

# Strongest Postcondition Semantics as the Formal Basis for Reverse Engineering\*

Gerald C. Gannod<sup>†</sup> and Betty H. C. Cheng<sup>‡</sup>

Department of Computer Science  
Michigan State University  
East Lansing, Michigan 48824  
tel: (517) 355-8344; fax: (517) 432-1061  
{gannod,chengb}@cps.msu.edu

## Abstract

*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. Changing architectures and improvements in programming methods, including formal methods in software development and object-oriented programming, have prompted a need to reverse engineer and re-engineer program code. This paper describes the application of the strongest postcondition predicate transformer (strongest postcondition) as the formal basis for the reverse engineering of imperative program code.*

## 1 Introduction

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 [1]. Other problems caused by or due to software failure have been well documented and with the change in laws concerning liability [2], the need to reduce the number of problems due to software increases.

Software maintenance has long been a problem faced by software professionals, where the average age of software is between 10 to 15 years old [3]. With the development of new architectures and improve-

ments 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.

Formal methods in software development provide many benefits in the forward engineering aspect of software development [4]. One of the advantages of using formal methods in software development is that the formal notations are precise, verifiable, and facilitate automated processing [5]. *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.

This paper describes an approach to reverse engineering based on the formal semantics of the *strongest postcondition* predicate transformer  $sp$  [6], and the partial correctness model of program semantics introduced by Hoare [7]. Previously, we investigated the use of the *weakest precondition* predicate transformer  $wp$  as the underlying formal model for constructing formal specifications from program code [8, 9]. The difference between the two approaches is in the ability to directly apply a predicate transformer to a program (i.e.,  $sp$ ) versus using a predicate transformer as a guideline for constructing formal specifications (i.e.,  $wp$ ). The remainder of this paper is organized as follows. Section 2 provides background material for the areas of software maintenance and formal methods. The formal approach to reverse engineering based on  $sp$  is described in Sections 3 and 4, where Section 3

---

\*This work is supported in part by the National Science Foundation grants CCR-9407318, CCR-9209873, and CDA-9312389.

<sup>†</sup>This author is supported in part by a NASA Graduate Student Researchers Program Fellowship.

<sup>‡</sup>Please address all correspondences to this author.

discusses the *sp* semantics for assignment, alternation, and sequence, and Section 4 gives the *sp* semantics for iterative and procedural constructs. An example applying the reverse engineering technique is given in Section 5, while Sections 6 and 7 discuss related work, draw conclusions, and suggest future investigations.

## 2 Background

This section provides background information for the area of software maintenance and formal methods for software development. Included in this discussion is the formal model of program semantics used throughout the paper.

### 2.1 Software Maintenance

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 [10]. *Forward Engineering* is the process of developing a system by moving from high level abstract specifications to detailed, implementation-specific manifestations [10]. 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 [10]. *Restructuring* is the process of creating a logically equivalent system at the same level of abstraction [10]. This process does not require semantic understanding of the system and is best characterized by the act of transforming unstructured code into structured code. *Re-Engineering* is the examination and alteration of a system to reconstitute it in a new form, which potentially involves changes at the requirements, design, and implementation levels [10].

### 2.2 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 [11]) make use of informal techniques, thus increasing the potential for introducing ambiguity, inconsistency, and incompleteness in designs and implementations. In contrast, Formal methods used in software development are rigorous techniques for specifying, developing, and verifying computer software [4]. A formal method consists of a well-defined specification language with a set of well-defined inference rules that can be used to reason about a specification [4]. A benefit of formal methods

is that their notations are well-defined and thus, are amenable to automated processing.

**2.2.1 Program Semantics.** The notation  $Q \{ S \} R$  was originally introduced by Hoare [7] to indicate a partial correctness model of execution, where, given that a logical condition  $Q$  holds, if the execution of program  $S$  terminates, then logical condition  $R$  will hold. A rearrangement of the braces to produce  $\{ Q \} S \{ R \}$ , in contrast, represents a total correctness model of execution. That is, if condition  $Q$  holds, then  $S$  is guaranteed to terminate with condition  $R$  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 *weakest precondition*  $wp(S, R)$  describes the set of all states in which the statement  $S$  can begin execution and terminate with postcondition  $R$  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 1, produce a new predicate.

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

Table 1: Properties of the  $wp$  and  $wlp$  predicate transformers

**2.2.2 Strongest Postcondition.** Consider the predicate  $\neg wlp(S, \neg R)$ , which is the set of all states in which *there exists* an execution of  $S$  that terminates with  $R$  true. That is, we wish to describe the set of states in which satisfaction of  $R$  is possible [6]. The predicate  $\neg wlp(S, \neg R)$  is contrasted to  $wlp(S, R)$  which, is the set of states in which the computation of  $S$  either fails to terminate, or terminates with  $R$  true.

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 [6], which is the set of all states in which *there exists* a computation of  $S$  that begins with  $Q$  true. That is, given that  $Q$  holds, execution of  $S$  results in  $sp(S, Q)$  true, if  $S$  terminates (i.e.,  $sp(S, Q)$  assumes partial correctness). Finally,

we make the following observation about  $sp(S, Q)$  and  $wlp(S, R)$  and the relationship between the two predicate transformers, given the Hoare triple  $Q \{ S \} R$ :

$$\begin{aligned} Q &\Rightarrow wlp(S, R) \\ sp(S, Q) &\Rightarrow R \end{aligned}$$

The importance of this relationship is two-fold. First, it provides a formal 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  $sp$ .

**2.2.3 sp vs. wp.** Given a Hoare triple  $Q \{ S \} R$ , we note that  $wp$  is a reverse rule, in that a derivation of a specification begins with  $R$ , and produces a predicate  $wp(S, R)$ .  $wp$  assumes a total correctness model of computation, meaning that given  $S$  and  $R$ , if the computation of  $S$  begins in state  $wp(S, R)$ , the program  $S$  will halt with condition  $R$  true.

We contrast this model with the  $sp$  model, a forward derivation rule. That is, given a precondition  $Q$  and a program  $S$ ,  $sp$  derives a predicate  $sp(S, Q)$ .  $sp$  assumes a partial correctness model of computation meaning that if a program starts in state  $Q$ , then the execution of  $S$  will place the program in state  $sp(S, Q)$  if  $S$  terminates.

The use of these predicate transformers for reverse engineering have different implications. Using  $wp$  has the implication that a postcondition  $R$  is known. However, with reverse engineering we are interested in determining  $R$ , therefore  $wp$  can only be used as a guideline for performing reverse engineering. The use of  $sp$  assumes that a precondition  $Q$  is known and that a postcondition will be derived through the direct application of  $sp$ . As such,  $sp$  provides for a more natural application to reverse engineering.

### 3 Primitive Constructs

This section describes the derivation of formal specifications from the primitive programming constructs of assignment, alternation, and sequences. The Dijkstra language [12] 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, for reverse engineering purposes, describe specification derivation in terms of Hoare triples. Notationally, throughout the remainder of this paper, the notation  $\{ Q \} S \{ R \}$  will be used to indicate a partial correctness interpretation.

### 3.1 Assignment

An assignment statement has the form  $\mathbf{x} := \mathbf{e}$ ; where  $\mathbf{x}$  is a variable, and  $\mathbf{e}$  is an expression. The  $wlp$  of an assignment statement is expressed as  $wlp(\mathbf{x} := \mathbf{e}, R) = R_e^x$ , which represents the postcondition  $R$  with every free occurrence of  $x$  replaced by the expression  $e$ . This type of replacement is termed a textual substitution of  $x$  by  $e$  in expression  $R$  [13]. 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  $R_{\overline{E}}^{\overline{y}}$ , where each  $y_i$  is replaced by  $E_i$ , respectively, in expression  $R$ . The  $sp$  of an assignment statement is expressed as [6]

$$sp(\mathbf{x} := \mathbf{e}, Q) = (\exists v :: Q_v^x \wedge x = e_v^x), \quad (1)$$

where  $Q$  is the precondition,  $v$  is the previous value of  $\mathbf{x}$ , and ‘ $::$ ’ indicates that the range of the quantified variable  $v$  is not relevant in the current context.

Suppose we were to write the right hand side of Equation (1) as an infinite series of disjuncts. Then the equation might appear as

$$\begin{aligned} sp(\mathbf{x} := \mathbf{e}, Q) \equiv & (Q_{v_1}^x \wedge x = e_{v_1}^x) \vee \\ & (Q_{v_2}^x \wedge x = e_{v_2}^x) \vee \\ & \dots \vee \\ & (Q_{v_n}^x \wedge x = e_{v_n}^x) \vee \\ & \dots \end{aligned} \quad (2)$$

We could strengthen the definition of  $sp(\mathbf{x} := \mathbf{e}, Q)$  if we knew that  $(x = v) \wedge Q$  were a tautology, where  $v$  is a known initial value being held by  $x$ . That is, if we impose on a variable  $x$ , some initial value, called  $v$ , then there would be no need to write  $sp(\mathbf{x} := \mathbf{e}, Q)$  using the existential quantifier. Instead, we can write a specification for  $\mathbf{x} := \mathbf{e}$  as simply  $Q \wedge (x = e_v^x)$ . Therefore, given the imposition of initial (or previous) values on variables, the Hoare triple formulation for assignment statements is as follows:

$$\begin{aligned} \{ Q \} & \quad \text{/* precondition */} \\ x := e; & \\ \{ (x_{j+1} = e_{x_j}^x) \wedge Q \} & \text{/* postcondition */} \end{aligned}$$

where  $x_j$  represents the initial value of the variable  $x$ ,  $x_{j+1}$  is the subsequent value of  $x$ ,  $Q$  is the precondition. Subscripts are added to variables to convey historical information for a given variable.

Consider a program that consists of a series of assignments to a variable  $x$ , “ $\mathbf{x} := \mathbf{a}; \mathbf{x} := \mathbf{b}; \mathbf{x} := \mathbf{c}; \mathbf{x} := \mathbf{d}; \mathbf{x} := \mathbf{e}; \mathbf{x} := \mathbf{f}; \mathbf{x} := \mathbf{g}; \mathbf{x} := \mathbf{h};$ ” However contrived it may be, it is useful in illustrating the different ways that the effects of an assignment statement on a variable can be specified. For instance, Figure 1(a)

<pre> { x = X } x := a; { x = a ∧ X = X } x := b; { x = b ∧ a = a } x := c; { x = c ∧ b = b } x := d; { x = d ∧ c = c } x := e; { x = e ∧ d = d } x := f; { x = f ∧ e = e } x := g; { x = g ∧ f = f } x := h; { x = h ∧ g = g } ⋮ </pre>	<pre> { x<sub>0</sub> = X } x := a; { x<sub>1</sub> = a } x := b; { x<sub>2</sub> = b } x := c; { x<sub>3</sub> = c } x := d; { x<sub>4</sub> = d } x := e; { x<sub>5</sub> = e } x := f; { x<sub>6</sub> = f } x := g; { x<sub>7</sub> = g } x := h; { x<sub>8</sub> = h } ⋮ </pre>	<pre> { x<sub>0</sub> = X } x := a; { x<sub>1</sub> = a ∧ x<sub>0</sub> = X } x := b; { x<sub>2</sub> = b ∧ x<sub>1</sub> = a ∧ ... } x := c; { x<sub>3</sub> = c ∧ x<sub>2</sub> = b ∧ ... } x := d; { x<sub>4</sub> = d ∧ x<sub>3</sub> = c ∧ ... } x := e; { x<sub>5</sub> = e ∧ x<sub>4</sub> = d ∧ ... } x := f; { x<sub>6</sub> = f ∧ x<sub>5</sub> = e ∧ ... } x := g; { x<sub>7</sub> = g ∧ x<sub>6</sub> = f ∧ ... } x := h; { x<sub>8</sub> = h ∧ x<sub>7</sub> = g ∧ ... } ⋮ </pre>
(a) Code with strict <i>sp</i> application	(b) Code with historical subscripts	(c) Code with historical subscripts and propagation

Figure 1: Different approaches to specifying the history of a variable

depicts the specification of the program by strict application of the strongest postcondition.

Another possible way to specify the program is through the use of *historical* subscripts for a variable. A historical subscript is an integer number used to denote the *i*<sup>th</sup> textual assignment to a variable, where a textual assignment is an occurrence of an assignment statement in the program source (versus the number of times the statement is executed). An example of the use of historical subscripts is given in Figure 1(b). However, when using historical subscripts, special care must be taken to ensure the correctness of the specification when other programming constructs are introduced. That is, using the technique shown in Figure 1(b) is not sufficient. We must propagate the precondition of a given statement to the postcondition, as shown in Figure 1(c). The main motivation for using histories is to remove the need to apply textual substitution to a complex precondition and to provide historical context to complex disjunctive and conjunctive expressions. The disadvantage to using such a technique is that the propagation of the precondition can potentially be complex visually. Note that we have not changed the semantics of the strongest postcondition, but rather in our application of strongest postcondition, we append extra information that provides a historical context to all variables of a program during some “snapshot” or state of a program.

### 3.2 Alternation

An alternation statement using the Dijkstra language [12] is expressed as

```

if
  B1 → S1;
  ...
  || Bn → Sn;
fi;

```

where  $B_i \rightarrow S_i$  is a guarded command such that  $S_i$  is only executed if logical expression (guard)  $B_i$  is true. The *wlp* for alternation statements is given by [6]

$$wlp(\text{IF}, R) \equiv (\forall i : B_i : wlp(S_i, R)),$$

where **IF** represents the alternation statement. The equation states that the necessary condition to satisfy  $R$ , if the alternation statement terminates, is that given  $B_i$  is *true*, the *wlp* for each guarded statement  $S_i$  with respect to  $R$  holds. The *sp* for alternation has the form [6]

$$sp(\text{IF}, Q) \equiv (\exists i :: sp(S_i, B_i \wedge Q)). \quad (3)$$

The existential expression can be expanded into the following form

$$sp(\text{IF}, Q) \equiv (sp(S_1, B_1 \wedge Q) \vee \dots \vee sp(S_n, B_n \wedge Q)). \quad (4)$$

Expression (4) illustrates the disjunctive nature of alternation statements where each disjunct describes the postcondition in terms of both the precondition  $Q$  and the guard and guarded command pairs, given by  $B_i$  and  $S_i$ , respectively. This characterization follows the intuition that a statement  $S_i$  is only executed if  $B_i$  is true, and that only one of  $S_j$ ,  $1 \leq j \leq n$ , is executed. The translation of alternation statements follows accordingly from Expression (4). Using the Hoare triple notation, a specification is constructed as follows

```

{ Q }
if
  B1 → S1;
  ...
  || Bn → Sn;
fi;
{ (sp(S1, B1 ∧ Q) ∨ ... ∨ sp(Sn, Bn ∧ Q)) ∧ Q }

```

### 3.3 Sequence

For a given sequence of statements  $S_1; \dots; S_n$ , it follows that the postcondition for some statement  $S_i$  is the precondition for some subsequent statement  $S_{i+1}$ . The *wlp* and *sp* for sequences follow accordingly. The *wlp* for sequences is defined as follows [6]:

$$wlp(S_1; S_2, R) \equiv wlp(S_1, wlp(S_2, R)).$$

Conversely, the *sp* [6] is

$$sp(S_1; S_2, Q) \equiv sp(S_2, sp(S_1, Q)). \quad (5)$$

In the case of *wlp*, the set of states for which the sequence  $S_1; S_2$  can execute with  $R$  true (if the sequence terminates) is equivalent to the *wlp* of  $S_1$  with respect to the set of states defined by *wlp*( $S_2, R$ ). For *sp*, the derived state for the sequence  $S_1; S_2$  with respect to

the precondition  $Q$  is equivalent to the derived postcondition for  $S_2$  with respect to a precondition given by  $sp(S_1, Q)$ . The Hoare triple formulation and construction process is as follows.

$$\begin{array}{l} \{ Q \} \\ S_1; \\ \{ sp(S_1, Q) \wedge Q \} \\ S_2; \\ \{ sp(S_2, sp(S_1, Q) \wedge Q) \wedge sp(S_1, Q) \wedge Q \}. \end{array}$$

## 4 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 necessary because the formalisms for the  $wlp$  and  $sp$  for iteration are defined in terms of recursive functions [6, 13] that are, in general, difficult to practically apply.

### 4.1 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 more general terms, 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 determined. 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 construction of loop invariants [13]. 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 2 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 in line with the total correctness model, 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<sup>1</sup> is prompted for the proper specification of the statement. The following items are then identified in order to specify the loop:

- *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  $BB = B_1 \vee \dots \vee B_n$  and  $R$  is the postcondition).

### 4.2 Procedural Abstractions

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

```
proc p ( value  $\bar{x}$ ; value-result  $\bar{y}$ ; result  $\bar{z}$  );
    {P} { body } {Q}
```

where  $\bar{x}$ ,  $\bar{y}$ , and  $\bar{z}$  represent the **value**, **value-result**, and **result** parameters for the procedure, respectively. A parameter of type **value** means that the parameter

<sup>1</sup>A domain expert is a maintenance engineer who is familiar with the subject system and application area.

- 
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 2: Steps for abstracting the effect of iteration statements

---

is used only for input to the procedure. Likewise, a parameter of type **result** indicates that the parameter is used only for output from the procedure. Parameters that are known as **value-result** indicate that the parameters can be used for both input and output to the procedure. The notation  $\langle body \rangle$  represents one or more statements making up the “procedure”, while  $\{P\}$  and  $\{Q\}$  are the precondition and postcondition, respectively. The *signature* of a procedure appears as

$$\text{proc } p : (input\_type)^* \rightarrow (output\_type)^* \quad (6)$$

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

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

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 [13] using a total correctness model of execution. Given a procedure declaration of the above

form, the following condition holds [13]

$$\{PRT : P_{\bar{a}, \bar{b}}^{\bar{x}, \bar{y}} \wedge (\forall \bar{u}, \bar{v} :: Q_{\bar{u}, \bar{v}}^{\bar{y}, \bar{z}} \Rightarrow R_{\bar{u}, \bar{v}}^{\bar{b}, \bar{c}})\} p(\bar{a}, \bar{b}, \bar{c}) \{R\} \quad (7)$$

for a procedure call  $p(\bar{a}, \bar{b}, \bar{c})$ , where  $\bar{a}$ ,  $\bar{b}$ , and  $\bar{c}$  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  $\bar{u}$  and  $\bar{v}$ , 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 for 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 of Equation (7) 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  $p(\bar{a}, \bar{b}, \bar{c})$  can be represented by the Pascal-like block [13] found in Figure 3 where  $\langle body \rangle$  com-

---

```

begin
  ...
  { PR }
  p( $\bar{a}, \bar{b}, \bar{c}$ )
  { R }
  ...
end

↓

begin
  declare  $\bar{x}, \bar{y}, \bar{z}, \bar{u}, \bar{v}$ ;
  ...
  { PR }
   $\bar{x}, \bar{y} := \bar{a}, \bar{b}$ ;
  { P }
   $\langle body \rangle$ 
  { Q }
   $\bar{y}, \bar{z} := \bar{u}, \bar{v}$ ;
  { QR }
   $\bar{b}, \bar{c} := \bar{y}, \bar{z}$ ;
  { R }
  ...
end

```

Figure 3: Removal of procedure call  $p(\bar{a}, \bar{b}, \bar{c})$  abstraction

---

prises the statements of the procedure declaration for  $p$ ,  $\{ PR \}$  is the precondition for the call to procedure  $p$ ,  $\{ P \}$  is the specification of the program after the formal parameters have been replaced by actual parameters,  $\{ Q \}$  is the specification of the program after the procedure has been executed,  $\{ QR \}$  is the specification of the program after formal parameters have been assigned with the values of local variables, and  $\{ R \}$  is the specification of the program after the actual parameters to the procedure call have been “returned”. 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 straight-line programs to include non-recursive straight-line procedures. Making the appropriate  $sp$  substitutions, we can annotate the code sequence from Figure 3 to appear as follows:

$$\begin{aligned}
& \{ PR \} \\
& \bar{x}, \bar{y} := \bar{a}, \bar{b}; \\
& \{ P: \exists \bar{\alpha}, \bar{\beta} :: PR_{\bar{\alpha}, \bar{\beta}}^{\bar{x}, \bar{y}} \wedge \bar{x} = \bar{a}_{\bar{\alpha}, \bar{\beta}}^{\bar{x}, \bar{y}} \wedge \bar{y} = \bar{b}_{\bar{\alpha}, \bar{\beta}}^{\bar{x}, \bar{y}} \} \\
& \langle body \rangle \\
& \{ Q \} \\
& \bar{y}, \bar{z} := \bar{u}, \bar{v}; \\
& \{ QR: \exists \bar{\gamma}, \bar{\zeta} :: Q_{\bar{\gamma}, \bar{\zeta}}^{\bar{y}, \bar{z}} \wedge \bar{y} = \bar{u}_{\bar{\gamma}, \bar{\zeta}}^{\bar{y}, \bar{z}} \wedge \bar{z} = \bar{v}_{\bar{\gamma}, \bar{\zeta}}^{\bar{y}, \bar{z}} \} \\
& \bar{b}, \bar{c} := \bar{y}, \bar{z}; \\
& \{ R: \exists \bar{\vartheta}, \bar{\varphi} :: QR_{\bar{\vartheta}, \bar{\varphi}}^{\bar{b}, \bar{c}} \wedge \bar{b} = \bar{y}_{\bar{\vartheta}, \bar{\varphi}}^{\bar{b}, \bar{c}} \wedge \bar{c} = \bar{z}_{\bar{\vartheta}, \bar{\varphi}}^{\bar{b}, \bar{c}} \}
\end{aligned}$$

where  $\bar{\alpha}$ ,  $\bar{\beta}$ ,  $\bar{\gamma}$ ,  $\bar{\zeta}$ ,  $\bar{\vartheta}$ , and  $\bar{\varphi}$  are the initial values of  $\bar{x}$ ,  $\bar{y}$  (before execution of the procedure body),  $\bar{y}$  (after execution of the procedure body),  $\bar{z}$ ,  $\bar{b}$ , and  $\bar{c}$ , respectively. Recall that in Section 3.1, we described how the existential operators and the textual substitution could be removed from the calculation of the  $sp$ . Applying that technique to assignments and recognizing that formal and actual **result** parameters have no initial values, and that local variables are used to compute the values of the the **value-result** parameters, the above sequence can be simplified using the semantics of  $sp$  for assignments to obtain the following annotated code sequence:

$$\begin{aligned}
& \{ PR \} \\
& \bar{x}, \bar{y} := \bar{a}, \bar{b}; \\
& \{ P: PR \wedge \bar{x} = \bar{a} \wedge \bar{y} = \bar{b} \} \\
& \langle body \rangle \\
& \{ Q \} \\
& \bar{y}, \bar{z} := \bar{u}, \bar{v}; \\
& \{ QR: Q \wedge \bar{y} = \bar{u} \wedge \bar{z} = \bar{v} \} \\
& \bar{b}, \bar{c} := \bar{y}, \bar{z}; \\
& \{ R: QR \wedge \bar{b} = \bar{y} \wedge \bar{c} = \bar{z} \}
\end{aligned}$$

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

## 5 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 Figure 4, has four procedures, including three different implementations of “swap”. AUTOSPEC [8, 9, 14] is a tool that we have developed

---

```

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;
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;

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.

```

Figure 4: Example Pascal program

---

to support the derivational approach to the reverse engineering of formal specifications from program code.

Figures 5 and 6 depict the output of AUTOSPEC when applied to the program code given in Figure 4 where the notation `id{scope}instance` is used to indicate a variable `id` with scope defined by the refer-

encing environment for `scope`. The `instance` identi-

```

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
  I: (* (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
  J: (* (Max{2}1 = NumTwo0 & Min{2}1 = NumOne0) & U *)
  K: (* (((NumOne0 > NumTwo0) &
      (Max{0}1 = NumOne0 & Min{0}1 = NumTwo0)) |
      (not(NumOne0 > NumTwo0) &
      (Max{0}1 = NumTwo0 & Min{0}1 = NumOne0))) & U *)
  end
  L: (* (((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
M: (* (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
N: (* (Y{0}1 = X0 & X{0}1 = Y0 & temp{0}1 = X0) & U *)

```

Figure 5: Output created by applying AUTOSPEC to example

fier 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. For instance,

if a call to some arbitrary procedure called `foo` is invoked, then specifications for variables local to `foo` are labeled with an integer scope. Upon return, the specification of the calling procedure will have references to variables local to `foo`. Although the variables being referenced are outside the scope of the calling procedure, a specification of the input and output parameters for `foo` can provide valuable information, such as the logic used to obtain the specification for the output variables to `foo`. As such, in the specification for the variables local to `foo` but outside the scope of the calling procedure, we use the scope label `So`. Therefore, if we have a variable `q` local to `foo`, it might appear in a specification outside its local context as `q{foo}4`, where “4” indicates the fourth instance of variable `q` in the context of `foo`.

In Figure 5, the code for the procedure `FindMaxMin` contains an alternation statement, where lines `I`, `J`, `K`, and `L` specify the guarded commands of the alternation statement (`I` and `J`), the effect of the alternation statement (`K`), and the effect of the entire procedure (`L`), respectively.

Of particular interest are the specifications for the swap procedures given in Figure 5 named `swapa` and `swapb`. The variables `X` and `Y` are specified using the notation described above. As such, the first assignment to `Y` is written using `Y{0}1`, where `Y` is the variable, `{0}` describes the level of nesting (here it is zero), and `1` is the historical subscript, the one indicating the first instance of `Y` after the initial value. The final comment for `swapa` (Line `M`), which gives the specification for the entire procedure, reads as:

```
(* (Y{0}2 = X0 & X{0}1 = Y0 & Y{0}1 = Y0 + X0) & U *)
```

where `Y{0}2 = X0` is the specification of the final value of `Y`, and `X{0}1 = Y0` is the specification of the final value of `X`. In this case, the intermediate value of `Y`, denoted `Y{0}1`, with value `Y0 + X0` is not considered in the final value of `Y`.

Procedure `swapb` uses a temporary variable algorithm for swap. Line `N` is the specification after the execution of the last line and reads as:

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

where `Y{0}1 = X0` is the specification of the final value of `Y`, and `X{0}1 = Y0` is the specification of the final value of `X`.

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` with respect to the final values of the variables `X` and `Y`.

In addition, Figure 6 shows the formal specification of the `funnyswap` procedure. The semantics for the `funnyswap` procedure are similar to that of `swapb`. However, the parameter passing scheme used in this procedure is pass by value.

The specification of the main `begin-end` block of the program `MaxMin` is given in Figure 6. There are eight lines of interest, labeled **I**, **J**, **K**, **L**, **M**, **N**, **O**, and **P**, respectively. Lines **I** and **J** specify the effects of assignment statements. The specification at line **K** demonstrates the use of identifier scope labels, where in this case, we see the specification of variables `X` and `Y` from the context of `swapa`. Line **L** is another example of the same idea, where the specification of variables from the context of `swapb` (`X` and `Y`), are given. In the main program, no variables local to the scope of the call to `funnyswap` are affected by `funnyswap` due to the pass by value nature of `funnyswap`, and thus the specification shows no change in variable values, which is shown by line **M** of Figure 6. The effects of the call to procedure `FindMaxMin` provides another example of the specification of a procedure call (line **N**). Finally, line **P** is the specification of the entire program, with every precondition propagated to the final postcondition as described in Section 3.1. Notice here that we are concerned with the final value of the variables that are local to the program `MaxMin` (i.e., `a`, `b`, and `c`). Thus, according to our rules for historical subscripts, we are interested in the values of `a{0}3`, `b{0}3`, and `c{0}1`. In addition, by propagating the preconditions for each statement, we can analyze the logic that was used to obtain the values for the variables of interest.

## 6 Related Work

Previously, formal approaches to reverse engineering have used the semantics of the weakest precondition predicate transformer  $wp$  as the underlying formalism of their technique. The *Maintainer's Assistant* uses a knowledge-based transformational approach to construct formal specifications from program code via the use of a Wide-Spectrum Language (WSL) [15]. A WSL is a language that uses both specification and imperative language constructs. A knowledge-base manages the correctness preserving transformations of concrete, implementation constructs in a WSL to abstract specification constructs in the same WSL.

REDO [16] (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 used to reverse engineering COBOL involves the development of general guidelines for the process

---

```

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 *)

(* Main Program for MaxMin *)
begin
  a := 5;
I:  (* a{0}1 = 5 & U *)

  b := 10;
J:  (* b{0}1 = 10 & U *)

  swapa(a,b)
K:  (* (b{0}2 = 5 &
      (a{0}2 = 10 &
      (Y{swapa}2 = 5 &
      (X{swapa}1 = 10 & Y{swapa}1 = 15)))) & U *)

  swapb(a,b)
L:  (* (b{0}3 = 10 &
      (a{0}3 = 5 &
      (Y{swapb}1 = 10 &
      (X{swapb}1 = 5 & temp{swapb}1 = 10)))) & U *)

  funnyswap(a,b)
M:  (* (Y{funnyswap}1 = 5 & X{funnyswap}1 = 10 &
      temp{funnyswap}1 = 5) & U *)
  FindMaxMin(a,b,Largest,Smallest)
N:  (* (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;
O:  (* c{0}1 = Max{FindMaxMin}1 & U *)

end
P:  (* ((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 6: Output created by applying AUTOSPEC to example (cont.)

---

of deriving objects and specifications from program code as well as providing a framework for formally reasoning about objects [17].

In each of these approaches, the applied formalisms are based on the semantics of the *weakest precondition* predicate transformer  $wp$ . Some differences in applying  $wp$  and  $sp$  are that  $wp$  is a backward rule for program semantics and assumes a total correctness model of execution. However, the total correctness interpretation has no forward rule (i.e. no *strongest total postcondition stp* [6]). By using a partial correctness model of execution, both a forward rule ( $sp$ ) and backward rule ( $wlp$ ) can be used to verify and refine formal specifications generated by program understanding and reverse engineering tasks. The main difference between the two approaches is the ability to directly apply the strongest postcondition predicate transformer to code to construct formal specifications versus using the weakest precondition predicate transformer as a guideline for constructing formal specifications.

## 7 Conclusions

Formal methods provide many benefits in the development of software. Automating the process of abstracting formal specifications from program code is sought but, unfortunately, not completely realizable as of yet. However, by providing the tools that support the reverse engineering of software, much can be learned about the functionality of a system.

The level of abstraction of specification constructed using the techniques described in this paper 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 can be applied in order to obtain a formal specification from program code. As such, automated techniques for verifying the correctness of straight-line programs can be facilitated.

Current investigations focus on three areas. First, a method of reverse engineering that encompasses all major facets of imperative programming constructs, including iteration and recursion is being developed. To this end, we are in the process of defining the formal semantics of a widely used programming language and are applying our techniques to a NASA command and control application. Second, methods for constructing higher level abstractions from lower level abstractions is being investigated. Finally, a rigorous technique for re-engineering specifications from the imperative programming paradigm to the object-oriented programming paradigm will be created [14].

## References

- [1] N. G. Leveson and C. S. Turner, “An Investigation of the Therac-25 Accidents,” *IEEE Computer*, pp. 18–41, July 1993.
- [2] V. S. Flor, “Ruling’s Dicta Causes Uproar,” *The National Law Journal*, July 1991.
- [3] W. M. Osborne and E. J. Chikofsky, “Fitting pieces to the maintenance puzzle,” *IEEE Software*, vol. 7, pp. 11–12, January 1990.
- [4] J. M. Wing, “A Specifier’s Introduction to Formal Methods,” *IEEE Computer*, vol. 23, pp. 8–24, September 1990.
- [5] B. H. C. Cheng, “Applying formal methods in automated software development,” *Journal of Computer and Software Engineering*, vol. 2, no. 2, pp. 137–164, 1994.
- [6] E. W. Dijkstra and C. S. Scholten, *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.
- [7] C. A. R. Hoare, “An axiomatic basis for computer programming,” *Communications of the ACM*, vol. 12, pp. 576–580, October 1969.
- [8] B. 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.
- [9] 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.
- [10] E. J. Chikofsky and J. H. Cross II, “Reverse Engineering and Design Recovery: A Taxonomy,” *IEEE Software*, vol. 7, pp. 13–17, January 1990.
- [11] E. Yourdon and L. Constantine, *Structured Analysis and Design: Fundamentals Discipline of Computer Programs and System Design*. Yourdon Press, 1978.
- [12] E. W. Dijkstra, *A Discipline of Programming*. Prentice Hall, 1976.
- [13] D. Gries, *The Science of Programming*. Springer-Verlag, 1981.
- [14] G. C. Gannod and B. 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.
- [15] M. Ward, F. Calliss, and M. Munro, “The Maintainer’s Assistant,” in *Proceedings for the Conference on Software Maintenance*, IEEE, 1989.
- [16] K. Lano and P. Breuer, “From Programs to Z Specifications,” in *Z User Workshop* (J. E. Nicholls, Ed.), pp. 46–70, Springer-Verlag, 1989.
- [17] H. Haughton and K. Lano, “Objects Revisited,” in *Proceedings for the Conference on Software Maintenance*, pp. 152–161, IEEE, 1991.