This is to certify that the

thesis entitled

The Application of Formal Methods to the Reverse Engineering of Imperative Program Code

presented by

Gerald C. Gannod

has been accepted towards fulfillment
of the requirements for

M.S. degree in Computer Science

Major professor

Date 04/07/94

The Application of Formal Methods to the Reverse Engineering of Imperative Program Code

By

Gerald C. Gannod

A THESIS

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

MASTER OF SCIENCE

Computer Science Department

May 1994

ABSTRACT

The Application of Formal Methods to the Reverse Engineering of Imperative Program Code

By

Gerald C. Gannod

Formal methods in software development provide many benefits in the forward engineering aspect of software development. One of the advantages of using formal methods in software development is that the formal notations are precise, verifiable, and facilitate automated processing. Reverse engineering is the process of constructing high level representations from lower level instantiations of an existing system. There are two main objectives that the research described in this thesis covers. First, a new approach to the construction of formal specifications from program code has been developed. To this end, a propagational model of translation has been developed that provides the framework for performing reverse engineering using the direct translation of programming constructs into corresponding formal specifications. Using this framework, a number of translation rules for constructing formal specifications for programming primitives, including assignment, alternation, iteration, sequence, and procedure call, have been developed. The second objective of this research involves the movement of procedural software towards object-orientation. We have developed an approach that uses clustering and restructuring of formal specifications at the procedural level to identify objects embedded in programs.

To G.T. and R.C., I always thought I'd see you again.

ACKNOWLEDGMENTS

I'd like to thank my parents, siblings, and friends for their understanding and support. Well deserved recognition goes to the Committee for Institutional Cooperation Summer Research Opportunity Program and the McNair Post Baccalaureate Research Program at Michigan State University (specifically, Mary Lee and Shawn). Without that program, none of this would have been possible. I would also like to thank Bob Bourdeau and the rest of the students in the software bullpen for their insights into this work. My utmost respect and gratitude goes to Dr. Cheng, my advisor and mentor, for helping me find the way. Finally, to the dead (and dread) poets and jazz musicians of the world, may you always say the right verse and play the right note.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

1.1 Motivation

The demand for software correctness becomes more evident when accidents, sometimes fatal, are due to software errors. For example, recently it was reported that the software of a medical diagnostic system was the major source of a number of potentially fatal doses of radiation [5]. Other problems caused by software failure have been well documented and with the change in laws concerning liability [6], the need to reduce the number of problems due to software increases.

Formal methods in software development provide many benefits in the forward engineering aspect of software development [7, 8, 9, 10, 11]. One of the advantages of using formal methods in software development is that the formal notations are precise, verifiable, and facilitate automated processing [12]. Reverse Engineering is the process of constructing high level representations from lower level instantiations of an existing system. One method for introducing formal methods, and therefore taking advantage of the benefits of formal methods, is through the reverse engineering of existing program code into formal specifications.

1.2 Research Contributions

There are two main objectives that the research described in this thesis covers. First, a new approach to the construction of formal specifications from program code has been developed. To this end, a new propagational model of translation has been developed that provides the framework for performing reverse engineering using the direct translation of programming constructs into corresponding formal specifications. Using this framework, a number of translation rules for constructing formal specifications for programming primitives, including assignment, alternation, iteration, sequence, and procedure call, have been developed. Finally, this translational framework is applied to the imperative programming paradigm, using Pascal as a model [13].

The second objective of this research involves the movement of procedural software towards object-orientation. Other projects have focused on source code clustering [14] and functional abstraction [4]. We have developed an approach that uses clustering and restructuring of formal specifications at the procedural level to identify objects embedded in programs [15].

1.3 Organization

The remainder of this thesis is organized as follows. Chapter 2 provides background information pertinent to the area of software maintenance and the support of reverse engineering using formal methods. The research contributions are described in detail in Chapters 3 through 7. Chapter 3 describes the translational propagation model for the construction of formal specifications from program code. Chapters 4 and 5 discuss the translation rules for the assignment, alternation, iteration, sequence, and procedure call programming primitives. Chapter 6 discusses the application of the translational approach to the imperative programming paradigm. The method for identifying objects in programs using formal specifications is described in Chapter 7.

Chapter 8 presents related work in the area of reverse engineering, and Chapter 9 draws conclusions and discusses future investigations.

CHAPTER 2

Background

2.1 Software Engineering

The following definition of software engineering has been offered by Fritz Bauer [16]:

> The establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.

In 1968 and 1969, workshops were held in Garmisch, West Germany and Rome, Italy with the intent of addressing the growing problems associated with the construction of computer software. The phrase "software crisis" was coined to describe the increasing difficulty associated with the development of software. It was recognized that processes for developing and managing software were desperately needed in order to respond to the software crisis. Since then, a number of software life-cycles have been developed with the intent of producing "software that is reliable and works efficiently on real machines".

The classical life-cycle, better known as the "waterfall" life-cycle, provides a systematic process for developing software. It involves five phases including requirements analysis, design, implementation, testing, and maintenance [17].

The *requirements phase* is used to define the needs of a customer or user. The main task is to determine the required function of the software system based on information gathered about the problem domain. The *design phase* is used to refine the information gathered in the requirements phase by focusing on issues of software architecture, behavior, and presentation. The *implementation phase* involves the translation of the design into a machine executable form. Here the issues include efficiency considerations, low level algorithmic details, and language. The purpose of the *testing phase* is to verify that the software performs as expected. Testing strategies include black-box testing, white-box testing, and branch testing. The *maintenance phase* encompasses all operations that occur after the release of the software. Redesign, modification, and enhancement are all operations resulting from the maintenance of a system.

Other types of life-cycles have been created including the *reuse, prototype,* and *spiral* models [17]. Each has advantages over the classic life-cycle, but in general incorporate, to some degree, the phases defined by the waterfall model.

2.2 Software Maintenance

Software maintenance has long been a problem faced by software professionals, where the average age of software is between 10 to 15 years old [18]. With the development of new architectures and improvements in programming methods and languages, including formal methods in software development and object-oriented programming, there is a strong motivation to reverse engineer and re-engineer existing program code in order to preserve functionality, while exploiting the latest technology.

Reverse engineering of program code is the process of constructing a higher level abstraction of an implementation in order to facilitate the understanding of a system that may be in a "legacy" or "geriatric" state. Re-engineering is the process of examining, understanding, and altering a system with the intent of implementing the

system in a new form [19]. The benefits offered by re-engineering versus developing software from the original requirements is considered to be a solution for handling existing (legacy) code because much of the functionality of the existing software has been achieved over a period of time and must be preserved for many reasons, including providing continuity to current users of the software [20].

One of the most difficult aspects of re-engineering is the recognition of the functionality of existing programs. This step in re-engineering is known as reverse engineering. Identifying design decisions, intended use, and domain specific details are often the main obstacles to successfully re-engineering a system.

Several terms are frequently used in the discussion of re-engineering. [19]. *Forward engineering* is the process of developing a system by moving from high level abstract specifications to detailed, implementation-specific manifestations [19]. The explicit use of the word "forward" is used to contrast the process with *Reverse engineering*, the process of analyzing a system in order to identify system components, component relationships, and intended behavior [19]. *Restructuring* is the process of creating a logically equivalent system at the same level of abstraction [19]. This process does not require a semantic understanding of the system and is best characterized by the act of transforming unstructured code into structured code. A diagram depicting the relationships between forward engineering, reverse engineering, restructuring, and re-engineering is shown in Figure 2.1.

2.2.1 A Model of Re-engineering

Re-engineering can be described in terms of a model based on three concepts that are defined as follows [2]:

> *Refinement* is the gradual decrease in the abstraction level of a system representation and is caused by the successive replacement of existing

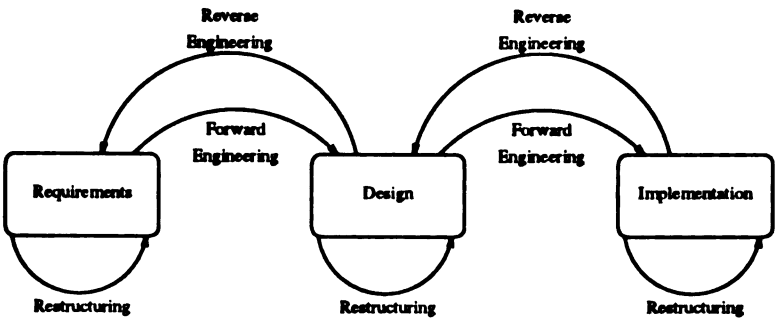


Figure 2.1. Relationship between re-engineering terms

system information with more detailed information.

Abstraction is the gradual increase in the abstraction level of a system representation and is created by the successive replacement of existing detailed information with information that is more abstract. Abstraction produces a representation that emphasizes certain system characteristics by suppressing information about others.

Alteration is the incorporation of one or more changes to a system representation without changing the degree of abstraction. Alteration includes the addition, deletion, and modification of information.

Figure 2.2 depicts the three concepts of refinement, abstraction, and alteration within a model of re-engineering, where the left triangle represents the original system, and the right triangle represents the new system. Abstraction is shown as the arrow angled upward and corresponds to reverse engineering. The arrow angled downward depicts refinement and corresponds to forward engineering. The horizontal arrow shows alteration and corresponds to restructuring.

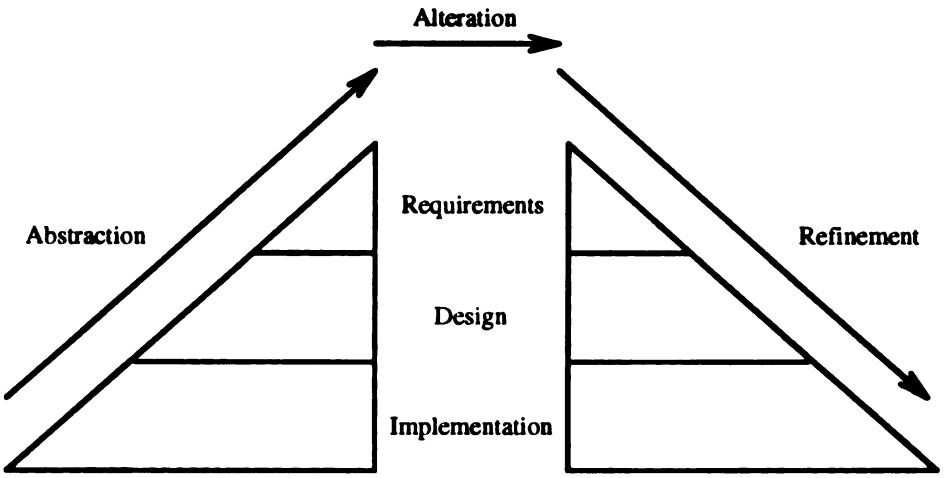


Figure 2.2. Graphical Depiction of a Re-engineering Model [2]

2.3 Applications of Reverse Engineering

Reverse engineering has many areas of application, including *redocumentation* and *design recovery*. Redocumentation is the creation or revision of a semantically equivalent representation within the same relative abstraction level [19]. The purpose of redocumentation is to recover or revise documentation that is missing or obsolete. Often redocumentation is in the form of data flow and control flow diagrams.

Design recovery is the process in which domain knowledge, external information, and deduction or fuzzy reasoning are added to the observations of a software system to identify meaningful higher level abstractions beyond those obtained directly by examining the system itself [19]. In addition to providing the abstractions for the re-engineering processes of rework and replacement, design recovery can facilitate the reuse of software components by providing the details of functionality.

2.4 Formal Methods

Although the waterfall development life-cycle provides a structured process for developing software, the design methodologies that support the life-cycle (i.e., Structured Analysis and Design) have shortcomings in that they have the potential for introducing ambiguity, inconsistency, and incompleteness in designs and implementations. Formal methods used in software development are systematic techniques for specifying, developing, and verifying computer software. Formal methods aid in eliminating the properties of ambiguity, inconsistency, and incompleteness of a system through the use of a well-defined specification language with a set of well-defined inference rules that can be used to reason about the specification. The remainder of this section describes the notation used throughout the thesis and defines some key aspects of program semantics.

2.4.1 Propositional Logic

A proposition is a logical expression that represents a *truth value* of *true (T)* or *false (F)*. Operators consist of conjunction ($\wedge$), disjunction ($\vee$), implication ($\Rightarrow$), and equality ($=$) Propositional logic can be described using the following rules [1]:

1. T and F are propositions
2. An identifier is a proposition. (An identifier is a sequence of one or more digits and letters, the first of which is a letter.)
3. If b is a proposition, then so is $(\neg b)$.
4. If b and c are propositions, then so are $(b \wedge c)$, $(b \vee c)$, $(b \Rightarrow c)$, and $(b = c)$.

Table 2.1 gives the BNF grammar for propositions using precedence rules to eliminate parenthesized expressions [1].

⟨*proposition*⟩	::=	⟨*imp-expr*⟩
	|	⟨*proposition*⟩ = ⟨*imp-expr*⟩
⟨*imp-expr*⟩	::=	⟨*expr*⟩
	|	⟨*imp-expr*⟩ ⇒ ⟨*expr*⟩
⟨*expr*⟩	::=	⟨*term*⟩
	|	⟨*expr*⟩ ∨ ⟨*term*⟩
⟨*term*⟩	::=	⟨*factor*⟩
	|	⟨*term*⟩ ∧ ⟨*factor* ⟩
⟨*factor*⟩	::=	¬⟨*factor*⟩
	|	(⟨*proposition*⟩)
	|	*T*
	|	*F*
	|	⟨*identifier*⟩

Table 2.1. BNF Grammar for Propositional Logic [1]

2.4.2 Predicate Logic

Propositional logic can be extended by replacing identifiers in a proposition with any expression (termed an *atomic expression*) that can be evaluated to *true* or *false* and by allowing the use of existential (∃) and general (∀) quantification. This extension is known as *predicate logic.* Atomic expressions are made up of four different types of symbols: *individual symbols* or *constants*, *variables*, *functions*, and *predicates.* A predicate is a mapping from a list of constants to the value *true* or *false.*

Existential quantification can be defined in the following manner. Given the expression

$$E_j \vee E_{j+1} \vee \ldots \vee E_k,$$

where j and k are integers, and E_i is a predicate, the existential quantification operator (∃) can be used to abbreviate the expression in the following way

$$(\exists i : j \leq i \leq k : E_i).$$

General quantification ($\forall$) can be expressed in a similar fashion. Given the expression

$$E_j \wedge E_{j+1} \wedge \ldots \wedge E_k,$$

general quantification has the following form

$$(\forall i : j \leq i \leq k : E_i),$$

where j and k are integers and E_i is a predicate. General quantification can be expressed in terms of existential quantification, and vice versa, as is shown below

$$(\forall i : R_i : E_i) \equiv \neg(\exists i : R_i : \neg E_i),$$

where R_i is a predicate that describes the range, E_i is an arbitrary predicate, and i is the quantified variable

2.4.3 Program Semantics

The notation $Q \{ S \} R$ was originally introduced by Hoare [21] to indicate a partial correctness model of execution, where given that logical condition Q holds, if the execution of statement S terminates, then logical condition R will hold. A rearrangement of the braces in the form of $\{ Q \} S \{ R \}$, in contrast, represents a total correctness model of execution. That is, if logical condition Q holds, then statement S is guaranteed to terminate with logical condition R true. If we consider a state space for computation then, in terms of some predicate R, we can partition computations into the following three mutually exclusive classes [22]:

eternal: All computations that fail to terminate

finally R**:** All computations terminating in a final state with R true

finally $\neg R$: All computations terminating in a final state with $\neg R$ true.

An analogous characterization of the computation state space in terms of the initial state of a program can be given as follows [22]:

initially Q: All computations starting in an initial state with Q true.

initially $\neg Q$: All computations starting in an initial state with $\neg Q$ true.

A *precondition* describes the initial state of a program, and a *postcondition* describes the final state. Given a statement S and a postcondition R, the following predicates can be defined in terms of the characterization of the computation state space [22]:

$wp(S, R)$	The set of all states in which the computation under control of S belongs to the class "finally R".
$wlp(S, R)$	The set of all states in which the computation under control of S belongs to the class "eternal" or "finally R".

That is, the *weakest precondition* $wp(S, R)$ describes the set of all states in which the statement S can begin execution and terminate with R true, and the *weakest liberal precondition* $wlp(S, R)$ is the set of all states in which the statement S can begin execution and establish R as true if S terminates. In this respect, $wp(S, R)$ establishes the total correctness of S, and $wlp(S, R)$ establishes the partial correctness of S. The *wp* and *wlp* are called predicate transformers because they take predicate R and, using the properties listed in Table 2.2, produce a new predicate. An interesting

$wp(S, A)$	$\equiv wp(S, true) \wedge wlp(S, A)$
$wp(S, A)$	$\Rightarrow \neg wlp(S, \neg A)$
$wp(S, false)$	$\equiv false$
$wp(S, A \wedge B)$	$\equiv wp(S, A) \wedge wp(S, B)$
$wp(S, A \vee B)$	$\Rightarrow wp(S, A) \vee wp(S, B)$
$wp(S, A \rightarrow B)$	$\Rightarrow wp(S, A) \rightarrow wp(S, B)$

Table 2.2. Properties of the *wp* and *wlp* predicate transformers

characterization of $wlp(S, R)$ describes the set of computations for which "finally R" is merely possible [22].

$\neg wlp(S, \neg R)$ The set of all states in which *there exists* a computation under control of S that belongs to the class "finally R".

This use of wlp is contrasted to $wlp(S, R)$ which, recall, is the set states in which the computation under control of S belongs to the class "eternal" or "finally R".

An analogous characterization can be made in terms of the computation state space that describes initial conditions using the *strongest postcondition* $sp(S, Q)$ predicate transformer [22].

$sp(S, Q)$ The set of all states in which *there exists* a computation under control of S that belongs to the class "initially Q".

That is, given that Q holds, execution of S results in $sp(S, Q)$ true. It can be proven that $sp(S, Q)$ and $wlp(S, R)$ are converses of one another [22]. Therefore, given the Hoare triple $Q \{ S \} R$ we observe that $sp(S, Q)$ and $wlp(S, R)$ have the following relationship:

$$Q \Rightarrow wlp(S, R),$$

$$sp(S, Q) \Rightarrow R.$$

The importance of this property is two-fold. First, it provides a basis for translating programming statements into formal specifications. Second, the symmetry of *sp* and *wlp* provides a method for verifying the correctness of a reverse engineering process that utilizes the properties of *sp*.

CHAPTER 3

A Translational Approach to Reverse Engineering

Reverse engineering of program code into formal specifications facilitates the utilization of the benefits of formal methods in projects where formal methods may not have previously been used. Program annotations in the form of Hoare triples [21] provide a natural way for presenting the formal specification of sequential programs. In this chapter we present an approach to reverse engineering that uses the properties of some well defined predicate transformers to specify programs.

3.1 Approach

Chapter 2 introduced the notation $Q \{ S \} R$ and $\{ Q \} S \{ R \}$ to denote partial and total correctness, respectively. We will use $\{ Q \} S \{ R \}$ to indicate a general triple of *precondition*, *statement*, *postcondition*.

When an interpretation is needed in a formal context (i.e., partial correctness, total correctness), one will be explicitly stated. The general approach described herein for reverse engineering is a top-down, sequential translation (derivation) of imperative programming language primitives using the properties defined by the strongest post-

condition predicate transformer *sp*. The approach is: Given that some precondition Q is known, a postcondition R is constructed using the properties of *sp* as applied to a statement S.

3.2 Model of Programs with Annotations

Program annotations in the form of preconditions and postconditions can be used to document, in-line, the specification of a program. Given a program P with a sequence of n statements

$$S_1; S_2; \ldots; S_{i-1}; S_i; \ldots; S_{n-1}; S_n$$

a sequence of specifications can be used to annotate the program such that the specification indicates the "state" of the program at a given point of execution. For instance, the above sequence can be annotated to appear as

$$\{Sp_0\}S_1; \{Sp_1\}S_2; \ldots; \{Sp_2\}S_{i-1}; \{Sp_3\}S_i; \ldots; \{Sp_{n-2}\}S_{n-1}; \{Sp_{n-1}\}S_n\{Sp_n\}$$

where specifications are delimited by the "{ }" notation, and Sp_j is the specification of program P after execution of statement S_j. Alternatively, we can annotate the program with comments in the following manner:

$$\{C_0\}S_1; \{C_1\}S_2; \ldots; \{C_{i-2}\}S_{i-1}; \{C_{i-1}\}S_i; \ldots; \{C_{n-2}\}S_{n-1}; \{C_{n-1}\}S_n\{C_n\}$$

where C_i is a comment annotated to the program after statement S_i and before statement S_{i+1}. We can model a program P with annotations as a 4-tuple

$$P(V, S, A_I, Value_I) \tag{3.1}$$

where V is a set of variables, S is a sequence of statements, A_I is a sequence of annotations, and $Value_I$ is an interpretation function, where I denotes the interpretation.

The model can be extended to accommodate multiple types of annotations, each with different interpretations. Consider the sequence

$$\{Sp_{u0}\}\{Sp_{v0}\}S_1; \{Sp_{u1}\}\{Sp_{v1}\}S_2; \ldots; S_{i-1}; \{Sp_{ui-1}\}\{Sp_{vi-1}\}S_i; \{Sp_{ui}\}\{Sp_{vi}\}\ldots, \tag{3.2}$$

where Sp_{uj} and Sp_{vj} each denote annotations for a statement S_j with different interpretations. The appropriate model would appear as

$$P(V, S, \{A_u, A_v\}, \{Value_u, Value_v\}) \tag{3.3}$$

where V and S are as before, A_u and A_v are annotation sequences, and $Value_u$ and $Value_v$ define the interpretations of A_u and A_v, respectively.

The remainder of this section describes each of the components of the general program model and provides an interpretation of $Value_I$ for reverse engineering.

3.2.1 Variables

Variable decorations are used to indicate a temporal ordering of logical instances of a variable. In Z, decorations $v?$ and $v!$ are used with a variable v to indicate logical input and logical output instances, respectively [23]. Other specification languages use prime (i.e. v', v'', etc.) and vector decorations ($\vec{v}$) to indicate temporal orderings [24]. In our model, a variable is represented using subscripted integer indices (i.e. $v_0, v_1, \ldots$), where an ordering is imposed by the index and each v_i represents a logical instance. There are three types of values that a logical instance can take. The first is as an unconditional expression. In this case, the statement generating the instance is an assignment statement. The second type of value is a conditional value, where the value of the logical instance depends on one or more Boolean conditions. Using disjunction, a single logical instance can be used to describe a value that is dependent

on a Boolean condition. Consider the simple statement

```
if A then x := 1 else x := 2 fi
```

We can describe the i^{th} instance of the variable **x** after execution of the statement by stating $x_i = 1 \vee x_i = 2$. The final type of instance is a range instance. Again a single logical instance can be used to describe the variable. Consider the sequence

```
x := 0;
j := 0;
a[] := A;
n := N;
do j <> n ->
   if a[j] = T  -> x,j := x + 1,j + 1
      a[j] <> T -> j := j + 1
   fi
od
```

where we are interested in the specification of the variable x after execution of the **do-od** statement. In this case, x can take a value anywhere between 0 and N. We can describe the i^{th} instance of **x** by stating $(\exists j : 0 \leq j \leq N : x_i = j)$.

Notationally, a variable v is a sequence with the following form

$$v_0, v_1, \ldots, v_k, \tag{3.4}$$

where for $0 \leq i \leq k$, v_i represents a logical instance. The size k of a sequence v is bounded from above by the number of annotations in a program P. For a given program $P(V, S, A_I, Value_I)$, $V(P)$ denotes the set of all defined variables,

$$V(P) = \{u | u \text{ is a variable with form given by (3.4)}\} \tag{3.5}$$

This treatment of variables makes the following assumptions:

1. *All variables in V(P) are local to P.*

This condition ensures that a variable in $V(P)$ is specified within the correct scope. The main motivation is to use the program model to specify subprograms of P.

2. *All variables are untyped.*

   Symbolically this assumption poses no problems. However, in practice, programming languages allow mixed type operations with appropriate transformations.

3. *All variables are simple.*

   Structured types can be represented as simple variables by decomposing the structures into the appropriate components. Pointers are also excluded from the class of variables handled in this formalization. Pointers merit a more rigorous treatment and are beyond of the scope of this work.

3.2.2 Statements

There are five main types of statements: *assignment*, *alternation*, *iteration*, *sequence*, and *procedure call*. $S(P)$ denotes the sequence of statements of a program P. The ordering of the statement sequence given by $S(P)$ corresponds directly to the order found in the source text of a program. The sequence is of the form

$$S(P) = s_1, s_2, \ldots, s_n \tag{3.6}$$

where n is the number of statements in the text of the program.

3.2.3 Annotations

$A(P)$ denotes the sequence of annotations of a program P. The sequence has form

$$A(P) = a_1, a_2, \ldots, a_{n+1} \tag{3.7}$$

where n is the number of statements in the text of the program and $n+1$ is the number of annotations. As stated earlier, there can be any number of annotation sequences, each with a different interpretation. The interpretation function for predicate logic annotations has the form

$$Value_I : A \rightarrow \Sigma^* \tag{3.8}$$

where the subscript I identifies a particular type of interpretation of the annotation, and Σ^* denotes the set of valid predicate logic expressions. The range for A with respect to $Value_I$ changes depending on the interpretation of the annotation. The notion of annotation interpretation is described in more depth in Section 3.2.5.

3.2.4 Relating Statements and Annotations

A convenience operation in the model is the association of annotations with a given statement using *prec* and *succ*. Symbolically, *prec* and *succ* represent *precedes* and *succeeds* and are defined as follows, where (3.9) and (3.10) are function signatures and (3.11) and (3.12) define the semantics for *prec* and *succ*, respectively. The *prec* function returns the annotation preceding statement S and the *succ* function returns the annotation succeeding a statement S.

$$prec : S \rightarrow A \tag{3.9}$$

$$succ : S \rightarrow A \tag{3.10}$$

where

$$prec(s_i) \equiv a_i, \quad 1 \leq i \leq n, n = |S| \tag{3.11}$$

and

$$succ(s_i) \equiv a_{i+1}, \quad 1 \leq i \leq n, n = |S| \tag{3.12}$$

3.2.5 Annotation Interpretations

Annotations can have many interpretations. For instance, suppose signature 3.8 is interpreted as a weakest precondition assertion using predicate logic, denoted $Value_{wp}$. Then for some statement s we have $prec(s) = a_{prec}$ and $succ(s) = a_{succ}$ such that $Value_{wp}(a_{succ}) = R$ and $Value_{wp}(a_{prec}) = wp(s, R)$. Additionally, recall that many different annotation sequences can exist for a given program (See Expressions (3.2) and (3.3)). So, in addition to the *wp* interpretation for predicate logic, a second interpretation to a program can be given using a second sequence of annotations. Allowing different interpretations for annotations facilitates the use of different specification techniques, each of which can provide an alternative perspective of a given program.

3.3 An Annotation Interpretation based on *sp*

The previous section described an annotation interpretation based on the semantics of the weakest precondition predicate transformer *wp*. Another interpretation for annotations provides the framework for the reverse engineering techniques described in the following chapters using the strongest postcondition predicate transformer *sp*. When given a statement and a predicate logic expression, *sp* derives a predicate logic specification of the statement. The formal presentation of the semantics of *sp* with respect to primitive programming structures is given in Chapters 4 and 5. Using the

definition of sp we define an annotation interpretation, denoted $Value_{sp}$, with the signature given above by Expression (3.8) and semantics as follows:

$$Value_{sp}(succ(s_i)) \equiv sp(s_i, Value_{sp}(pre(s_i))) \wedge \underline{Value_{sp}(pre(s_i))_p} \qquad (3.13)$$

where $s_i \in S(P), 1 \leq i \leq |S(P)|$ for some program P. Note that the second conjunct (underlined) is a *propagation* of the precondition for a given statement s_i. However, also note that we subscript the second conjunct with a p to indicate that the propagated conjunct is subject to a *propagation interpretation*, named so in order to distinguish it from *annotation* interpretations given by $Value_I$.

The interpretation of $Value_I$ as a strongest postcondition annotation provides the basis of a model for the creation of formal specifications from program code. Thus, given some initial annotation a_1, each step in the reverse engineering process involves the derivation of a formal specification of some statement using precondition assertions and a propagated expression that provides context for the derivation.

3.3.1 Translational Propagation

There are two types of propagation that can be used to interpret the second conjunct of Expression (3.13). The first type of propagation is *dependent* propagation. In this interpretation, each propagated item is dependent on the value given by the translation of a statement by sp. The second type of propagation is *independent* propagation, where, as one might expect, the propagated item is independent of translation.

The difference in the interpretations lies in the definition of the initial precondition annotation $a_1 \in A(P)$, whose value is given by $Value(a_1)$. In the dependent

interpretation, the initial annotation has the following value

$$Value_{sp}(a_1) \equiv (\forall v : v \in V(P) \wedge k = |v| : (v_0 = v^0) \wedge (v_1 = \epsilon) \ldots \wedge (v_k = \epsilon)) \quad (3.14)$$

where v^i is the value of instance v_i and ϵ denotes an undefined value. In the dependence interpretation the initial condition of a program is a specification of all the instances of all the variables of a program. When the initial precondition of a program is defined as such, then as a formal specification is derived for each statement s, propagation of $Value_{sp}(prec(s))$ must be modified to reflect the changes to each instance of a variable.

In the independent interpretation for propagated items the initial annotation a_1 has the following value

$$Value_{sp}(a1) \equiv (\forall v : v \in V(P) : v_0 = v^0). \quad (3.15)$$

That is, the initial condition for a program is a specification of the values of the initial instances of all variables. This convention allows for the propagation of statement preconditions without requiring modifications to the propagated expression Q_p.

CHAPTER 4

Primitive Constructs

In this chapter we discuss the derivation of formal specifications from the primitive programming constructs of assignment, alternation, and sequences. The Dijkstra language [25] is used to represent each primitive construct, but the techniques are applicable to the general class of imperative languages. For each primitive we first describe the semantics of the predicate transformers *wlp* and *sp* as they apply to each primitive and then describe the specification derivation using Hoare triples.

4.1 Assignment

An assignment statement has the form **x**:= **e**; where **x** is a variable, and **e** is an expression. The *wlp* of an assignment statement is expressed as $wlp(\mathbf{x}:=\mathbf{e}, R) = R^x_e$, which represents textual substitution, where every occurrence of x is replaced by the expression e using the postcondition R[1]. If x corresponds to a vector $\overline{y}$ of variables and e represents a vector $\overline{E}$ of expressions, then the *wlp* of the assignment is of the form $R^{\overline{y}}_{\overline{E}}$, where each y_i is replaced by E_i, respectively, in expression R. The *sp* of an assignment statement is expressed as

$$sp(\mathbf{x}:=\mathbf{e}, Q) = (\exists v :: Q^x_v \wedge x = e^x_v), \tag{4.1}$$

where v is the previous value of $\mathbf{x}$, and '::' indicates that the range of the quantified variable x_0 is not used in the current context.

Recall from Expression (3.13) that we assume a propagation of preconditions in the construction of formal specifications and that the initial precondition of a program is given by Expression (3.15). In terms of specifying programs using variable instances, sp is expressed as $sp(\mathbf{x}:=\mathbf{e}, Q) = (\exists x_{i-1} :: Q \wedge x_i = e^x_{x_{i-1}})$, where x_i and x_{i-1} are the current and previous values of $\mathbf{x}$, respectively. We make the conjecture that the removal of the quantification for the initial values of a variable is valid if the precondition Q has a conjunct that specifies the textual substitution. That is, Q^x_v in Expression (4.1) is a redundant operation if, initially, Q has a conjunct of the form $x = v$. As such, if given the Hoare triple for a statement s_i, with $Statement(s_i) = \mathbf{x}:=\mathbf{e}$ and $U = prec(s_{i-1})$, the following annotated sequence is created. q

$$
\begin{array}{ll}
\{(x_j = X) \wedge U\} & \texttt{/* precondition */} \\
x := e; & \\
\{(x_{j+1} = e^x_{x_j}) \wedge (x_j = X) \wedge U\} & \texttt{/* postcondition */}
\end{array}
$$

which satisfies the sp for assignment.

4.2 Alternation

An alternation statement using the Dijkstra language [25] is expressed as

$$
\begin{array}{l}
\texttt{if} \\
\qquad \mathtt{B_1} \rightarrow \mathtt{S_1}; \\
\qquad \ldots \\
\qquad [\!] \; \mathtt{B}_n \rightarrow \mathtt{S}_n; \\
\texttt{fi;}
\end{array}
$$

where $\mathtt{B}_i \rightarrow \mathtt{S}_i$ is a guarded command such that S_i is only executed if logical expression (guard) $\mathtt{B}_i$ is true. The wlp for an alternation statement is given by

$$wlp(\mathtt{IF}, R) \equiv (\forall i : \mathtt{B}_i : wlp(\mathtt{S}_i, R)),$$

where IF represents the alternation statement. The equation states that the necessary condition to satisfy R, if the alternation statement terminates, is that given $\mathtt{B}_i$ is *true*, the *wlp* for each guarded statement S_i with respect to R holds. The *sp* for alternation has form

$$sp(\mathtt{IF}, Q) \equiv (\exists i :: sp(\mathtt{S}_i, \mathtt{B}_i \wedge Q)), \tag{4.2}$$

The existential expression can be expanded to be of the form

$$sp(\mathtt{IF}, Q) \equiv (sp(\mathtt{S}_1, \mathtt{B}_1 \wedge Q) \vee \ldots \vee sp(\mathtt{S}_n, \mathtt{B}_n \wedge Q)). \tag{4.3}$$

Expression (4.3) illustrates the disjunctive nature of alternation statements where each disjunct describes the postcondition in terms of both the precondition Q and the guard-guarded command pairs, given by $\mathtt{B}_i$ and $\mathtt{S}_i$. This characterization is intuitive in that a statement $\mathtt{S}_i$ is only executed if $\mathtt{B}_i$ is true, and that only one of $\mathtt{S}_j$, $1 \leq j \leq n$, is executed. The translation of alternation statements follows accordingly from Expression (4.3). Using the Hoare triple notation, a specification is constructed as follows

$\{\ Q\ \}$
if
$\quad \mathtt{B}_1 \rightarrow \mathtt{S}_1;$
$\quad \ldots$
$\quad [\!] \ \mathtt{B}_n \rightarrow \mathtt{S}_n;$
fi;
$\{\ (sp(\mathtt{S}_1, B_1 \wedge Q) \vee \ldots \vee sp(\mathtt{S}_n, B_n \wedge Q)) \wedge Q_p\ \}$

where Q_p is a propagation of the precondition of the statement being specified.

4.3 Sequence

For a given sequence of statements $\mathtt{S}_1; \ldots;\ \mathtt{S}_n$, the notion that the postcondition for some statement $\mathtt{S}_i$ is the precondition for some other statement $\mathtt{S}_{i+1}$ is natural. The

wlp and *sp* for sequences follow accordingly. The *wlp* for sequences is defined in the following manner

$$wlp(\mathtt{S}_1;\mathtt{S}_2, R) \equiv wlp(\mathtt{S}_1, wlp(\mathtt{S}_2, R)).$$

Conversely, the *sp* is

$$sp(\mathtt{S}_1;\mathtt{S}_2, Q) \equiv sp(\mathtt{S}_2, sp(\mathtt{S}_1, Q)). \tag{4.4}$$

In the case of *wlp*, the set of states for which the sequence $\mathtt{S}_1;\mathtt{S}_2$ can execute with R true (if the sequence terminates) is equivalent to the *wlp* of $\mathtt{S}_1$ with respect to the set of states defined by $wlp(\mathtt{S}_2, R)$. For *sp*, the derived state for the sequence $\mathtt{S}_1;\mathtt{S}_2$ with respect to the precondition Q is equivalent to the derived postcondition for $\mathtt{S}_2$ with respect to a precondition given by $sp(\mathtt{S}_1, Q)$. The Hoare triple formulation and construction process is as follows

$$\begin{array}{l}
\{\ Q\ \} \\
\mathtt{S}_1; \\
\{\ sp(\mathtt{S}_1, Q) \wedge Q_p \} \\
\mathtt{S}_2; \\
\{\ sp(\mathtt{S}_2, sp(\mathtt{S}_1, Q)) \wedge Q_p\ \}.
\end{array}$$

CHAPTER 5

Iterative and Procedural Constructs

The programming constructs of assignment, alternation, and sequence can be combined to produce straight-line programs (programs without iteration or recursion). The introduction of iteration and recursion into programs provides for more powerful computation ability. However, constructing formal specifications of iterative and recursive programs can be problematic, even for the human specifier. This section discusses the formal specification of iteration and procedural abstractions without recursion. We deviate from our previous convention of providing the formalisms for *wlp* and *sp* for each construct and use an operational definition of how specifications are constructed. This approach is necessitated by the fact that the formalisms for the *wlp* and *sp* for iteration are defined in terms of recursive functions [1, 22] that are in general difficult to practically apply.

5.1 Iteration

Iteration allows for the repetitive application of a statement. Iteration, using the Dijkstra language, has the form

```
do
        B1 → S1;
        ...
      || Bn → Sn;
od;
```

In general, the iteration statement may contain any number of guarded commands of the form $B_i \rightarrow S_i$, such that the loop is executed as long as any guard B_i is true.

In the context of iteration, a *bound function* determines the upper bound on the number of iterations still to be performed on the loop. An *invariant* is a predicate that is true before and after each iteration of a loop. The problem of constructing formal specifications of iteration statements is difficult because the bound functions and the invariants must be identified. However, for a partial correctness model of execution, concerns of boundedness and termination fall outside of the interpretation, and thus can be relaxed.

Gries defines guidelines for developing loops through the identification of loop invariants [1]. The methods of *deleting a conjunct*, *replacing a constant by a variable*, *enlarging the range of a variable*, and *adding a disjunct* can provide insight into the automated construction of a specification from program code. For instance, a loop written using the method of *replacing a constant by a variable* must identify the upper (lower) bound of an incremented (decremented) variable. Furthermore, determining the statements that ensure progress towards termination is facilitated by the properties associated with this class of loops. Figure 5.1 gives the steps for constructing a specification for a loop that was developed using the *replace a constant by a variable* strategy for the loop invariant. Although these characteristics are more total correctness in nature, the insight provided by identifying these properties in loops aids in specifying a loop using partial correctness interpretations.

When no automated strategy can be applied to a loop, the domain expert* is

*A domain expert is a maintenance engineer who is familiar with the subject system.

prompted for the proper specification of the statement. The following items are then identified in order to confirm that the specification of the loop is complete:

- *invariant (P):* an expression describing the conditions prior to entry and upon exit of the iterative structure.
- *guards (B):* Boolean expressions that restrict the entry into the loop. Execution of each guarded command, $B_i \rightarrow S_i$ terminates with P *true*, so that P is an invariant of the loop.

$$\{P \wedge B_i\} S_i \{P\}, \text{ for } 1 \leq i \leq n$$

When none of the guards is *true* and the invariant is *true*, then the postcondition of the loop should be satisfied ($P \wedge \neg BB \rightarrow R$, where $BB = B_1 \vee \ldots \vee B_n$ and R is the postcondition).

---

1. The abstraction algorithm begins with the template for a quantified expression of the form
$$(Qi : range(i) : expression(i)),$$
where Q represents one of the quantifier symbols $\forall, \exists, \Sigma$.
2. The quantified variable(s) are determined by examining the identifiers occurring in guards B_j.
3. The ranges of the quantified variables are determined by finding statements occurring prior to entry into the loop that assign values to incremented (decremented) variables and their occurrences in the guards.
4. For each guarded command, the corresponding statement list includes statements that ensure progress towards termination; the postcondition for the remaining statements constitutes *expression*(i).
5. The bound function becomes the difference between the upper (lower) bound for a variable that is being incremented (decremented) and its value during loop iterations.

Figure 5.1. Steps for abstracting the effect of iteration statements

---

5.2 Procedural Abstractions without Recursion

This section describes the construction of formal specifications from code containing the use of non-recursive procedural abstractions. A procedure declaration can be represented using the following notation

$$\begin{array}{l}\textbf{proc } p \textbf{ (value } \overline{x}\textbf{; value-result } \overline{y}\textbf{; result } \overline{z} \textbf{);} \\ \quad \{P\}\langle \text{ body } \rangle\{Q\}\end{array}$$

where $\overline{x}$, $\overline{y}$, and $\overline{z}$ represent the **value**, **value-result**, and **result** parameters for the procedure, respectively. The notation ⟨ body ⟩ represents one or more statements making up the "procedure", while $\{P\}$ and $\{Q\}$ are the precondition and postcondition, respectively. The syntactic *signature* of a procedure appears as

$$\textbf{proc } p : (input\_type)^* \rightarrow (output\_type)^* \tag{5.1}$$

where the Kleene star (*) indicates zero or more repetitions of the preceding unit, *input_type* denotes the name of an input parameter to the procedure *p*, and *output_type* denotes the name of an output parameter of procedure *p*. A specification of a procedure can be constructed to be of the form

$$\begin{array}{l}\{\ \textbf{P:}\ U\ \} \\ \textbf{proc p} : E_0 \rightarrow E_1 \\ \langle body \rangle \\ \{\ \textbf{Q:}\ sp(body, U) \wedge U_p\ \}\end{array}$$

where E_0 is one or more input parameter types with attribute **value** or **value-result**, and E_1 is one or more output parameter types with attribute **value-result** or **result**. The postcondition for the body of the procedure, *sp(body, U)*, is constructed using the previously defined guidelines for assignment, alternation, and iteration as applied to the statements of the procedure body.

Gries defines a theorem for specifying the effects of a procedure call [1] using a total correctness model of execution. Given a procedure declaration of the above

form, the following condition holds [1]

$$\{PRT : P^{\overline{x},\overline{y}}_{\overline{a},\overline{b}} \wedge (\forall \overline{u}, \overline{v} :: Q^{\overline{y},\overline{z}}_{\overline{u},\overline{v}} \Rightarrow R^{\overline{b},\overline{c}}_{\overline{u},\overline{v}})\}\ p(\overline{a}, \overline{b}, \overline{c})\ \{R\} \tag{5.2}$$

for a procedure call $p(\overline{a}, \overline{b}, \overline{c})$, where $\overline{a}$, $\overline{b}$, and $\overline{c}$ represent the actual parameters of type **value**, **value-result**, and **result**, respectively. Local variables of procedure p used to compute **value-result** and **result** parameters are represented using $\overline{u}$ and $\overline{v}$, respectively. Informally, the condition states that PRT must hold before the execution of procedure p in order to satisfy R. In addition, PRT states that the precondition to procedure p must hold for the parameters passed to the procedure and that the postcondition for procedure p implies R for each **value-result** and **result** parameter. The formulation in terms of a partial correctness model of execution is identical, assuming that the procedure is straight-line, non-recursive, and terminates. Using this theorem for the procedure call, an abstraction of the effects of a procedure call can be derived using a specification of the procedure declaration. That is, the construction of a formal specification from a procedure call can be performed by inlining a procedure call and using the strongest postcondition for assignment. A procedure call $p(\overline{a}, \overline{b}, \overline{c})$ can be represented by the Pascal-like block [1] found in Figure 5.2, where $\langle body \rangle$ comprises the statements of the procedure declaration for p. By representing a procedure call in this manner, parameter binding can be achieved through multiple assignment statements and a postcondition R can be established by using the sp for assignment. Removal of a procedural abstraction allows for the extension of the notion of a straight-line program to include non-recursive straight-line procedures. Making the appropriate sp substitutions, we can annotate the code sequence from Figure 5.2 to appear as follows:

$$
\begin{array}{l}
\{\ PR\ \} \\
\overline{\mathbf{x}},\overline{\mathbf{y}} := \overline{\mathbf{a}},\overline{\mathbf{b}}; \\
\{\ P\text{: } \exists \overline{\alpha},\overline{\beta} :: PR^{\overline{\mathbf{x}},\overline{\mathbf{y}}}_{\overline{\alpha},\overline{\beta}} \wedge \overline{\mathbf{x}} = \overline{\mathbf{a}}^{\overline{\mathbf{x}},\overline{\mathbf{y}}}_{\overline{\alpha},\overline{\beta}} \wedge \overline{\mathbf{y}} = \overline{\mathbf{b}}^{\overline{\mathbf{x}},\overline{\mathbf{y}}}_{\overline{\alpha},\overline{\beta}}\ \} \\
\langle body \rangle \\
\{\ Q\ \} \\
\overline{\mathbf{y}},\overline{\mathbf{z}} := \overline{\mathbf{u}},\overline{\mathbf{v}}; \\
\{\ QR\text{: } \exists \overline{\gamma},\overline{\zeta} :: Q^{\overline{\mathbf{y}},\overline{\mathbf{z}}}_{\overline{\gamma},\overline{\zeta}} \wedge \overline{\mathbf{y}} = \overline{\mathbf{u}}^{\overline{\mathbf{y}},\overline{\mathbf{z}}}_{\overline{\gamma},\overline{\zeta}} \wedge \overline{\mathbf{z}} = \overline{\mathbf{v}}^{\overline{\mathbf{y}},\overline{\mathbf{z}}}_{\overline{\gamma},\overline{\zeta}}\ \} \\
\overline{\mathbf{b}},\overline{\mathbf{c}} := \overline{\mathbf{y}},\overline{\mathbf{z}}; \\
\{\ R\text{: } \exists \overline{\vartheta},\overline{\varphi} :: QR^{\overline{\mathbf{b}},\overline{\mathbf{c}}}_{\overline{\vartheta},\overline{\varphi}} \wedge \overline{\mathbf{b}} = \overline{\mathbf{y}}^{\overline{\mathbf{b}},\overline{\mathbf{c}}}_{\overline{\vartheta},\overline{\varphi}} \wedge \overline{\mathbf{c}} = \overline{\mathbf{z}}^{\overline{\mathbf{b}},\overline{\mathbf{c}}}_{\overline{\vartheta},\overline{\varphi}}\ \}
\end{array}
$$

where $\overline{\alpha}$, $\overline{\beta}$, $\overline{\gamma}$, $\overline{\zeta}$, $\overline{\vartheta}$, and $\overline{\varphi}$ are the initial values of $\overline{\mathbf{x}}$, $\overline{\mathbf{y}}$ (before execution of the procedure body), $\overline{\mathbf{y}}$ (after execution of the procedure body), $\overline{\mathbf{z}}$, $\overline{\mathbf{b}}$, and $\overline{\mathbf{c}}$, respectively. Using the conjecture of Section 4.1 regarding assignments as well as the facts that formal and actual **result** parameters have no initial values, and local variables are used to compute the values of the **value-result** parameters, the above sequence can be simplified using the semantics of *sp* for assignments to obtain the following annotated code sequence:

$$
\begin{array}{l}
\{\ PR\ \} \\
\overline{\mathbf{x}},\overline{\mathbf{y}} := \overline{\mathbf{a}},\overline{\mathbf{b}}; \\
\{\ P\text{: } PR \wedge \overline{\mathbf{x}} = \overline{\mathbf{a}} \wedge \overline{\mathbf{y}} = \overline{\mathbf{b}}\ \} \\
\langle body \rangle \\
\{\ Q\ \} \\
\overline{\mathbf{y}},\overline{\mathbf{z}} := \overline{\mathbf{u}},\overline{\mathbf{v}}; \\
\{\ QR\text{: } Q \wedge \overline{\mathbf{y}} = \overline{\mathbf{u}}^{\overline{\mathbf{y}}}_{\overline{\gamma}} \wedge \overline{\mathbf{z}} = \overline{\mathbf{v}}^{\overline{\mathbf{y}}}_{\overline{\gamma}}\ \} \\
\overline{\mathbf{b}},\overline{\mathbf{c}} := \overline{\mathbf{y}},\overline{\mathbf{z}}; \\
\{\ R\text{: } QR \wedge \overline{\mathbf{b}} = \overline{\mathbf{y}} \wedge \overline{\mathbf{c}} = \overline{\mathbf{z}}\ \}
\end{array}
$$

where Q is derived using $sp(\langle body \rangle, P)$.

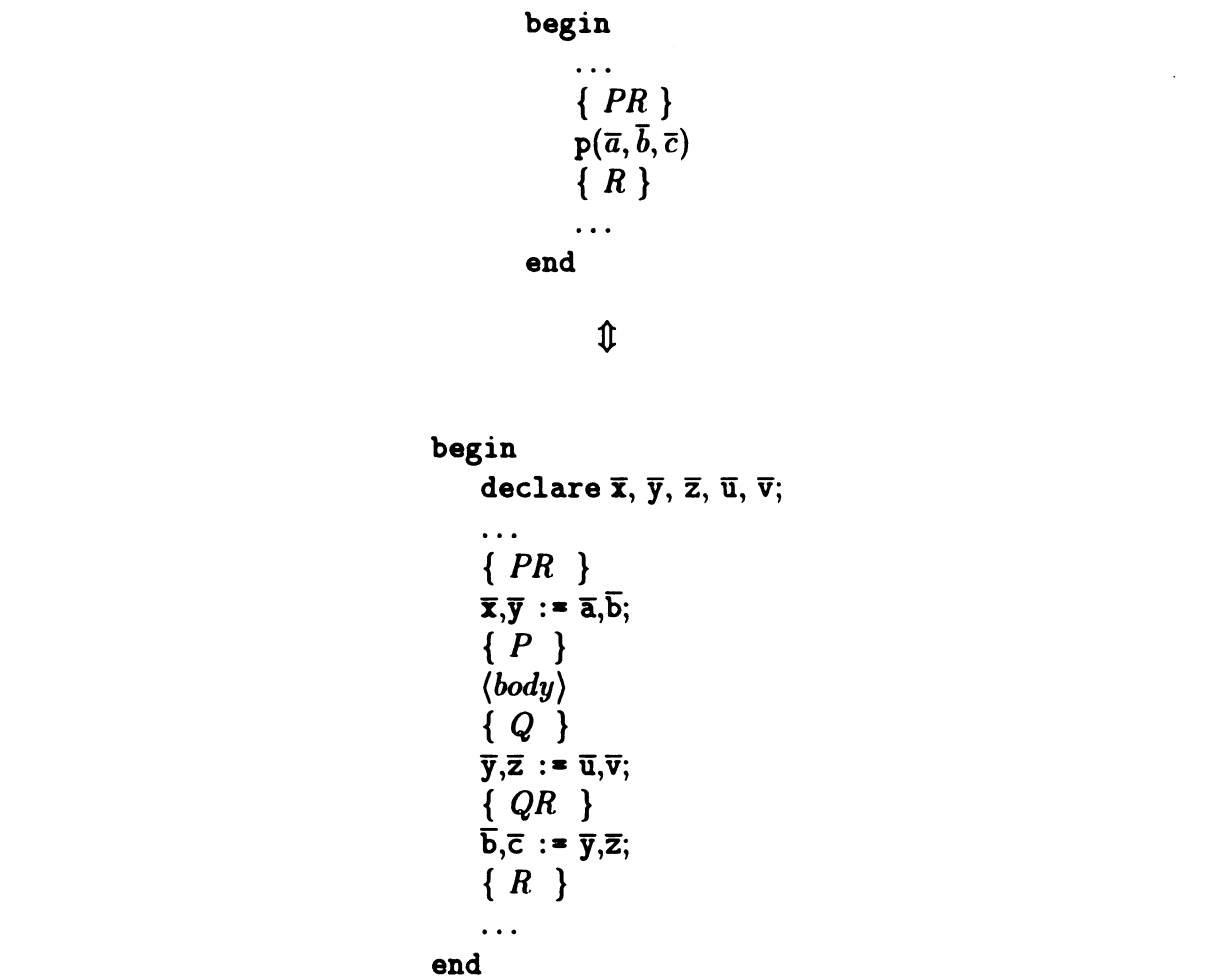


Figure 5.2. Removal of procedure call $p(\bar{a}, \bar{b}, \bar{c})$ abstraction

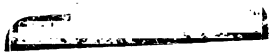
CHAPTER 6

Moving towards Object-Orientation

The main characteristics of object-oriented analysis, design, and programming are encapsulation, modularity, and inheritance. Well-defined object interfaces and a clear behavioral knowledge about an object can facilitate software reuse. In recent years, there has been an increase in the development and use of object-oriented programming languages, including C++, Smalltalk, and Modula-3. Re-engineering efforts have been focused, in part, to the conversion of software from the imperative paradigm to the object-oriented paradigm [4, 14, 26]. This chapter discusses an approach for identifying objects embedded in programs and restructuring formal specifications to reflect that identification.

6.1 Identification of Classes Using Formal Specifications

The methods described in Chapters 4 and 5 produce specifications that have properties that are amenable to the identification of candidate objects in program code. An *object* is a self-contained module that includes both the data and procedures (meth-



ods) that operate on that data. An object can be considered to be an abstract data type (ADT), that is, a user-defined data type with a specific set of allowable operations. A class is a collection of objects that have common use [27]. This section describes an aggregation heuristic for identifying objects based on the examination of procedure signatures.

6.1.1 Guidelines

Using the above definition of an object, a set of guidelines for identifying objects is as follows:

1. Construct a list of all data structures contained in a program. These data structures should not include primitive types.
2. For each data structure contained in the list of program data structures, group together the operations that refer to the data structure as input in the syntactic signature specification of the procedure (See Equation 5.1 for the format of signatures). This grouping along with the associated data structure is an *object candidate.*
3. In case of conflicts, (i.e., a procedure contains two non-primitive data structures as input in the signature) one of three actions can be taken
   (a) Determine if one data structure is composed of one or more occurrences of the other data structure. This step is performed by checking the definitions of data structures.
   (b) Determine whether the output of the procedure excludes either data structure. If it can be shown that the procedure does not modify a data structure then associate the procedure with the data structure that is modified.
   (c) In the cases where no determination can be made as to how to associate a procedure with a respective data structure, query the domain expert on the appropriate association.

The process of identifying candidate objects is facilitated by the format of the formal specifications for procedures. It is emphasized that the datatypes identified by this technique are only *candidate* objects. However, once the object definition has been constructed, the formal nature of the specification facilitates formal reasoning about the objects and can aid in the validation of candidate objects as true objects.

6.1.2 Specification Language

An object specification is constructed by collecting the procedure specifications associated to a data structure and declaring the data structure to be a candidate object. The BNF grammar for the language used to specify potential object classes is given in Figure 6.1. The definition uses the roman font to describe non-terminals; bold type defines keywords; the Kleene star (*) is used to denote one or more repetitions of the preceding unit; square brackets ([ ]) indicate optional items; parentheses ('()') indicate groupings. The non-terminal, *expression*, represents a predicate logic expression.

---

component = **type** type_name: *has* (*method*)*
has = **has** (*data_description*)*
data_description =
variable : type_name
method = **method** method_name: (type_name)* → (type_name)*
in((variable: type_name)*)
local((variable: type_name)*)
out((variable: type_name)*)
{ **pre**: *expression* }
{ **post**: *expression* }
expression = true
| false
| (*expression*)
| ¬ *expression*
| *expression* ∧ *expression*
| *expression* ∨ *expression*
| *expression* ⇒ *expression*
| *expression* ⟺ *expression*
| (∀ variable : type :: *expression*)
| (∃ variable : type :: *expression*)
| predicate_name [(*term* (, *term*)*)]
| *term* $\stackrel{\text{def}}{=}$ *expression*

term = variable
| function_name [(*term* (, *term*)*)]

Table 6.1. Grammar for object specifications

6.1.3 Example Identification of Candidate Objects

The first step of the analysis is to identify the non-primitive data structures of the program. This process is performed by examining the signatures of the procedure specifications and extracting the unique non-primitive data structure names. Analysis of the program **QeS** (shown in Figure 6.1) produces the specifications given in Figure 6.2. Analysis of the formal specifications identifies non-primitive data structures named **St**, **Qu**, and **elementtype**. In continuing the analysis of the data structure **St**, the procedures **mns**, **mts**, **tp**, and **po** are grouped with **St** through examination of the specification signatures. Procedure **pu** is initially grouped with **St** and **elementtype**, but further analysis leads to a strict association of **pu** to **St** since the input **elementtype** is not modified by **pu**. The subsequent object definition for **St** appears in Figure 6.3.

```
program QeS(output);

const
   maxlength = 50;

type
   St  = record
           t : integer;
           e : array[1..maxlength] of elementtype
         end;
   Qu = record
           e : array[1..maxlength] of elementtype;
           f, r : integer
         end;
   elementtype : Qu;

procedure mns ( var S : St );
   begin
      S.t := maxlength + 1;
   end;

function mts ( S : St ) : boolean;
   begin
      if S.t > maxlength then
         mts := true
      else
         mts := false
   end;

function tp ( var S : St ) : elementtype;
   begin
      if mts(S) then
         writeln('error');
      else
         tp := S.e[S.top]
   end;

procedure po ( var S : St );
   begin
      if mts(S) then
         writeln('error');
      else
         S.t := S.t + 1;
   end;

procedure pu ( x : elementtype; var S : St );
   begin
      if S.t = 1 then
         writeln('error');
      else
         begin
            S.t := S.t - 1;
            S.e[S.t] := x
         end;
   end;

procedure mnq ( var Q : Qu );
   begin
      Q.f := maxlength;
      Q.r := maxlength;
   end;

function mtq ( var Q : Qu ) : boolean;
   begin
      if Q.r = Q.f then
         mtq := true
      else
         mtq := false
   end;

function fr ( var Q : Qu ) : elementtype;
   begin
      if mtq(Q) then
         writeln('error')
      else
         fr := Q.e[Q.f];
   end;

procedure en ( x : elementtype; var Q : Qu );
   begin
      if Q.r = maxlength then
         Q.r := 1
      else
         Q.r := Q.r + 1;
      if Q.r = Q.f then
         writeln('error')
      else
         Q.e[Q.r] := x;
   end;

procedure de ( var Q : Qu );
   begin
      if mtq(Q) then
         writeln('error')
      else
         if Q.f = maxlength then
            Q.f := 1
         else
            Q.f := Q.f + 1;
   end;

var
   ex_s : St;
   ex_q : Qu;

begin

(* QeS Body *)

end
```

Figure 6.1. Example Pascal Code

proc mns : St → St
in(S : St)
out(S : St)
{ pre: $true$ **}**
{ post: $S.t = maxlength + 1 \wedge true$ **}**

proc tp : St → St × elementtype
in(S : St)
out(S : St, tp : elementtype)
{ pre: $domain(S)$ **}**
{ post: $(((S.t > maxlength) \wedge (mts = true)) \wedge$
$sp(writeln('\texttt{error}'), true)) \vee$
$((\neg(S.t > maxlength) \wedge (mts = false)) \wedge$
$(tp = S.e[S.t])) \wedge domain(S)$ **}**

proc po : St → St
in(S : St)
out(S : St)
{ pre: $domain(S)$ **}**
{ post: $(((S.t > maxlength) \wedge (mts = true)) \wedge$
$sp(writeln('\texttt{error}'), true)) \vee$
$((\neg(S.t > maxlength) \wedge (mts = false)) \wedge$
$(S.t_1 = S.t_0 + 1)) \wedge$
$domain(S)$ **}**

proc pu : St × elementtype → St
in(S : St, x : elementtype)
out(S : St)
{ pre: $domain(S) \wedge domain(x)$ **}**
{ post: $(S.t = 1 \wedge sp(writeln('\texttt{error}'), true)) \vee$
$(\neg(S.t = 1) \wedge (S.t_1 = S.t_0 \wedge S.e[S.t_1 = x]))$
$\wedge domain(S) \wedge domain(x)$ **}**

proc mnq : Qu → Qu
in(Q : Qu)
out(Q : Qu)
{ pre: $true$ **}**
{ post: $Q.f = maxlength \wedge$
$Q.r = maxlength \wedge true$ **}**

proc mtq : Qu → Qu × boolean
in(Q : Qu)
out(Q : Qu, mtq : boolean)
{ pre: $domain(Q)$ **}**
{ post: $(Q.r = Q.f \wedge mtq = true) \vee$
$(\neg(Q.r = Q.f) \wedge mtq = false)$
$\wedge domain(Q)$ **}**

proc fr : Qu → Qu × elementtype
in(Q : Qu)
out(Q : Qu , fr : elementtype)
{ pre: $domain(Q)$ **}**
{ post: $((Q.r = Q.f \wedge mtq = true) \wedge$
$(sp(writeln('\texttt{error}'), true))) \vee$
$((Q.r = Q.f \wedge mtq = true) \wedge$
$(fr = Q.e[Q.f])) \wedge domain(Q)$ **}**

proc en : elementtype × Qu → Qu
in(x : elementtype , Q : Qu)
out(Q : Qu)
{ pre: $domain(Q) \wedge domain(x)$ **}**
{ post: $((Q.r_0 = maxlength \wedge Q.r_1 = 1) \vee$
$(\neg(Q.r_0 = maxlength) \wedge$
$(Q.r_1 = Q.r_0 + 1))) \wedge$
$(((Q.r = Q.f) \wedge sp(writeln('\texttt{error}'), true)) \vee$
$(\neg(Q.r = Q.f) \wedge (Q.e[Q.r] = x)))$
$\wedge domain(Q)$ **}**

proc de : Qu → Qu
in(Q : Qu)
out(Q : Qu)
{ pre: $domain(Q)$ **}**
{ post: $((Q.r = Q.f \wedge mtq = true) \wedge$
$(sp(writeln('\texttt{error}'), true))) \vee$
$((\neg(Q.r = Q.f) \wedge mtq = false) \wedge$
$(((Q.f = maxlength) \wedge Q.f = 1) \vee$
$(\neg(Q.f_0 = maxlength) \wedge$
$(Q.f_1 = Q.f_0 + 1)))) \wedge domain(Q)$ **}**

Figure 6.2. A Formal Specification of Example Code in Figure 6.1

type St:
has
  t : integer;
  e : array of elementtype;

method mns : St → St
  in(S : St)
  out(S : St)
  { **pre:** $true$ }
  { **post:** $S.t = maxlength + 1 \wedge true$ }
method mts : St → boolean
  in(S : St)
  out(mts : boolean)
  { **pre:** $domain(S)$ }
  { **post:**$(((S.t > maxlength) \wedge (mts = true)) \vee$
    $(\neg(S.t > maxlength) \wedge (mts = false))) \wedge domain(S)$ }
method tp : St → St × elementtype
  in(S : St)
  out(S : St, tp : elementtype)
  { **pre:** $domain(S)$ }
  { **post:** $(((S.t > maxlength) \wedge (mts = true)) \wedge sp(writeln(\text{'error'}), true)) \vee$
    $((\neg(S.t > maxlength) \wedge (mts = false)) \wedge (tp = S.e[S.t])) \wedge domain(S)$ }
method po : St → St
  in(S : St)
  out(S : St)
  { **pre:** $domain(S)$ }
  { **post:** $(((S.t > maxlength) \wedge (mts = true)) \wedge sp(writeln(\text{'error'}), true)) \vee$
    $((\neg(S.t > maxlength) \wedge (mts = false)) \wedge (S.t_1 = S.t_0 + 1)) \wedge domain(S)$ }
method pu : St × elementtype → St
  in(S : St, x : elementtype)
  out(S : St)
  { **pre:** $domain(S) \wedge domain(x)$ }
  { **post:** $(S.t = 1 \wedge sp(writeln(\text{'error'}), true)) \vee$
    $(\neg(S.t = 1) \wedge (S.t_1 = S.t_0 \wedge S.e[S.t_1 = x)) \wedge domain(S)$ }

Figure 6.3. Specification of Object *St*

CHAPTER 7

Modeling the Representation and Abstraction of Imperative Languages

This chapter presents the rules and representations that embody the application of the translational approach to reverse engineering for the Pascal language. For ease in understanding the representations, two complementary description mechanisms are used. First, a diagramming technique, known as the *Object-Modeling Technique* (OMT) [28], is used to represent the object-oriented relationships of programming constructs.* Second, the formal specification language Larch Shared Language (LSL) [29] is used to formally specify the abstract data types that support the automated construction of formal specifications from program code.

*Note that the OMT approach includes the use of *object models*, *data flow diagrams*, and *statecharts*. In this discussion, OMT refers exclusively to the use of object models.

7.1 Properties of Block Structured Languages

The fundamental concepts of block structured languages originated with the development of ALGOL 60 [30] and have since been incorporated into the design of many programming languages, such as Ada [31] and Pascal [32]. Programs written using block structured languages are organized into nested blocks, where each block introduces a new local referencing environment. Because of their widespread and longstanding use, block structured languages have become a common object of study by maintenance engineers [33]. In order to analyze and maintain programs written using block structured languages, an understanding of the rules that govern block structured languages is necessary. The remainder of this section describes the fundamental concepts underlying block structured languages, where Pascal is used as an example of imperative (procedural) languages.

7.1.1 Static Scope Rules

Static scope rules provide the definition of the context of a variable declaration within a program. In order to determine the intended behavior of a subject system, it is important that the rules that define the scope of variables within a system are understood. When abstracting a formal specification from program code, the scope of a variable and its potential values play a major role in expressing the effects of statements. The static scope rules for block structured languages are as follows [30]:

1. The declarations at the beginning of any block define the local identifiers for a block.
2. Any identifiers referenced within a block for which no local declaration exists refer to the immediate parental block for a declaration. If no declaration is located in the parent block then the next ancestor is referenced. This identification process continues until the declaration is found (success) or no declaration is found (i.e., the top-most environment is reached and no declaration is present).

3. Declarations in nested child blocks are completely hidden from parent blocks and cannot be referenced by parents or ancestors.

4. Named subblocks in the form of subprograms are members of the parent's local referencing environment.

7.1.2 Statements

Imperative programming language constructs generally consist of four different types of basic statements regardless of whether the language is block structured or not. The programming constructs are *assignment*, *alternation*, *iteration*, and *sequence*. It is important to note that alternation, iteration, and sequence statements can contain one or more nested statements within the body of the statement. For instance, an alternation statement in Pascal can appear as

```
if (a = b) then
   begin
      x := q;
      y := r;
   end;
```

where the **if** statement contains a **begin-end** statement and, additionally, the **begin-end** statement contains two assignment statements. This property will be referred to as the *nesting property*.

7.1.3 Procedural Abstractions

A concern in the use of subprograms is parameter association. Knowing the methods for binding formal and actual parameters as well as determining the parameter transmission schemes (i.e., value, value-result) of a language are necessary prerequisite tasks for the correct abstraction of formal specifications from program code. The rules for parameter association in Pascal are as follows [34]:

1. The number of formal and actual parameters for a given function or subprogram must be identical.

2. The types of the parameters must be the same. Actual parameters of type **integer** can be coerced to type **real**.

3. The actual parameters associated with **var** formal parameters must be variables. They cannot be constants or expressions.

7.2 Modeling

In order to facilitate the automated construction of formal specifications from Pascal programs, a model of the analysis of properties contained in Section 7.1 has been developed. This section describes the formal and graphical models used to the represent Pascal language and to abstract formal specifications from Pascal programs.

7.2.1 Variables, Symbols, and Types

Section 7.1 discussed properties (including static scope rules) relevant to the reverse engineering of program code written using imperative block structured languages. Static scope rules define the methods for determining the context of an identifier within a program. As is common in compiler construction [35], a hierarchical approach has been developed for both determining scope and recording histories of identifiers within a program. This approach is implemented in the form of an abstract data type (ADT) called **SymbolTable**.

A **SymbolTable** is an ADT containing a set of **SymbolTableElements** (referred to as the *symbols* set) and a link to a parent referencing environment. Each **SymbolTableElement** object in the *symbols* set represents an identifier, a table of the values of the instances for that identifier (*history*), and an index used to reference the last instance of the identifier in the *history* table. The **SymbolTableElement** objects contained in the *symbols* set are uniquely identified within the set by the name given to each identifier. The notation convention for referring to the name, history table, and index of the last instance added to the history table for a given

identifier i will be $i.name$, $i.history$, and $i.index$, respectively. The formal specification of the **SymbolTableElement** ADT, written in the Larch Shared Language, is given in Figure 7.2.1 and is used to define a method for determining the effects of certain programming constructs. In the specification, *SymTabElement* is the name

---

SymTabElement : **trait**
includes *Integer*
introduces
% % new defines a new element
% %
new : *Str* → *Ste*
% % name gives the name of the identifier
% %
name : *Ste* → *Str*
% % addHist adds history information to an identifier's table
% %
addHist : *Ste*, *Int*, *Val* → *Ste*
% % retrieve gets a specific instance of an identifier
% %
retrieve : *Ste*, *Int* → *Val*
% % lastIndex gives the index to the last item for the
% % identifier in the table
% %
lastIndex : *Ste* → *Int*
% % ∈ determines membership in a list
% %
_ ∈ _ : *Int*, *Ste* → *Bool*

asserts ∀ *ste*, *t* : *Ste*, *v* : *Val*, *str* : *Str*, *i*, *il* : *Int*
¬(*i* ∈ *new*(*str*))
i ∈ *addHist*(*t*, *il*, *v*) == (*i* = *il*) ∨ (*i* ∈ *t*)
retrieve(*addHist*(*t*, *i*, *v*), *il*) ==
if *i* = *il* **then** *v* **else** *retrieve*(*t*, *il*)
lastIndex(*new*(*str*)) == 0
lastIndex(*addHist*(*t*, *i*, *v*)) ==
if (*i* ∈ *t*) **then** *lastIndex*(*t*) **else** *lastIndex*(*t*) + 1
name(*new*(*str*)) == *str*
name(*addHist*(*ste*, *i*, *v*)) == *name*(*ste*)

Figure 7.1. Formal Specification of **SymbolTableElement** using LSL

---

of the **SymbolTableElement** ADT, the **includes** keyword indicates that the def-

inition for the *Integer* type is used, the **introduces** keyword delimits the signature of the operators of the ADT, and the **asserts** keyword introduces the semantics of the operators, including the types of the arguments for the operators. The formal definitions for the operations of **SymbolTableElement** are used in the next section in the discussion of the abstraction process.

Figure 7.2 contains a graphical depiction of the **SymbolTable** ADT using the OMT notation that shows **SymbolTable** as an *aggregate* of *zero or more* **SymbolTableElements** and *zero or one* **SymbolTables**, where aggregation is symbolically represented by a diamond, the "zero or more" relation is represented by the filled circle, and the "zero or one" relation is represented by the hollow circle.

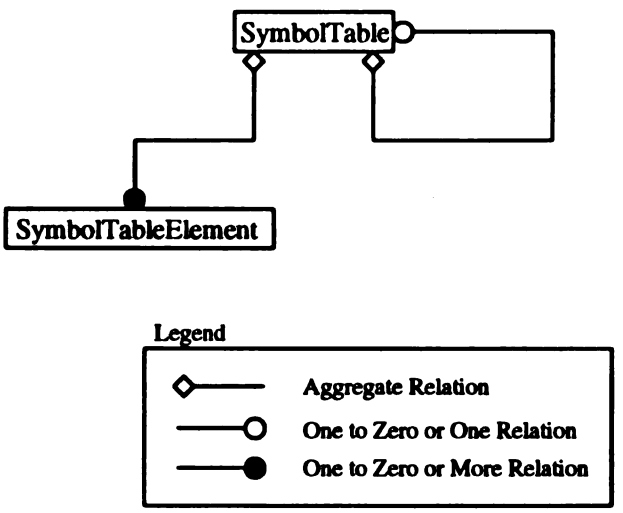


Figure 7.2. OMT Object Model of **SymbolTable**

7.2.2 Statements, Rules, and Formal Specifications

Section 7.1.2 defined the *nesting property* of programming statements. For abstraction purposes, it becomes useful to recognize that nested statements are contained, to some extent, within a single statement. For instance, consider the code sequence given in Figure 7.3. This sequence has eleven separate statements including the **begin-end** sequences. At the highest level of abstraction this sequence has one statement, the outer **begin-end** statement. The next level of granularity contains three statements, the assignment statement at line 2, the **if** statement beginning at line 3, and the assignment statement at line 14. The **if** statement beginning at line 4 contains two statements in the form of an assignment statement and a **while** statement.

```
 1 begin
 2    x := y;
 3    if D then
 4       if L then
 5          x := a;
 6       else
 7          while i <> n
 8             begin
 9                x := x + t[i];
10                i := i + 1
11             end
12    else
13       x := z;
14    q := z;
15 end
```

Figure 7.3. Example Sequence of Pascal Code

An object model is a representation in OMT that defines the hierarchy and static relationships of object classes. An instance object model depicts a snapshot of the dynamic relationships between objects in a system. Figure 7.4 contains the object model of the **Statement** ADT, where the triangle symbol denotes inheritance. As such, the diagram explicitly states that **Sequence**, **Assignment**, **Alternation**, **Iteration**,

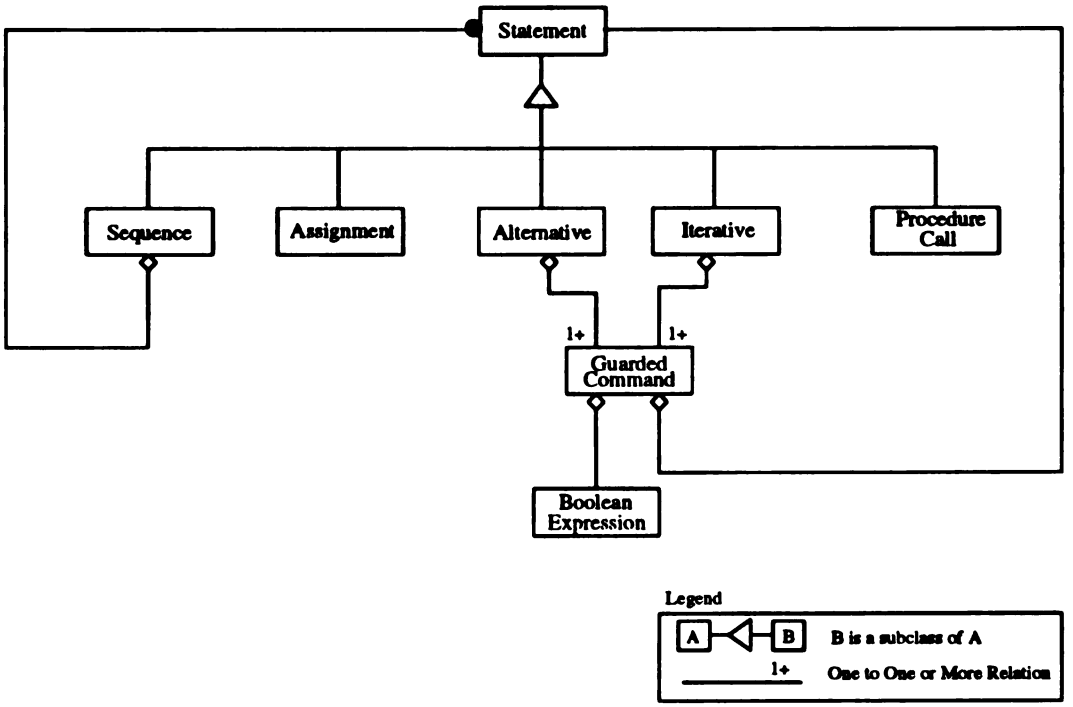


Figure 7.4. Object Model of Statements

and **ProcedureCall** are **Statements**. A **GuardedCommand**, also depicted in the diagram, is a construct that is only executed when a given boolean expression (called the *guard*) is true. Furthermore, the diagram depicts the *nesting property* through the use of the aggregation symbol. Notice that **Sequences** can contain zero or more **Statements**, and **Alternation** and **Iteration** statements can contain one or more **GuardedCommands**. Figure 7.5 shows the object instance model for the code sequence of Figure 7.3. Each oval represents a particular instance of an object class and the parenthesized words identify the type of the instance. The remaining label gives information about the instance.

An implementation of the rules given by Expressions (4.1), (4.2) and (4.4) rely heavily on the assumption that nested programming statements can be abstracted

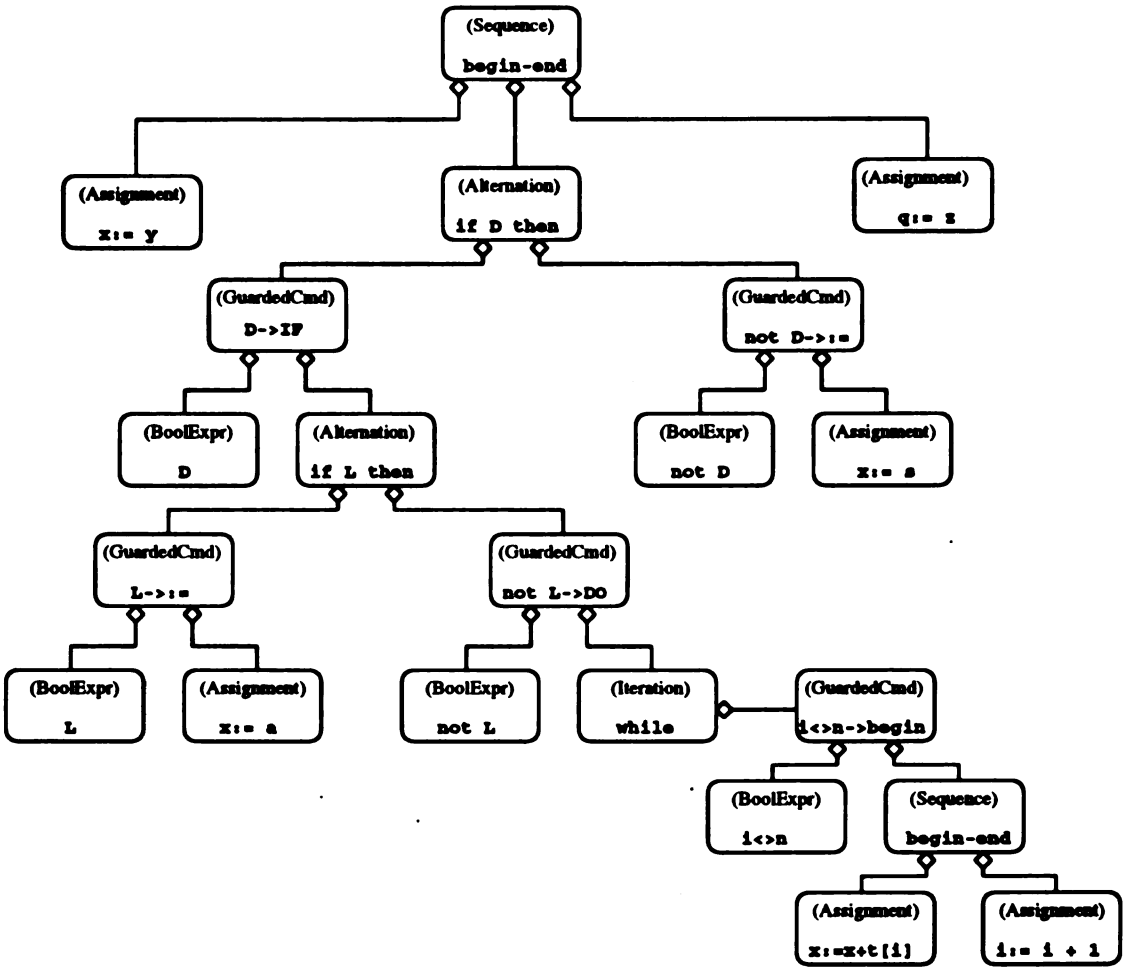


Figure 7.5. Object Instance Model for code sequence of Figure 7.3

into single statements with autonomous contexts that depend only on sequence and procedure calls. In order to support the abstraction of specifications from programming constructs, the notion of a referencing environment local to a programming statement is introduced. It is important to note that in order to abstract away the details underlying an implementation, it becomes necessary to treat programming statements as "mini-programs" with the assumption that programming statements are single entry, single exit. Currently, this assumption effectively excludes **goto** state-

ments but facilitates the hierarchical management of the abstraction process using **SymbolTable** objects, where one **SymbolTable** is used per nested sequence.

One of the main purposes of the **SymbolTable** ADT is to support the construction of specifications. Recall that the *sp* of the assignment statement is $sp(\mathbf{x}:=\mathbf{expr}, Q) = (\exists x_0 :: Q^x_{x_0} \wedge x = e^x_{x_0})$. In order for the conjunctive expression $Q^x_{x_0} \wedge x = e^x_{x_0}$ to be satisfied, expressions must be partially evaluated. The history capabilities of a **SymbolTable** directly supports the evaluation of expressions by providing information about the values of various instances of a given identifier and by providing a means for storing the effects of an assignment statement. Expressions are evaluated using textual substitution of the value of the last instance of each identifier contained in the expression. For example, an expression typically found in a program might be as follows

$$q + r - s + t. \tag{7.1}$$

An expression can be processed such that the identifiers in the expression are replaced with the representation of the last instance of each identifier. Expression (7.1), for example, can be translated into an internal representation such that each identifier in (7.1) is replaced with the corresponding internal representation of the last instance for the identifier. If it is assumed that the last instance for q, r, s, and t are q_1, r_3, s_0, and t_1, respectively, then Expression (7.1) would be translated into the following

$$q_1 + r_3 - s_0 + t_1, \tag{7.2}$$

where the subscripts represent the i^{th} instance of an identifier. For example, in Expression (7.2), r_3 represents the third instance of identifier r. Once the initial substitution of the last instance for each identifier is completed, the representation of an instance of an identifier is replaced with the actual value of the instance. For illustration purposes, assume that the values of instances q_1, r_3, s_0, and t_1 are 5, 9, s_0, and x_1, respectively. Then Expression (7.2) would be translated into the following

$$5 + 9 - s_0 + x_1. \tag{7.3}$$

This process repeats until all terms of the expression are either initial instances (e.g. s_0), conditional instances, (that is, the instance depends on the evaluation of a number of logical conditions, a concept explained in the following section), or constant values.

Consider the code sequence in Figure 7.6(a). Using a bottom-up approach to spec-

```
0. (* x0 = X & U *)
1.  if B then
2.     if L then
3.        begin
4.           x := p;
5.           x := q
6.        end
7.     else
8.        begin
9.           x := r;
10.          x := x + q;
11.          x := x - p;
12.       end
13.  else
14.     x := s
15. (* ??? *)
```

(a)

```
0.  (* x0 = X & U *)
1.  if (a = b) then
2.     if (c = d) then
3.        begin
4.           x := p;
4.1          (* x1 = p & U *)
5.           x := q
5.1          (* x2 = q & U *)
6.        end
7.     else
8.        begin
9.           x := r;
9.1          (* x1 = r & U *)
10.          x := x + q;
10.1         (* (x2 = x1 + q) & U *)
11.          x := x - p;
11.1         (* (x3 = x2 - p) & U *)
12.       end
13.  else
14.     x := s
14.1    (* x1 = s & U *)
15.  (* ??? *)
```

(b)

Figure 7.6. Code sequences partially annotated with specifications

ify the sequence requires that the deepest level of nesting be specified first, followed by the next to deepest level, and so on. This process yields the annotated code shown in Figure 7.6(b), where specifications are delimited with Pascal comment notation, '(* *)'. In our notation we use '&' and '|' to denote logical 'and' and 'or', respectively. Combining the specifications of lines 4.1, 5.1, 9.1, 10.1, 11.1, and 14.1 can often lead to an incoherent specification for line 15 due to the multiple instance subscripts for

identifier **x**. As mentioned earlier, each level of nesting in our approach is managed by using separate referencing environments through the use of **SymbolTable** objects. Using this approach we define the notion of *dirty sets* and *last instances*, which will aid in the definition of methods for correctly combining the specifications of nested statements. These methods can then be used to determine the specification such as that required for line 15.

Definition 1 (Last Instance)

Assuming that a **begin-end** *sequence is single entry, single exit, then a* **last instance** *is the most recent value of any identifier accessible during the context of the sequence.*

In some cases, the last instance may be the same as the initial instance, while in other cases the last instance is different. Given that a **SymbolTable** object, called *symtab*, is used to manage an arbitrary level of nesting, we define the *last_instance* of an identifier *id* to be

$$last\_instance(id, symtab) \equiv \begin{cases} retrieve(id,lastIndex(id)) & id \in symtab.symbols \\ undefined & \text{otherwise} \end{cases}$$

where *retrieve(id,lastIndex(id))* represents the value obtained by referencing the last item in the history table for identifier *id*, and the identifier *id* must be accessible in the current environment.

Definition 2 (Dirty Set)

A **dirty set** *is a construct that can be used to determine whether an assignment statement has been performed on an identifier in a nested sequence of statements.*

In addition to being useful for combining specifications of nested statements, a *dirty set* also aids in simplifying specifications. Formally, a *dirty set* is defined in the following manner

$$dirty\_set(orig,target) \equiv \left\{ y \in orig \mid \begin{array}{l} last\_instance(y, orig) \neq \\ last\_instance(y, target) \end{array} \right\},$$

where *orig* and *target* are **SymbolTables** of the containing and contained nested statements, respectively.

The method for abstracting specifications from Pascal alternation statements uses three **SymbolTables**. The main **SymbolTable** is used to manage the entire alternation construct while two auxiliary **SymbolTables** are used to manage the identifiers of the guarded command constructs, assuming that the alternation statement is composed of two guarded commands, that is, one guarded command corresponds to the **if** case, and the other corresponds to the **else** case. (For **case** statements, an auxiliary **SymbolTable** object would be used to manage each separate case.) By using *dirty sets*, identifiers that are modified within the scope of the guarded commands contained within an alternation statement can be determined. That is, a set of all identifiers that are conditional are specified formally as follows

$$I = \{dirty\_set(main, aux_1) \cup dirty\_set(main, aux_2)\},$$

where *main* represents the main **SymbolTable**, and aux_1 and aux_2 refer to the **SymbolTables** used to manage the guarded commands of the alternation statement. Upon determining the identifiers that fit this criterion, the main **SymbolTable** is updated in order to satisfy the following condition:

$$\begin{array}{lll} (\forall i : & i \in I & : \ (\exists j : j \in aux_1 : i.name = j.name \wedge \\ & & \quad last\_instance(j, aux_1) = last\_instance(i, orig)) \vee \\ & & \ (\exists j : j \in aux_2 : i.name = j.name \wedge \\ & & \quad last\_instance(j, aux_2) = last\_instance(i, orig))) \end{array}$$

which states that all identifiers in the dirty set are in one of the auxiliary **SymbolTables**.

Finally, with knowledge about the values of last instances of identifiers that would result from the execution of nested subblocks, a formal specification can be completed.

Consider again the sequence of code given in Figure 7.6. The final version of the code with annotated specifications of the alternation statements using the notation **id{nest}instance** for identifiers, where **id** is the identifier in question, **nest** is the level of nesting, and **instance** is the instance number within the current context, is given in Figure 7.7.

```
 1.   if (a = b) then
 2.      if (c = d) then
 3.         begin
 4.           x := p;
 4.1          (* (x{3}1 = p0) & U *)
 5.           x := q;
 5.1          (* (x{3}2 = q0) & U *)
 6.         end
 6.1        (* ((x{3}1 = p0) & (x{3}2 = q0)) & U *)
 7.      else
 8.         begin
 9.            x := r;
 9.1           (* (x{3}1 = r0) & U *)
10.           x := (x + q);
10.1          (* (x{3}2 = r0 + q0) & U *)
11.           x := (x - p);
11.1          (* (x{3}3 = ((r0 + q0) - p0)) & U *)
12.        end
12.1       (* ((x{3}1 = r0) & (x{3}2 = r0 + q0) & (x{3}3 = r0 + q0 - p0)) & U *)
12.2    (* (((c0 = d0) & ((x{3}1 = p0) & (x{1}1 = q0))) |
12.3       ( not(c0 = d0) & ((x{3}1 = r0) & (x{3}2 = r0 + q0)
12.4                      &  (x{1}1 = r0 + q0 - p0)))) & U *)
13.   else
14.      x := s;
14.1     (* (x{1}1 = s0) & U *)
15.   (* (((a0 = b0) & (((c0 = d0) & ((x{3}1 = p0) & (x{0}1 = q0))) |
15.1                    (not(c0 = d0) & ((x{3}1 = r0) & (x{3}2 = r0 + q0) &
15.2                                     (x{0}1 = (r0 + q0 - p0)))))) |
15.3      ((not(a0 = b0)) & (x{0}1 = s0))) & U *)
```

Figure 7.7. Example with specifications annotated by the AUTOSPEC [3] system

7.3 Example

The following example demonstrates the use of four major programming constructs described in this paper (assignment, alternation, sequence, and procedure call) along

with the application of the translation rules for abstracting formal specifications from code. The program, shown in Figures 7.8(a) and 7.8(b), has four procedures, including three different implementations of "swap". AUTOSPEC [3, 15, 13] is a tool that we have developed to support the derivational approach to the reverse engineering of formal specifications from program code. Figures 7.9 and 7.10 depict the output of AUTOSPEC when applied to the program code given in Figures 7.8(a) and 7.8(b), respectively, where the notation **id{scope}instance** is used to indicate a variable **id** with scope defined by the referencing environment for **scope**. The **instance** identifier is used to provide an ordering of the assignments to a variable. The **scope** identifier has two purposes. When **scope** is an integer, it indicates the level of nesting within the current program or procedure. When **scope** is an identifier, it provides information about variables specified in a different context.

Of particular interest are the specifications for the swap procedures given in Figure 7.9 named **swapa** and **swapb**. The implementation of **swapa** uses an arithmetic based algorithm for swapping the values of two variables, while **swapb** uses a temporary variable algorithm. Although each implementation of the swap operation is different, the code in each procedure effectively produces the same results, a property appropriately captured by the respective specifications for **swapa** and **swapb**. In addition, Figure 7.10 shows the formal specification of the **funnyswap** procedure. The parameter passing scheme used in this procedure is pass by value, a property reflected by the specification of the effects of the call to **funnyswap** in Figure 7.10. In the specification, no variables local to the scope of the call to **funnyswap** are affected, and thus the specification shows no change in variable values. The procedure **FindMaxMin** provides another example of the specification of alternation statements, with the specification of the procedure shown in Figure 7.9, and the effect of the call to the procedure given in Figure 7.10. This example illustrates the use of textual substitution in the specification of the effects of a call to a procedure that computes

the maximum and minimum of two variables.

```
program MaxMin ( input, output );

   var
      a, b, c, Largest, Smallest : real;

procedure FindMaxMin( NumOne, NumTwo:real;
                      var Max, Min:real );
   begin
      if NumOne > NumTwo then
         begin
            Max := NumOne;
            Min := NumTwo;
         end
      else
         begin
            Max := NumTwo;
            Min := NumOne;
         end
   end;

procedure swapa( var X:integer; var Y:integer );

   begin
      Y := Y + X;
      X := Y - X;
      Y := Y - X;
   end;

procedure swapb( var X:integer; var Y:integer );

   var
      temp : integer;
   begin
      temp := X;
      X := Y;
      Y := temp
   end;
```

(a)

```
procedure funnyswap( X:integer; Y:integer );

   var
      temp : integer;
   begin
      temp := X;
      X := Y;
      Y := temp
   end;

begin
   a := 5;
   b := 10;
   swapa(a,b);
   swapb(a,b);
   funnyswap(a,b);
   FindMaxMin(a,b,Largest,Smallest);
   c := Largest;
end.
```

(b)

Figure 7.8. Example Pascal program

```
program MaxMin( input, output );

var
   a, b, c, Largest, Smallest : real;

procedure FindMaxMin( NumOne, NumTwo:real; var Max, Min:real );

begin
   if (NumOne > NumTwo) then
      begin
         Max := NumOne;
         (* Max{2}1 = NumOne0 & U *)
         Min := NumTwo;
         (* Min{2}1 = NumTwo0 & U *)
      end
      (* (Max{2}1 = NumOne0 & Min{2}1 = NumTwo0) & U *)
   else
      begin
         Max := NumTwo;
         (* Max{2}1 = NumTwo0 & U *)
         Min := NumOne;
         (* Min{2}1 = NumOne0 & U *)
      end
      (* (Max{2}1 = NumTwo0 & Min{2}1 = NumOne0) & U *)
   (* (((NumOne0 > NumTwo0) &
           (Max{0}1 = NumOne0 & Min{0}1 = NumTwo0)) |
         (not(NumOne0 > NumTwo0) &
           (Max{0}1 = NumTwo0 & Min{0}1 = NumOne0))) & U *)
end
(* (((NumOne0 > NumTwo0) &
        (Max{0}1 = NumOne0 & Min{0}1 = NumTwo0)) |
   (not(NumOne0 > NumTwo0) &
        (Max{0}1 = NumTwo0 & Min{0}1 = NumOne0))) & U *)

procedure swapa( var X:integer; var Y:integer );

begin
   Y := (Y + X);
   (* (Y{0}1 = (Y0 + X0)) & U *)
   X := (Y - X);
   (* (X{0}1 = ((Y0 + X0) - X0)) & U *)
   Y := (Y - X);
   (* (Y{0}2 = ((Y0 + X0) - ((Y0 + X0) - X0))) & U *)
end
(* (Y{0}2 = X0 & X{0}1 = Y0 & Y{0}1 = Y0 + X0) & U *)

procedure swapb( var X:integer; var Y:integer );

var
   temp : integer;

begin
   temp := X;
   (* (temp{0}1 = X0) & U *)
   X := Y;
   (* (X{0}1 = Y0) & U *)
   Y := temp;
   (* (Y{0}1 = X0) & U *)
end
(* (Y{0}1 = X0 & X{0}1 = Y0 & temp{0}1 = X0) & U *)
```

Figure 7.9. Output created by applying AUTOSPEC to example

```
procedure funnyswap( X:integer; Y:integer );

var
   temp : integer;

begin
   temp := X;
   (* (temp{0}1 = X0) & U *)
   X := Y;
   (* (X{0}1 = Y0) & U *)
   Y := temp;
   (* (Y{0}1 = X0) & U *)
end
(* (Y{0}1 = X0 & X{0}1 = Y0 & temp{0}1 = X0) & U *)

begin
   a := 5;
   (* a{0}1 = 5 & U *)
   b := 10;
   (* b{0}1 = 10 & U *)
   swapa(a,b)
   (* (b{0}2 = 5 &
         (a{0}2 = 10 &
         (Y{swapa}2 = 5 & (X{swapa}1 = 10 & Y{swapa}1 = 15)))) & U *)
   swapb(a,b)
   (* (b{0}3 = 10 &
         (a{0}3 = 5 &
         (Y{swapb}1 = 10 & (X{swapb}1 = 5 & temp{swapb}1 = 10)))) & U *)
   funnyswap(a,b)
   (* (Y{0}1 = 5 & X{funnyswap}1 = 10 & temp{funnyswap}1 = 5) & U *)
   FindMaxMin(a,b,Largest,Smallest)
   (* (Smallest{0}1 = Min{FindMaxMin}1 &
         Largest{0}1 = Max{FindMaxMin}1 &
         (((5 > 10) &
            (Max{FindMaxMin}1 = 5 & Min{FindMaxMin}1 = 10)) |
         (not(5 > 10) &
            (Max{FindMaxMin}1 = 10 & Min{FindMaxMin}1 = 5)))) & U *)
   c := Largest;
   (* c{0}1 = Max{FindMaxMin}1 & U *)
end
(* ((c{0}1 = Max{FindMaxMin}1) &
   ( Smallest{0}1 = Min{FindMaxMin}1 & Largest{0}1 = Max{FindMaxMin}1 &
   (((5 > 10) &
      (Max{FindMaxMin}1 = 5 & Min{FindMaxMin}1 = 10)) |
   (not(5 > 10) &
      (Max{FindMaxMin}1 = 10 & Min{FindMaxMin}1 = 5)))) &
   ( Y{funnyswap}1 = 5 & X{funnyswap}1 = 10 & temp{funnyswap}1 = 5 ) &
   ( b{0}3 = 10 &
     a{0}3 = 5 &
       (Y{swapb}1 = 10 & X{swapb}1 = 5 & temp{swapb}1 = 10)) &
   ( b{0}2 = 5 &
     a{0}2 = 10 &
       (Y{swapa}2 = 5 & X{swapa}1 = 10 & Y{swapa}1 = 15)) &
   (b{0}1 = 10 & a{0}1 = 5))))) & U *)
```

Figure 7.10. Output created by applying AUTOSPEC to example (cont.)

CHAPTER 8

Related Work

Many approaches have been suggested for reverse engineering program code. This chapter discusses reverse engineering efforts that use formal and semi-formal approaches for the reverse engineering of program code into procedural and object-oriented formal specifications.

8.1 Formal Approaches

This section describes reverse engineering methods that take advantage of logical and mathematical properties of programs for the abstraction of program function as well as in the representation of program specifications.

8.1.1 Function Abstraction

It has been shown that program flowcharts can be decomposed into basic constructs that represent sequence, alternation, and iteration [36]. Using these concepts, a strategy outlining an approach for abstracting function behaviour from programs has been suggested [37]. The strategy takes advantage of the mathematical properties of structured programs in order to to abstract program function. The end result is the determination of a precise representation of a program's function. At the

heart of the approach is the concept of *proper programs*, *prime programs*, and *small primes*. A proper program is a flowchart program with a single entry and single exit. An irreducible proper program is known as a prime program, and a small prime consists of the most atomic of program statements including sequence, alternation, and iteration.

Using the approach defined by Linger et al. a specification is constructed by determining the function of each small prime contained in a program.

Overall, the approach to abstracting program function proceeds as follows:

1. Restructure the object program into a structured format.
2. Analyze the program data in order to localize the scope of variables.
3. Compute the function of each prime.

The restructuring step is essential since the rest of the approach relies on structured programming constructs as the basis for function abstraction. Special emphasis is put on the second step in order to reduce the difficulty of abstracting the specifications from the code. The final step involves a number of algorithms centered around identifying the function of each program prime.

Sequence statements, at the most basic level, consist of assignment statements of the form:

$$x := e$$

where x is a variable and e is some expression. A trace-table technique is used for determining the function of sequence statements. The purpose of the trace-table is to record the effect of an assignment on the state space of all variables in the same

scope of the statement. For example, consider the following sequence of code:

$$\begin{aligned} X &:= X + Y; \\ Y &:= X - Y; \\ X &:= X - Y; \end{aligned}$$

A trace-table containing a column for each variable within the current scope of execution can be constructed using a subscripted notation (i.e. X_i) to denote the value of a variable X in each separate state i. The trace table for the sequence shown above would appear as [37]:

Statement	Value of X	Value of Y
Initially	X_0	Y_0
$X := X + Y$	$X_1 = X_0 + Y_0$	$Y_1 = Y_0$
$Y := X - Y$	$X_2 = X_1 = X_0 + Y_0$	$Y_2 = X_1 - Y_1 = X_0$
$X := X - Y$	$X_3 = X_2 - Y_2 = Y_0$	$Y_3 = Y_2 = X_0$

Analysis of the trace-table allows for the determination that the sequence of code performs a swap of the values of X and Y.

Alternation abstraction is based on analyzing the guarded command conditions of the statements. An alternative statement is one that takes the form:

IF *E* THEN *C1* ELSE *C2* ENDIF

where B is a guard and C_n is a guarded command. The specification created from abstracting the function from alternative statements involves the use of the conditions in the statement and the specification of the subsequent guarded statements. For example, consider the following statement

IF b THEN g ELSE h END-IF,

where b is a guard, and g and h are statements. A specification is constructed with the following form

$$([b] = true \rightarrow [g] | [b] = false \rightarrow [h]),$$

which states that whenever $[b]$ is $true$, perform statement g, and whenever $[b]$ is false, perform statement h.

Linger et al. describe a number of strategies for understanding the function of iterative constructs. One approach is through the use of *program slicing.* Program slicing is a technique whereby the effect of a loop on an isolated variable is specified for each of the variables in the loop. Upon completion, the specifications are combined to in order to determine a specification for the entire loop. Another proposed approach for understanding loops uses *pattern matching* with the help of *superpatterns.* Superpatterns are templates that outline common formats of programming constructs, particularly loops. This proceeds by performing program slicing on a loop and comparing the results with a number of standard superpatterns.

8.1.2 A Knowledge Based Transformational System

Transformational systems have been widely used for forward engineering. The application of transformational systems to reverse engineering has been proposed by Ward et al. [38, 39, 40]. The approach relies on proving that two versions of a program are equivalent. A kernel language that includes both imperative programming constructs and general specifications as part of a single language is defined. By using a kernel or *Wide Spectrum Language* (WSL) that is mathematically based, the problem of abstracting a specification of a program can be reduced to first transforming a program into another form (a specification) and then proving the equivalence of the original and the new form.

The Maintainer's Assistant is a system that incorporates the notions of Wide

Spectrum Languages and a Knowledge Base for performing transformations of program code into specifications. The rest of this section outlines the major aspects of the Maintainer's Assistant.

Wide Spectrum Languages

Software development classically proceeds through multiple stages ranging from requirements specification to program development. Figure 8.1 illustrates this concept [39]. Typically each stage uses a different language to express the problem. For instance, natural language may be used for requirements stage S_0 while a formal specification language may be used for specification stage S_n as can be seen in Figure 8.2 [39]. A WSL provides a unified language for use in all stages starting from S_0 and ending in P. The WSL encorporates both executable imperative constructs as

$$S_0 \leftrightarrow S_1 \leftrightarrow S_2 \leftrightarrow \ldots \leftrightarrow S_n \leftrightarrow P$$

Figure 8.1. Stages of Development

$$SL_0 \leftrightarrow SL_1 \leftrightarrow SL_2 \leftrightarrow \ldots \leftrightarrow SL_n \leftrightarrow PL$$

Figure 8.2. Languages for Development

well as non-executable specifications. By providing a strong mathematical basis for a

WSL, proving a transformation can be performed by proving the equivalence of two formulae [39].

Knowledge Base

To assist in performing transformations of programs into specifications the Maintainer's Assistant encorporates the use of a knowledge base [40]. This knowledge base is centered around identifying programs plans. Program plans can be divided into two classes, *General Program Plans* and *Program Class Knowledge.* Use of a general plan knowledge base aids in identifying commonly occuring activities in programs. Program Class knowledge is used for specific types of activities. For instance, program class knowledge could exist for maintaining a C compiler. This knowledge would include information on design decisions typically found in developing a compiler.

Transformations

Transformation has been used in many instances for forward engineering tasks [41]. Transformation is the action of substituting part of a program with a new part that is meaning preserving (i.e., the new part computes the same thing as the old part). There are typically three steps to transformations. The first is a determination of where to apply a transformation, next is a determination of what transformations to apply, and finally creation of a new program part to replace the matched pattern. Transformations can take two forms, vertical and lateral. Vertical transformations with respect to program development are used to move from an abstract representation to a concrete one while lateral transformations are used to specify equivalence between two expressions at the same level of abstraction.

There are several types of transformations supported by the Maintainer's Assistant. Two approaches are used to perform transformations; the catalog approach,

and the generative set approach. The catalog approach uses the knowledge base described in the previous section. The generative set approach uses a set of elementary transformations to build more complex transformations.

The Maintainer's Assistant has the ability to perform the following types of transformations:

Syntactic Transformations: This type of transformation is synonymous with restructuring.

Action Systems: These transformations are used to simplify languages that allow goto statements.

Non-Terminal Recursion: Used to remove non-terminal recursion by using a stack of "postponed obligations".

Local Semantic Transformations: These are transformations for changing the flow of computation.

8.2 Object-Orientation

Object-Oriented Analysis, Design, and Programming have become increasingly popular in recent years. The encapsulation, inheritance, and reuse properties of Object-Oriented Programming has made the approach attractive to many software developers. A desire to migrate existing systems from an imperative paradigm to an object-oriented paradigm has prompted the development of a number of reverse engineering methods for facilitating the migration. This section describes two approaches for abstracting objects from imperative programs.

8.2.1 Object Identification

One approach to identifying objects in procedural code has been proposed by Liu and Wilde [14]. Identification is centered around the characterization of candidate objects based on common routines, data types, and data items. These properties are

arranged as a tuple of the following format:

$$CandidateObject = (F, T, D)$$

where F is a set of routines, T is a set of types, and D is a set of data objects. Candidate objects may not be completely disjoint which means that there may be an overlap between data objects and routines. Additionally, there is no clear distinction between an object and an object class. This definition of a candidate object differs from the classical notion of objects but is necessary.

Candidate objects can overlap since rejection would limit the domain and exclude the manifestations of objects whose only failing is that they were produced using poor programming practices. The lack of a clear distinction between classes and class objects is due to the fact that it may be easier in some cases to identify object classes and then identify instances of the classes. In other cases it may be easier to reverse this process by identifying the instances of the classes, and then the classes.

There are two methods of object identification. The first uses global and static data and relates operations that operate on that data. The second uses data types and relates routines that use the data types as parameters or return values.

Global Based Object Identification

The method for identifying candidate objects based on global and static data uses the following steps [14]:

1. For each global variable x, let $P(x)$ be the set of routines that use x.
2. Using each $P(x)$ as a vertex, construct a graph $G = (V,E)$ such that:

   $V = \{P(x) \mid x \text{ is shared by at least two routines}\}$
   $E = \{\overline{P(x_1)P(x_2)} \mid P(x_1) \cap P(x_2) \neq \emptyset\}$
   where $\overline{P(x_1)P(x_2)}$ is an edge joining two vertices $P(x_1)$ and $P(x_2)$.

3. Construct a candidate object tuple *(F,T,D)* from each strongly connected component $C = (V_c, E_c)$ in G such that:

$$\begin{aligned} F &= \bigcup_{P(x) \in V_c} P(x) \\ T &= \oslash \\ D &= \bigcup_{P(x) \in V_c} \{x\} \end{aligned}$$

The major drawback to using the global based object identification method is that the candidate objects that are found can often be very large since all functions that reference a global variable will be included in the F component of the candidate object tuple. Other procedures will be needed to constrain the search so that acceptable objects are identified.

Types Based Object Identification

The second type of object finding method is a *types-based* algorithm. The steps to this method are as follows [14]:

1. (Ordering) A topological order of all types must be defined such that:
   (a) If type x is used to define type y, then x is a *sub-type* of y and y is a *supertype* of x, denoted by $x \ll y$
   (b) If $x \ll y$ and $y \ll z$ then $x \ll z$ (*Transitivity*).
2. (Initial Classification) A relationship matrix *R(F,T)* is constructed where rows are routines and columns are types of formal parameters and return values. Values of *R(F,T)* are either 0 or 1 with 1 indicating that for an entry *R(f,t)*, *t* is a subtype of the type of a formal parameter or of a return value of routine *f*.
3. (Classification Adjustment) For each row *f* in the matrix *R*, set *R(f,t)* to 0 if there are any other supertypes of *t* in the same row that has been marked 1. This handles the fact that a supertype dominates the role of classification.
4. (Grouping) Collect routines into groups based on the sharing of types. Routines f_1 and f_2 are related if there exists a type t such that $R(f_1, t) = R(f_2, t) = 1$.
5. (Construction) Construct a candidate object *(F,T,D)* from each group where:

   $F = \{f \mid$ *the routine f is a member of the group*$\}$
   $T = \{t \mid$ *R(f,t) = 1 for some f in F*$\}$
   $D = \emptyset$

As was the case with the previous method, the types based object finding method can identify classes that are too large.

8.2.2 REDO - Objects

REDO (Restructuring, Maintenance, Validation and Documentation of Software Systems) is an Espirit II project whose objective is to improve applications by making them more maintainable through the use of reverse engineering techniques. The approach in reverse engineering of COBOL involve the development of general guidelines for the process of deriving objects and specifications from program code as well as providing a framework for formally reasoning about objects [4, 42, 43]. This approach also involves the identification of objects as well as the systematic specification of classes using an extension to the *Z* notation, known as *Z++*.

The approach involves three stages defined as follows

1. Translation of COBOL to UNIFORM, an intermediate language

2. Creation of outline objects through the use of data flow diagrams. Code is partitioned into single-entry, single-exit functions called *phases*. These phases are abstracted into functions.

3. Simplifying transforms are applied to the abstracted functions and the objects identified in the previous stage. The functions are incorporated into the objects and a formal specification using *Z* and *Z++* is created.

Translation from COBOL to UNIFORM

The translation to UNIFORM is performed in order to add extra information about the subject system and to eliminate redundant or obsolete programming structures.

Higher Level Abstractions

A technique known as *phasing* is used to for identifying flows of information from data structure to data structure. For example, consider the code contained in Figure 8.3 [4]. The phasing process identifies the following phases:

```
OPEN INPUT file-1;
   PERFORM process file-1;
OPEN OUTPUT file-2;
OPEN OUTPUT file-3;
OPEN INPUT file-4;
   PERFORM process file-4;
CLOSE file-1;
CLOSE file-2;
CLOSE file-3;
CLOSE file-4;
```

Figure 8.3. COBOL File Operations [4]

- file-1, file-4 → file-2, file-3
- file-4 → file-2, file-3
- file-4 → file-3

These phases identify the possible flows of information from the data structures on left hand side to data structures on the right hand side. Identifying phases in a program provides insight to the existence of objects. A technique for abstracting the functionality of program phases is used to reduce programming structures into two essential constructs [44]: functional composition and conditional expressions.

Simplification and specification

Once data types, the attributes of the data types, and the operations of the data types have been abstracted from code they are formalized by generating a specification in a formal notation. The *Z++* is an extension of the *Z* notation that adds object-oriented properties to the specification language. A *Z++* schema consists of four distinct sections for describing attributes *(OWNS)*, predicates of the properties of the internal state of a class object *(INVARIANT)*, types of operations available to a class *(OPERATIONS)*, and definitions of the operations available to a class *(ACTIONS)*.

The use of a formal notation provides advantages over the approach described in Section 8.2.1 by allowing for the formal reasoning about the objects once the specification has been created.

8.3 Comparison of Approaches

The approach to reverse engineering described in this thesis uses a translational technique based on the semantics of the strongest postcondition predicate transformer. This differs from the Functional Abstraction approach of Linger et al. in that their approach is informal and assumes that the combination of specifications constructed for small primes will result in a formal specification for a whole program.

The Maintainer's Assistant is a tool that uses a combination of transformation and knowledge based approaches. This differs greatly from the approach presented in this thesis in that much of the burden of specification construction relies on choosing transformations to apply to a program written using a WSL. The potential number of transformations needed to construct a specification for a program can be large, even for small programs.

The approach suggested by Liu and Wilde for identifying objects in program code provides the basis for the approach described in this thesis. Our approach, however,

differs in that we use formal specifications as a means for identifying objects, and the level of granularity differs greatly due to the fact that we consider structured types only.

The approach taken by the REDO project for identifying objects is tightly coupled to COBOL and UNIFORM. Their use of formal specifications in the form of *Z* and *Z++* is, however, attractive because the formal notations allow for formal reasoning.

CHAPTER 9

Conclusions and Future Investigations

The level of abstraction of specification constructed using the techniques described in this thesis are at the "as-built" level, that is, the specifications contain implementation specific information. For straight-line programs (programs without iteration or recursion) the techniques described herein are complete in that a fully automated construction of a formal specification from program code is achievable. As such, automated techniques for verifying the correctness of straight-line programs can be facilitated.

The Object Modeling Technique (OMT) is a method for modeling system requirements through the use of three complementary diagramming notations. The OMT notation is attractive because it provides an easy to use, visually-oriented notation for capturing system requirements and other types of high level information about a system. In practice, the lack of formalism in the notations can be problematic. However, a formalization of OMT in terms of algebraic specifications has recently been developed [45] for describing the states and state transitions of a system.

Future investigations will focus on three areas. First, a method of reverse engineering that encompasses all major facets of imperative programming constructs,

including iteration and recursion will be developed. Second, methods for constructing higher level abstractions from lower level abstractions will be investigated. Finally, a rigorous technique for re-engineering specifications from the imperative programming paradigm to the object-oriented programming paradigm will be created.

Iteration and recursive procedure calls pose potential difficulties since we need to construct a postcondition with respect to bound functions, loop invariants, and termination conditions. The investigations into this component will involve extending the existing rules for assignment, alternation, iteration, and procedure calls to include a generalized procedure for handling iteration and recursion.

"As-built" specifications have the property of being too tightly coupled to the programs that they describe. That is, they suffer from implementation bias. The second component of this research is an investigation into creating more abstract representations of the "as-built" specifications. Diagrams, in the form of statecharts, data flow diagrams, and object diagrams, provide an informal specification of systems and can be constructed both manually and automatically. OMT is a notation that uses each of these diagramming techniques. A formal treatment of OMT in terms of algebraic specifications coupled with the constraints provided by "as-built" specifications will be used to facilitate the task of abstracting higher level specifications from the lower level "as-built" specifications. Additionally, this integration of techniques will provide the basis for constructing high level requirements in terms of object-oriented models and algebraic specifications.

A prototype to support the construction of "as-built" specifications will be developed. As each subsequent component of the proposed investigations is performed (i.e. construction of higher levels of abstraction and creation of object-oriented models), the prototype will evolve to reflect the new capabilities. The results of the investigations will be encapsulated in the form of a prototype that will support the methods for the partial automation of the construction of object-oriented specifications by way of

a shift in paradigms. That is, the main target is to develop a tool that will aid in the construction of algebraic specifications from programs. This will involve using "as-built" specifications as constraints for identifying objects embedded in specifications of imperative program code and specifying objects using an algebraic specification language. That is, we will be performing the re-engineering of formal specifications for an imperative system into a formal specification for an object-oriented system. Using the integration described above for OMT and the formal strategy for reverse engineering, a set of transformations will be developed for reconstituting the high level specifications for the original imperative system into a high level specification for an object-oriented system.

BIBLIOGRAPHY

BIBLIOGRAPHY

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

[2] E. Byrne, "A Conceptual Foundation for Software Re-engineering," in *Proceedings for the Conference on Software Maintenance*, pp. 226–235, IEEE, 1992.

[3] B. H. Cheng and G. C. Gannod, "Abstraction of Formal Specifications from Program Code," in *Proceedings for the IEEE 3rd International Conference on Tools for Artificial Intelligence*, pp. 125–128, IEEE, 1991.

[4] H. Haughton and K. Lano, "Objects Revisited," in *Proceedings for the Conference on Software Maintenance*, pp. 152–161, IEEE, 1991.

[5] N. G. Leveson and C. S. Turner, "An Investigation of the Therac-25 Accidents," *IEEE Computer*, pp. 18–41, July 1993.

[6] V. S. Flor, "Ruling's Dicta Causes Uproar," *The National Law Journal*, July 1991.

[7] J. M. Wing, "A Specifier's Introduction to Formal Methods," *IEEE Computer*, vol. 23, pp. 8–24, September 1990.

[8] S. L. Gerhart, "Applications of formal methods: Developing virtuoso software," *IEEE Software*, vol. 7, pp. 7–10, September 1990.

[9] N. G. Leveson, "Formal Methods in Software Engineering," *IEEE Transactions on Software Engineering*, vol. 16, pp. 929–930, September 1990.

[10] R. A. Kemmerer, "Integrating Formal Methods into the Development Process," *IEEE Software*, vol. 7, pp. 37–50, September 1990.

[11] A. Hall, "Seven myths of formal methods," *IEEE Software*, vol. 7, pp. 11–19, September 1990.

[12] B. H. Cheng, "Synthesis of Procedural Abstractions from Formal Specifications," in *Proc. of COMPSAC'91*, pp. 149–154, September 1991.

[13] G. C. Gannod and B. H. Cheng, "Facilitating the Maintenance of Safety-Critical Systems Using Formal Methods," *The International Journal of Software Engineering and Knowledge Engineering*, vol. 4, no. 2, 1994.

[14] S. Liu and N. Wilde, "Identifying Objects in a Conventional Procedural Language: An Example of Data Design Recovery," in *Proceedings for the Conference on Software Maintenance*, IEEE, 1990.

[15] G. C. Gannod and B. H. Cheng, "A Two Phase Approach to Reverse Engineering Using Formal Methods," *Lecture Notes in Computer Science: Formal Methods in Programming and Their Applications*, vol. 735, pp. 335–348, July 1993.

[16] F. L. Bauer, "Software Engineering," in *Information Processing 71*, p. 530, North Holland, 1972.

[17] R. S. Pressman, *Software Engineering A Practitioner's Approach.* McGraw-Hill, 1992.

[18] W. M. Osborne and E. J. Chikofsky, "Fitting pieces to the maintenance puzzle," *IEEE Software*, vol. 7, pp. 11–12, January 1990.

[19] E. J. Chikofsky and J. H. Cross II, "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software*, vol. 7, pp. 13–17, January 1990.

[20] E. J. Byrne and D. A. Gustafson, "A Software Re-engineering Process Model," in *COMPSAC*, ACM, 1992.

[21] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, pp. 576–580, October 1969.

[22] E. W. Dijkstra and C. S. Scholten, *Predicate Calculus and Program Semantics.* Springer-Verlag, 1990.

[23] J. Spivey, *The Z Notation: A Reference Manual.* Prentice Hall, 1987.

[24] C. B. Jones, *Systematic Software Development Using VDM.* Prentice Hall International Series in Computer Science, Prentice Hall International (UK) Ltd., second ed., 1990.

[25] E. W. Dijkstra, *A Discipline of Programming.* Prentice Hall, 1976.

[26] F. Lindstrom, "Re-engineering of old systems to an object-oriented architecture," in *OOPSLA*, ACM, 1991.

[27] A. Winblad, S. Edwards, and D. King, *Object-Oriented Software.* Addison-Wesley, 1990.

[28] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design.* Englewood Cliffs, New Jersey: Prentice Hall, 1991.

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

[30] T. Pratt, *Programming Languages: Design and Implementation.* Prentice-Hall, 1984.

[31] J. Ichbiah, "Rationale for the Design of the Ada Programming Language," *SIGPLAN Notices*, vol. 14, 1979.

[32] N. Wirth, "The programming language Pascal," *Acta Informatica*, vol. 1, pp. 35–63, 1971.

[33] P. Benedusi, A. Cimitile, and U. DeCarlini, "A Reverse Engineering Methodology to Reconstruct Hierarchical Data Flow Diagrams for Software Maintenance," in *Proceedings for the Conference on Software Maintenance*, pp. 180–189, IEEE, 1989.

[34] S. Leestma and L. Nyhoff, *Pascal Programming and Problem Solving.* Macmillan, second ed., 1987.

[35] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1986.

[36] R. Linger, H. Mills, and B. Witt, *Structured Programming: Theory and Practice.* Addison-Wesley, 1979.

[37] P. Hausler, M. Pleszkock, R. Linger, and A. Hevner, "Using Function Abstraction to Understand Program Behavior," *IEEE Software*, vol. 7, pp. 55–63, January 1990.

[38] M. Ward, F. Calliss, and M. Munro, "The Maintainer's Assistant," in *Proceedings for the Conference on Software Maintenance*, IEEE, 1989.

[39] T. Bull, "An Introduction to the WSL Program Transformer," in *Proceedings for the Conference on Software Maintenance*, pp. 242–250, IEEE, 1990.

[40] F. Calliss, M. Khalil, M. Munro, and M. Ward, "A Knowledge-Based System for Software Maintenance," in *Proceedings for the Conference on Software Maintenance*, pp. 319–324, IEEE, 1988.

[41] D. R. Smith, "KIDS: A Semi-automatic Program Development System," *IEEE Transactions on Software Engineering*, vol. 16, pp. 1024–1043, September 1990.

[42] K. Lano, "Formal Approaches to Reverse-Engineering," Tech. Rep. 2487-TN-PRG-1067, Oxford University, March 1991.

[43] K. Lano, "Test Results for the Reverse-Engineering Tool Set," Tech. Rep. 2487-TN-PRG-1074, Oxford University, September 1991.

[44] K. Lano and H. Haughton, "Extracting Design and Functionality from Code," in *Proceedings for the 5th International Workshop on Computer Aided Software Engineering*, pp. 74–82, IEEE, 1992.

[45] R. H. Bourdeau and B. H. Cheng, "A Formal Semantics for the Integration of Object and Dynamic Models," Tech. Rep. CPS-93-32, Michigan State University, December 1993.