above. In addition, the partial order structure facilitates inserting new specifications into a library and helps increase the efficiency of the retrieval process [12]. It is of interest to determine if the library has certain lattice-like properties [14]. In particular, it is of interest to determine the

least upper bound (lub) since the lub can be used to identify common behavior. Similarly, it is of interest to determine the greatest lower bound (glb) since glb can be used to identify compositional behavior [14]. In the context of these properties, we are interested in deriving abstract specifications from as-built specifications using specification matching criteria as ordering relations. As such, as the techniques described in the next section are applied, the abstract specifications that are derived satisfy the properties listed above by construction.

4 SPECIFICATION GENERALIZATION

Formal approaches to software reuse rely heavily upon the use of specification match criterion, where a search query using formal specifications is used to search a library of components indexed by specifications. Jeng and Cheng addressed the use of formal methods and component libraries to support software reuse [11], and Chen and Cheng address the construction of software based on architectural specifications [4]. A difficulty for all formal approaches to software reuse is the creation of the formal indices. components [7].

Many of the techniques that utilize formal specifications to specify and retrieve reusable components from component libraries attempt to identify candidate components by searching the library for those components that satisfy a specific match criterion. Similarly, as stated by Definition 2, if we have a library specification that is an abstraction match for an as-built specification, then the library specification is a generalization of the as-built specification. However, it is possible that there are no candidates in the specification library that constitute an abstraction match with the as-built specification. In this case, some other technique must be used to *derive* abstractions of the as-built specification rather than via a library search. The primary focus of this paper is the description of such a technique. In this section we describe an approach to reverse engineering based on preserving the partial order relationship between an as-built specification and a derived abstraction of that as-built specification.

Basic Approach

Consider a specification *I* that consists of precondition I_{pre} and postcondition I_{post} . Assuming that the relation \preceq is a partially ordered matching operator, we would like to identify a specification *A* such that $I \preceq A$. That is, we would like to derive a specification *A* that is an abstraction of *I*. In fact, we can derive such a specification by modifying *I* so that we have a specification *I'* that satisfies the relationship that $I \preceq I'$. If, for instance, \preceq is a plug-in match operator, then by either strengthening the precondition I_{pre} , weakening the postcondition I_{post} , or both, we produce a specification *I'* that satisfies the property that $I \preceq I'$. A modification of *I'* to produce a specification *I''* that satisfies the property $I' \preceq I''$ provides another level of abstraction such that $I \preceq I' \preceq I''$.

A likely situation is shown in Figure 5, where a specification

I has been decomposed into several different specifications, each describing a different behavior such that the conjunction of their behaviors is the original specification I . In addition, several of these specifications can be decomposed into other specifications, each at a different level of abstraction. Since the each level of specification is derived such that an abstraction match relation exists between the specifications, the entire set of specifications forms a partial order.

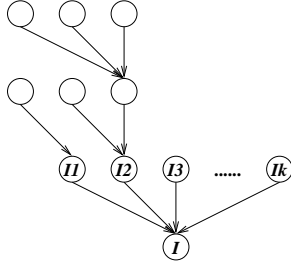


Figure 5: Specification Generalization

Using a brute force approach for specification generalization can result in the construction of an exponential number of specifications, all of which satisfy the partial order constraints of the original specification. For instance, the program in Figure 6 shows a typical bubble sort program with logical annotations contained within the curly braces ‘{’ and ‘}’. The annotations, constructed using the strongest postcondition semantics by a prototype system called AUTOSPEC, use the notation ‘&’ to indicate a logical and (‘^’), ‘exists’ to indicate an existential quantification, and ‘forall’ to indicate a general quantification. In addition, the notation $v.V$ represents the value of a variable v . The specification annotations shown in Figure 6 is the result of applying the strongest postcondition based approach where the precondition is constructed using the signature of the program. The specification for the program after applying simplifying rewrite rules and eliminating superfluous information in the **requires** is as follows:

```

spec BubbleSort ( $a[] : \text{int}, n : \text{int}$ )  $\longrightarrow$   $\text{root} : \text{real}$  (1)
locals  $i, j, t : \text{int}$ 
requires  $n = |a|$ 
modifies  $a$ 
ensures  $(i \geq n) \wedge (j \leq n) \wedge \text{perm}(a_{-1}, a) \wedge$ 
 $(\exists u : 1 \leq u \leq n : (t = a_{-1}[u])) \wedge$ 
 $(\forall k : 1 \leq k < n :$ 
 $(\forall r : k + 1 < r \leq n : a_{-1}[r] \geq a_{-1}[r - 1])),$ 

```

where the **ensures** clause (postcondition) states that after the execution of the program, the variable i is greater than or equal to the size of the array a , the variable j is less than or equal to the size of the array a , the variable t has some value equivalent to some element of the array, all the elements of the array are ordered in ascending fashion, and the final array is a permutation of the original array. Given the five conjuncts in Specification (1), it is possible to construct at least thirty-one different specifications that satisfy the par-

tial order property of the abstraction match operator.

```

program BubbleSort (inputs : int a[]; int n;
                    outputs : int a[]; )
decl
  int i; int j; int y; int x; int t;
lced
begin
  { (((t.V = t_0) ^ ((x.V = x_0) ^ ((y.V = y_0) ^
  ((j.V = j_0) ^ (i.V = i_0)))) ^ ((n.V = n_0) ^ (a.V = a_0))) }
  i := 1;
  { (((((t.V = t_0) ^ ((x.V = x_0) ^ ((y.V = y_0) ^ ((j.V = j_0) ^
  (_cnst1 = i_0)))) ^ ((n.V = n_0) ^ (a.V = a_0))) ^ (i.V = 1)) }
  do
    (i < n) ->
    j := n;
    { (((i.V < n.V) ^ ((i.V = k) ^ (n.V = n_0)) ^ (j.V = n.V)) }
    y := (i + 1);
    { (((((i.V < n.V) ^ ((i.V = k) ^ (n.V = n_0)) ^ (j.V = n.V) ^
    (y.V = (i.V + 1)))) }
    do
      (j > y) ->
      x := (j - 1);
      { (((((j.V > y.V) ^ (((i.V = k) ^ (n.V = n_0)) ^ (j.V = M))) ^
      (x.V = (j.V - 1)) ^
      (a[j.V] = a_j0)) ^ (a[x.V] = a_x0)) }
    if
      (a[j.V] < a[x.V]) ->
      t := a[j];
      { (((((a[j.V] < a[x.V]) ^ (((j.V > y.V) ^ (((i.V = k) ^
      (n.V = n_0)) ^ (j.V = M))) ^ (x.V = (j.V - 1))) ^
      (t.V = a[j.V])) ^ (a[j.V] = a_j0)) ^ (a[x.V] = a_x0)) }
      a[j] := a[x];
      { ((((((_cnst2 < a[x.V]) ^ (((j.V > y.V) ^ (((i.V = k) ^
      (n.V = n_0)) ^ (j.V = M))) ^ (x.V = (j.V - 1))) ^
      (t.V = _cnst2)) ^ (a[j.V] = a_j0)) ^ (_cnst2 = a_j0)) ^
      (a[x.V] = a_x0)) }
      a[x] := t;
      { ((((((((_cnst2 < _cnst3) ^ (((j.V > y.V) ^ (((i.V = k) ^
      (n.V = n_0)) ^ (j.V = M))) ^ (x.V = (j.V - 1))) ^
      (t.V = _cnst2)) ^ (a[j.V] = _cnst3)) ^ (a[x.V] = _cnst2)) ^
      (_cnst2 = a_j0)) ^ (_cnst3 = a_x0)) }
    fi;
    { ((((((((_cnst2 < _cnst3) ^ (((j.V > y.V) ^ (((i.V = k) ^
    (n.V = n_0)) ^ (j.V = M))) ^ (x.V = (j.V - 1))) ^
    (t.V = _cnst2)) ^ (a[j.V] = _cnst3)) ^ (a[x.V] = _cnst2)) ^
    (_cnst2 = a_j0)) ^ (_cnst3 = a_x0)) }
    j := (j - 1);
    { ((((((((_cnst2 < _cnst3) ^ (((j.V > y.V) ^ (((i.V = k) ^
    (n.V = n_0)) ^ (j.V = M))) ^ (x.V = (j.V - 1))) ^ (t.V = _cnst2))
    ^ (a[j.V] = _cnst3)) ^ (a[x.V] = _cnst2)) ^
    (_cnst2 = a_j0)) ^ (_cnst3 = a_x0)) ^ (j.V = (M - 1)) }
  od;
  { (((((j.V = (k + 1)) ^ (x.V = (j.V - 1))) ^
  (forall r : ((k + 1) < r) ^ (r <= n_0) : (a_1[r] >= a_1[r-1])) ^
  (exists u : (((k + 1) < u) ^ (u <= n_0) : (t = a_1[u]))) }
  i := (i + 1);
  { (((((j.V = (k + 1)) ^ (x.V = (j.V - 1))) ^
  (forall r : ((k + 1) < r) ^ (r <= n_0) : (a_1[r] >= a_1[r-1])) ^
  (exists u : (((k + 1) < u) ^ (u <= n_0) : (t = a_1[u]))) ^
  (i.V = (_cnst7 + 1))) }
  od;
  { (((((i.V < n.V) ^ (((i.V > n_0) ^
  (forall v : ((1 <= v) ^ (v < n_0) :
  (forall r : ((v + 1) < r) ^ (r <= n_0) : (a_1[r] >= a_1[r-1]))) ^
  (exists u : (((k + 1) < u) ^ (u <= n_0) : (t = a_1[u]))) ^
  perm(a_1, a))) }
end

```

Figure 6: Bubble Sort Program Annotated by AUTOSPEC

In order to handle the complexity of this situation we make the assumption that the reverse engineering programmer is in control of the abstraction process. To support this process we are developing a support tool called SPECGEN that visually displays the partially ordered sets of specifications that are constructed using the specification generalization technique. In the following sections, we describe several guidelines that

can be used to construct abstractions from a specification. Section 4 discusses the guidelines from the point of view of weakening the postcondition, and Section 4 discusses the guidelines for strengthening a precondition.

Weakening the postcondition

Let I be a specification with precondition I_{pre} and postcondition I_{post} and let I' be a specification such that $I'_{pre} \leftrightarrow I_{pre}$ and $I_{post} \rightarrow I'_{post}$. As such, $I \preceq I'$, since

$$\begin{aligned} & ((I'_{pre} \leftrightarrow I_{pre}) \wedge (I_{post} \rightarrow I'_{post})) \\ & \rightarrow \\ & ((I'_{pre} \rightarrow I_{pre}) \wedge (I_{post} \rightarrow I'_{post})). \end{aligned} \quad (2)$$

Expression (2) provides a basis for deriving abstractions from a specification by weakening a postcondition I_{post} to produce a postcondition I'_{post} . Several options are available for weakening the postcondition including those listed in Table 2, which includes *delete a conjunct*, *add a disjunct*, \wedge to \vee transformation, and \wedge to \rightarrow transformation.

Operation	I_{post}	I'_{post}
Delete a conjunct	$A \wedge B \wedge C$	$A \wedge C$
Add a disjunct	$A \wedge B$	$(A \wedge B) \vee C$
\wedge to \rightarrow	$A \wedge B$	$A \rightarrow B$
\wedge to \vee	$A \wedge B$	$A \vee B$

Table 2: Weakening the postcondition

Delete a conjunct.

Given a specification in conjunctive form (not necessarily a normal form), deletion of a conjunct weakens a specification by removing additional or constraining conditions. For example, consider Figure 7, where the specification $abcde$ represents the ensures clause of the specification in Expression (1). In the Hasse diagram, the vertex label 'xy' represents the logical conjunction $x \wedge y$. Each successive level of abstraction is derived by deleting a conjunct from the lower levels of abstraction. There are a few guidelines that can be used to identify the appropriate conjunct for deletion.

For instance, if a conjunct specifies behavior that is **local** to a procedure and has no impact on the output variables of the system, then that conjunct is a candidate for deletion. Examples include specifications of the value of a loop index or temporary variables. Another guideline is based on **independence** where if a conjunct specifies some behavior that is logically independent of the remaining conjuncts, then that conjunct is a candidate for deletion. As an example, consider the expression $(x = c) \wedge (c = y) \wedge (z = n)$. The conjunct $(z = n)$ is independent of the conjuncts $(x = c)$ and $(c = y)$. Yet another guideline is based on property **preservation** where if a conjunct captures some behavior that must be expressed in the higher level specification, then the remaining conjuncts are candidates for deletion.

These guidelines are by no means comprehensive. Ulti-

mately, a maintenance engineer using this approach must decide whether to delete a specific conjunct in a specification.

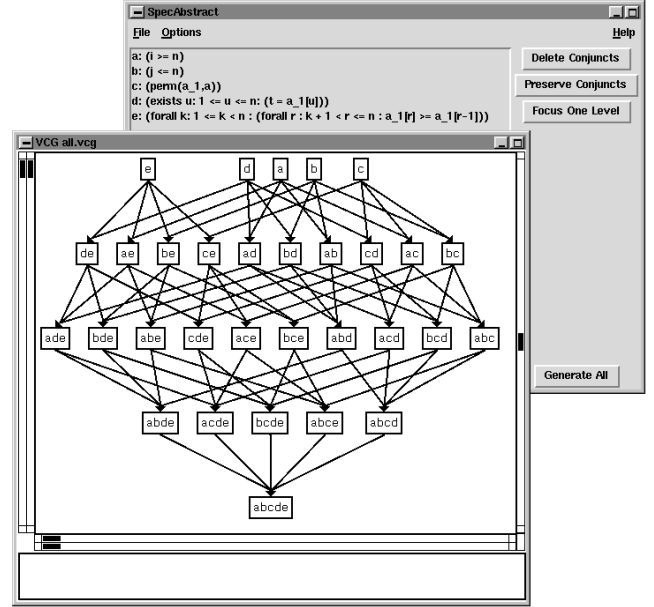


Figure 7: Bubble Sort Specification Brute Force Abstraction

Add a disjunct.

Given a specification in any form, adding a disjunct weakens a specification by generalizing or increasing the scope of the specification. The addition of a disjunct should be used in very few instances since the new disjunct potentially introduces superfluous behavior that may not be reflected in the original system.

Conjunction to implication or disjunction transformations.

Given a specification in conjunctive form (not necessarily a normal form), transformation of the conjunction to an implication or disjunction provides a logical weakening of the specification and facilitates manipulation of the specification using several standard equivalence transformations. Our ongoing investigations include determining the usefulness of these transformation techniques to derive specification abstractions.

Strengthening the precondition

Let I be a specification with precondition I_{pre} and postcondition I_{post} and let I' be a specification such that $I'_{pre} \rightarrow I_{pre}$ and $I_{post} \leftrightarrow I'_{post}$. As such, $I \preceq I'$, since

$$\begin{aligned} & ((I'_{pre} \rightarrow I_{pre}) \wedge (I_{post} \leftrightarrow I'_{post})) \\ & \rightarrow \\ & ((I'_{pre} \rightarrow I_{pre}) \wedge (I_{post} \rightarrow I'_{post})). \end{aligned} \quad (3)$$

Expression (3) provides a basis for deriving abstractions from a specification by strengthening a precondition I_{pre} to produce a precondition I'_{pre} . Weakening a postcondition has many advantages over strengthening a precondition in the context of deriving abstractions from specifications. The pri-

mary advantage is a consequence of the reverse engineering activity in that we are interested in deriving a specification of the behavior of a program. This behavior is captured in the specification of the postcondition rather than the precondition. The utility of strengthening the precondition is that it provides a mechanism for identifying a narrower set of conditions that can be used to constrain the domain of input. The available techniques for strengthening the precondition include *adding a conjunct* and *deleting a disjunct*.

Add a conjunct.

Given a specification in a conjunctive (not necessarily normal) form, adding a conjunct to the precondition provides further conditions that are required in order for the specification to achieve the desired behavior.

Delete a disjunct.

Given a specification in any form, deleting a disjunct will make the precondition more specialized (e.g., less general) in the initial conditions required to satisfy a behavior.

Example

Consider once again the example in Figure 6 and the corresponding postcondition specification in Expression (1). In this section we focus on constructing an abstraction of the specification by weakening the postcondition. Since the specification is in a conjunctive normal form, it is appropriate to use the *delete a conjunct* strategy to construct an abstraction. In a completely brute force approach we would derive four abstractions for each of the five produced in the first step. However, we advocate a user-driven process that relies on a user to guide the direction of the abstraction steps. Figure 7 depicts the brute force application of the delete a conjunct strategy, where the expression “*abcde*” at the bottom of the graph represents the specification of Expression (1), where “*a*” represents the conjunct $(i \geq n)$, “*b*” represents the conjunct $(j \leq n)$, “*c*” represents the conjunct $perm(a_1, a)$, “*d*” represents the conjunct $(\exists u : 1 \leq u \leq n : (t = a_1[u]))$, and “*e*” represents the conjunct $(\forall k : 1 \leq k < n : (\forall r : k + 1 < r \leq n : a_1[r] \geq a_1[r - 1]))$.

In the first step, the “*a*” conjunct can be deleted since the “*a*” conjunct involves a specification of the value of an iteration variable. Deleting one conjunct from the specification “*bcde*” results in four different specifications. Using the same reasoning as in the previous step, we consider only the specification that excludes the conjunct “*b*”. Figure 8 shows the partially ordered set of specifications that result from deleting “*a*” and “*b*” where the resulting specification is “*cde*” which states that the output array is a permutation of the input array, that the variable *t* takes the value of some element of the array, and the array is ordered in increasing value. At this point, three abstractions are possible. However, the conjunct $(\exists u : 1 \leq u \leq n : (t = a_1[u]))$ specifies information about the temporary variable *t*, and as such we consider only the specification *ce*, which is equivalent to:

$$perm(a_1, a) \wedge (\forall k : 1 \leq k < n : (\forall r : k + 1 < r \leq n : a_1[r] \geq a_1[r - 1])).$$

This specification states that after execution of the program, the output array is a permutation of the input array, and that the array is ordered in increasing value.

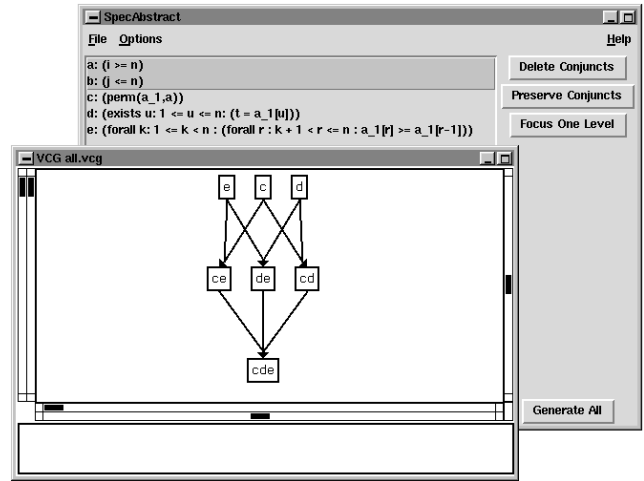


Figure 8: Specification after deletion of “*a*” and “*b*”

By focusing attention on a few conjuncts, the complexity of the task of constructing specification abstractions can be reduced since many of the other possible abstractions for the original as-built specification can be removed from consideration. In addition, by using a few simple support tools, the difficulty of deriving the abstractions can be greatly reduced.

5 APPLICATION TO A NASA JPL GROUND-BASED FLIGHT SYSTEM

In our previous investigations we described a technique for analyzing C programs using the strongest postcondition predicate transformer. In this section we present a case study that applies the *sp* technique for C programs to a module from a ground-based mission control system used by the NASA Jet Propulsion Laboratory. The system is responsible for the translation of user commands into appropriate spacecraft mnemonics, enabling users to modify spacecraft mission operations. This particular module takes a sequence of elements from a file and returns an index to a subsequence of elements specified by begin and end indices.

Code Analysis

First Code Sequence.

One code sequence that was analyzed appears as follows:

```
if (!skip_gcmd_sfdu(fd, L2))
{
    inform_user(
        "line %d: copy failed: bad SFDU header (%s)",
        body_lineno, file);
    dontoutput = 1;
    close(fd);
    if (params->cmdcnt1) master_unlock();
    return(NULL);
}
```

The purpose for this code sequence is to abort processing if

the file header is corrupted. The precondition for the block is

```
(fd >= 0 & fd = FH0 &
begin = B0 & end = E0 & file .> F0),
```

which makes assertions about the initial values of several variables and pointers, where the & is the logical connective ‘^’. The specification states that fd has the initial value FH0, and that the value is greater than or equal to 0. The specification also states that the variables begin and end have the values B0 and E0, respectively. Finally, the specification states that the pointer file points to some object F0.

The following annotation describes the behavior of the code when the conditional path is taken in the case that skip_gcmd_sfdu evaluates to zero:

```
(params->cmdcntl != 0 & master_unlocked() &
  closed(fd) & dontoutput = 1 &
  skip_gcmd_sfdu(fd, L2) = 0 &
  fd >= 0 & fd = FH0 & begin = B0 &
  end = E0 & file .> F0) |
(params->cmdcntl = 0 & closed(fd) &
  dontoutput = 1 & fd >= 0 &
  skip_gcmd_sfdu(fd, L2) = 0 & fd = FH0 &
  begin = B0 & end = E0 & file .> F0),
```

which is equivalent to

```
((params->cmdcntl != 0 & master_unlocked()) |
  params->cmdcntl = 0 ) &
closed(fd) & dontoutput = 1 &
  skip_gcmd_sfdu(fd, L2) = 0
  & fd >= 0 & fd = FH0 & begin = B0
  & end = E0 & file .> F0 .
```

This specification states that in addition to the precondition being true, the file FH0 is closed, the variable dontoutput is set to 1, and depending on whether the params->cmdcntl has the value 0, the master key is unlocked. In this system, processing is regarded as having failed whenever the variable dontoutput is set to a non-zero value. This specification recurs throughout this code when certain failure conditions are met.

The following postcondition annotation asserts:

```
skip_gcmd_sfdu(fd, L2) != 0 & fd >= 0 &
fd = FH0 & begin = B0 & end = E0 & file .> F0 ,
```

which states that in addition to the precondition being true, that the function skip_gcmd_sfdu evaluates to a non-zero value. This specification is reasonable since the body of the statement in question ends with a return statement. As such, the program only proceeds past the conditional statement if skip_gcmd_sfdu evaluates to a non-zero value.

Second Code Sequence.

Another interesting sequence of code appears in Figure 9. One of the activities that can be performed is to analyze the postcondition at lines 125-134 using the specification generalization technique described in Section 4. First, we can

rewrite the specification into an equivalent form by factoring terms so that the specification appears as follows:

```
((E0 = -1 & end = gcmd_hdr.elem_count) |
  (end <= gcmd_hdr.elem_count & end != -1 &
  end = E0)) & params->sc = gcmd_hdr.SC &      (4)
  get_gcmd_hdr(fd, gcmd_hdr) != 0 &
  skip_gcmd_sfdu(fd, L2) != 0 & fd >= 0 &
  fd = FH0 & begin = B0 & file .> F0 .
```

This specification states that the constant E0 is equal to -1 and end = gcmd_hdr.elem_count or that end = E0, end <= gcmd_hdr.elem_count, and end != -1. In addition, several conditions regarding the input file are true as well as conditions that describe the input file. At this point the specification is in a form suitable to apply the **delete a conjunct** strategy. Figure 10 shows the possible abstractions for the specification when we delete the file related conjuncts. Successive application of the strategy leads to the abstraction of the behavior described by the previous annotation. This specification corresponds to the specification ‘a’ in the Hasse diagram in Figure 10 and appears as follows:

```
((E0 = -1 & end = gcmd_hdr.elem_count) |
  (end <= gcmd_hdr.elem_count & end != -1 &
  end = E0)),
```

which states that E0 = -1 and the variable end has the value gcmd_hdr.elem_count, or, end = E0, end != -1 and end <= gcmd_hdr.elem_count. Essentially, this states that if this point in the program has been reached, the variable end has a value that is less than or equal to gcmd_hdr.elem_count and not equal to -1. As such, behavior of the program when end < -1 is unspecified and serves as the potential entry point for more detailed analysis of the program.

Third Code Sequence.

The final annotation that is of interest is found in Figure 11. The annotation is shown after a simplification step that factors conjuncts from a disjunction has been performed.

Informally, this specification makes assertions about a chain of elements and the relationship between the requested subsequence of elements and the elements read from a file. The following specification abstraction can be derived by applying the delete a conjunct strategy to this annotation by focusing on a few specific conjuncts:

```
(forall k : 1 <= k < begin : freed(elem_k)) & (5)
(forall k : end < k < gcmd_hdr.elem_count :
  freed(elem_k)) &
elem_end->next .> NULL &
orig_elem .> coset(elem_begin) &
(forall k : begin <= k < end :
  elem_k->next .> coset(elem_k+1) & zeroed(elem_k))
```

which states that all the elements outside the bounds specified by the begin and end indices have been freed and that the pointer orig_elem points to the same object that

```

108./*AS (params->sc = gcmd_hdr.SC &
109.  get_gcmd_hdr(fd, gcmd_hdr) != 0 & file .> F0 &
110.  fd >= 0 & fd = FH0 & begin = B0 & end = E0 &
111.  skip_gcmd_sfdu(fd, L2) != 0)AS*/
112./* make sure the file has enough elements */
113.if (end == -1)
114.  end = gcmd_hdr.elem_count;
115.else if (end > gcmd_hdr.elem_count)
116.{
117.  inform_user("line %d: copy: not enough elements \
118.            in GCMD file (%s)",body_lineno, file);
119.  dontoutput = 1;
120.  close(fd);
121.  if (params->cmdcnt1) master_unlock();
122.  return(NULL);
123.}
124.
125./*AS
126.(E0 = -1 & end = gcmd_hdr.elem_count &
127.  params->sc = gcmd_hdr.SC & E0 = E0 & file .> F0 &
128.  get_gcmd_hdr(fd, gcmd_hdr) != 0 & fd = FH0 &
129.  skip_gcmd_sfdu(fd, L2) != 0 & fd >= 0 & begin = B0 )
130.|
131.(end <= gcmd_hdr.elem_count & end != -1 & begin = B0
132. & params->sc = gcmd_hdr.SC & end = E0 & file .> F0
133. & get_gcmd_hdr(fd, gcmd_hdr) != 0 & fd = FH0 &
134. skip_gcmd_sfdu(fd, L2) != 0 & fd >= 0 ) AS*/
135.

```

Figure 9: Code Sequence: Lines 108–135

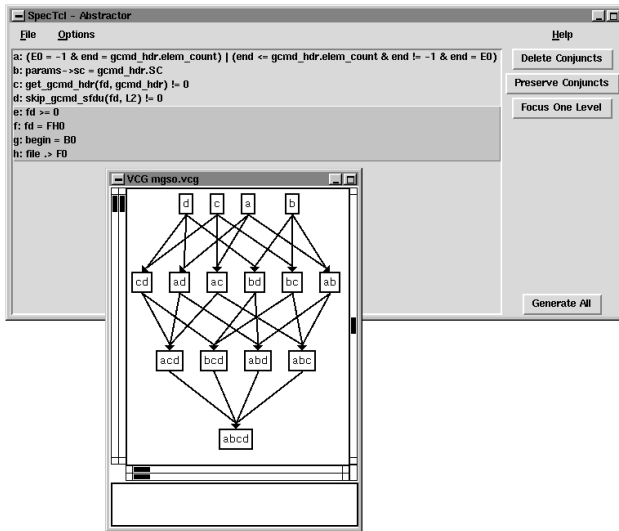


Figure 10: Annotation Abstractions

elem_begin points to and that all the elements within the begin and end bounds form a chain.

Discussion

The analysis of the code described in this section has led to several observations that empirically validate the appropriateness of the delete a conjunct strategy for specification generalization of *sp* specifications. First, the specifications that are constructed using *sp* are conjunctive in nature due to the semantics of the assignment statement. As such, application of the delete a conjunct strategy facilitates the analysis of specifications of program code by decomposing those specifications into smaller, more manageable pieces. Second,

```

closed(fd) &
(forall k:end < k < gcmd_hdr.elem_count:freed(elem_k) &
ep .> coset(elem_gcmd_hdr.elem_count) &
elem .> coset(ep) &
elem_end->next .> NULL &
orig_elem .> coset(elem_begin) &
checksum_gcmd_chain(gcmd_hdr, Obj0cnst1) = 0 &
elem_gcmd_hdr.elem_count .> NULL &
(forall k : 1 <= k < begin : freed(elem_k) &
(forall k : begin <= k < end :
  elem_k->next .> coset(elem_k+1) & zeroed(elem_k)) &
(E0 = -1 & end = gcmd_hdr.elem_count) |
(end <= gcmd_hdr.elem_count & end != -1 & end = E0)) &
params->sc = gcmd_hdr.SC &
get_gcmd_hdr(fd, gcmd_hdr) != 0 &
skip_gcmd_sfdu(fd, L2) != 0 &
fd >= 0 &
fd = FH0 &
begin = B0 &
file .> F0 .

```

Figure 11: Code annotation

analysis of annotations that occur within the code (as opposed to only analyzing the final postcondition) is an important activity for understanding the behavior of programs. As an example, our analysis of the code from lines 108–135 of the program facilitated the identification of possible faults in the behavior of the program that can be caused by negative inputs to the program. Finally, although the reverse engineered specifications describe logical abstractions, some mechanism must be provided in order to describe the abstractions in the form of a simpler “human” abstraction. For instance, instead of providing the specification in Expression (5) to a user, it would be desirable to state that since the procedure returns a subsequence of a list of elements, that the abstract behavior corresponds to a list subsequence *cliché* or *plan* [19], where a *plan* describes common or canonical program behavior.

6 RELATED WORK

Mili *et al.* describe an approach for structuring component libraries using refinement orderings [14]. Their approach uses relational specifications as the formalism for describing software components, and structures libraries using relational definitions of refinement. Our approach incorporates their ideas on the structure of libraries using partial order relations although our focus is on pre and postcondition specifications, rather than relational specifications. In addition, our primary goal is to use partial order matching operators to generalize specifications for purposes of reverse engineering.

Other approaches to reverse engineering focus on the construction of specifications, both informal and formal, and are based on the identification of *plans* [17] formal methods [2, 21], and transformation of programs into specifications [1]. Baxter and Mehlich [1] suggest an approach to reverse engineering using “backward transformation” where a series of transformations (semantic preserving rewrite rules), similar to those used in forward transformation, are used in an inverse manner. The use of a library is extensive in this approach where the contents of the library are semantic preserving transformations. Ward [21] also advocates a transformational approach while the REDO project focused

on translation and transformation of programs into specifications. In our approach, we assume that abstract behavior is derived by preserving an abstraction match relation between generalized and as-built specifications. As such, we do not rely on the existence of a domain library to provide specification matches.

7 CONCLUSIONS AND FUTURE WORK

Specification libraries have been used extensively and with relative success for code derivation [20]. In addition, formal specification libraries are the basis for many software component reuse approaches [23, 12].

When used alone, formal approaches to reverse engineering based on the use of strongest postcondition produce as-built specifications that suffer from an implementation bias. While this implementation bias does provide a degree of traceability, they are difficult to use for high-level reasoning and understanding. Using the specification match based approach, functional abstractions can be derived from the as-built specifications that are constructed using strongest postcondition. Furthermore, as demonstrated with our previous investigations [7], an integrated approach that combines the strongest postcondition technique with software component retrieval technology can be used to facilitate software component reuse, where the combination of reverse engineering and component retrieval techniques are used to populate reusable component libraries.

In order to further address the feasibility of such an approach we are currently constructing a quaternion specification library based on a PVS [15] specification library developed by NASA. In addition, we are analyzing a software library written in C that implements quaternion operations in order to investigate the use of reverse engineering as a means for facilitating software reuse.

REFERENCES

- [1] I. D. Baxter and M. Mehlich. Reverse Engineering is Reverse Forward Engineering. In *Proceedings of the Fourth IEEE Working Conference on Reverse Engineering*. IEEE, October 1997.
- [2] J. Bowen, P. Breuer, and K. Lano. The REDO Project: Final Report. Technical Report PRG-TR-23-91, Oxford University, 1991.
- [3] E. J. Byrne. A Conceptual Foundation for Software Re-engineering. In *Proceedings for the Conference on Software Maintenance*, pages 226–235. IEEE, 1992.
- [4] Y. Chen and B. H. C. Cheng. Facilitating an automated approach to architecture-based software reuse. In *Proceedings of the 12th International Conference on Automated Software Engineering*, 1997.
- [5] E. J. Chikofsky and J. H. Cross. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, January 1990.
- [6] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.
- [7] G. C. Gannod, Y. Chen, and B. H. C. Cheng. An Automated Approach to Supporting Software Reuse via Reverse Engineering. In *Proceedings of the 13th Automated Software Engineering Conference*. IEEE, 1998.
- [8] G. C. Gannod and B. H. C. Cheng. Strongest Postcondition as the Formal Basis for Reverse Engineering. *Journal of Automated Software Engineering*, 3(1/2):139–164, June 1996. A preliminary version appeared in the *Proceedings for the IEEE Second Working Conference on Reverse Engineering*, July 1995.
- [9] G. C. Gannod and B. H. C. Cheng. A specification matching based approach to reverse engineering. Technical Report MSUCPS-TR98-15, Michigan State University, April 1998.
- [10] J. Guttag and J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.
- [11] J. Jeng and B. H. C. Cheng. Using Automated Reasoning Techniques to Determine Software Reuse. *International Journal of Software Engineering and Knowledge Engineering*, 2(4):523–546, December 1992.
- [12] J.-J. Jeng and B. H. C. Cheng. Specification Matching for Software Reuse: A Foundation. In *Proceedings of the ACM Symposium on Software Reuse*, pages 97–105, 1995.
- [13] N. G. Leveson and C. S. Turner. An Investigation of the Therac-25 Accidents. *IEEE Computer*, 26(7):18–41, July 1993.
- [14] A. Mili, R. Mili, and R. T. Mittermeir. Storing and Retrieving Software Components: A Refinement Based System. *IEEE Transactions on Software Engineering*, 23(7), July 1997.
- [15] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.
- [16] J. Penix and P. Alexander. Toward Automated Component Adaptation. In *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering*, June 1997.
- [17] A. Quilici. A Memory-Based Approach to Recognizing Program Plans. *Communications of the ACM*, 37(5):84–93, May 1994.
- [18] Report by the Inquiry Board. ARIANE 5 Flight 501 Failure. Technical report, European Space Agency, 1996. J.L. Lions, Chairman of the Board.
- [19] C. Rich and R. C. Waters. *The Programmer's Apprentice*. ACM-Press, 1990.
- [20] D. R. Smith and E. A. Parra. Transformational Approach to Transportation Scheduling. In *Proceedings of the 8th Knowledge-Based Software Engineering Conference*, pages 60–68, Sept 1993.
- [21] M. Ward. Abstracting a Specification from Code. *Journal of Software Maintenance: Research and Practice*, 5:101–122, 1993.
- [22] J. M. Wing. A Specifier's Introduction to Formal Methods. *IEEE Computer*, 23(9):8–24, September 1990.
- [23] A. M. Zaremski and J. M. Wing. Specification Matching of Software Components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, 1997.