The Journal of Automated Software Engineering, Vol. 3, No. 1/2, Kluwer Academic Publishers, 1996

, , 1–27 ()

Strongest Postcondition Semantics as the Formal Basis for Reverse Engineering*

GERALD C. GANNOD** AND BETTY H.C. CHENG† {gannod,chengb}@cps.msu.edu
Department of Computer Science
Michigan State University
East Lansing, Michigan 48824-1027

Editor:

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 (sp) as the formal basis for the reverse engineering of imperative program code.

Keywords: formal methods, formal specification, reverse engineering, software maintenance

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 [17]. Other problems caused by or due to software failure have been well documented and with the change in laws concerning liability [8], 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 [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.

Formal methods in software development provide many benefits in the forward engineering aspect of software development [20]. One of the advantages of using formal methods in software development is that the formal notations are precise,

* This work is supported in part by the National Science Foundation grants CCR-9407318, CCR-9209873, and CDA-9312389.
** This author is supported in part by a NASA Graduate Student Researchers Program Fellowship.
† Please address all correspondences to this author.

verifiable, and facilitate automated processing [3]. *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 [10, 16, 19].

This paper describes an approach to reverse engineering based on the formal semantics of the *strongest postcondition* predicate transformer *sp* [7], and the partial correctness model of program semantics introduced by Hoare [13]. Previously, we investigated the use of the *weakest precondition* predicate transformer *wp* as the underlying formal model for constructing formal specifications from program code [4, 10]. 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 software maintenance and formal methods. The formal approach to reverse engineering based on *sp* is described in Sections 3 and 4, where Section 3 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. Related work is discussed in Section 6. Finally, Section 7 draws conclusions, and suggest futures investigations.

2. Background

This section provides background information for 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 significant obstacles to successfully re-engineering a system.

Several terms are frequently used in the discussion of re-engineering [5]. *Forward Engineering* is the process of developing a system by moving from high level abstract specifications to detailed, implementation-specific manifestations [5]. 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 [5]. *Restructuring* is the process of creating a logically equivalent system at the same level of abstraction [5]. This process does not require semantic understanding of the system and is best characterized by the task 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 [5].

Byrne described the re-engineering process using a graphical model similar to the one shown in Figure 1 [1, 2]. The process model appears in the form of two sectioned triangles, where each section in the triangles represents a different level of abstraction. The higher levels in the model are *concepts* and *requirements*. The lower levels include *designs* and *implementations*. The relative size of each of the sections is intended to represent the amount of information known about a system at a given level of abstraction. Entry into this re-engineering process model begins with system A, where *Abstraction* (or reverse engineering) is performed to an appropriate level of detail. The next step is *Alteration*, where the system is constituted into a new form at a different level of abstraction. Finally, *Refinement* of the new form into an implementation can be performed to create system B.

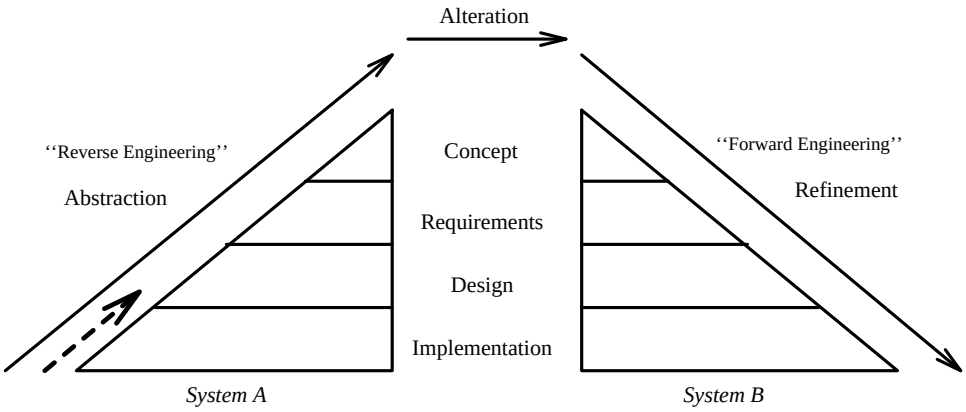


Figure 1. Reverse Engineering Process Model

This paper describes an approach to reverse engineering that is applicable to the *implementation* and *design* levels. In Figure 1, the context for this paper is represented by the dashed arrow. That is, we address the construction of formal low-level or "*as-built*" design specifications. The motivation for operating in such an implementation-bound level of abstraction is that it provides a means of traceability between the program source code and the formal specifications constructed using the techniques described in this paper. This traceability is necessary in order to facilitate technology transfer of formal methods. That is, currently existing development teams must be able to understand the relationship between the source code and the specifications.

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 [21]) 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 [20]. 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 [20]. A benefit of formal methods is that their notations are well-defined and thus, are amenable to automated processing [3].

2.2.1. Program Semantics

The notation $Q \{ S \} R$ [13] is used to represent 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.

Table 1. Properties of the *wp* and *wlp* predicate transformers

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

The context for our investigations is that we are reverse engineering systems that have desirable properties or functionality that should be preserved or extended. Therefore, the partial correctness model is sufficient for these purposes.

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 [7]. 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 [7], 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. As such, $sp(S, Q)$ 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 $Q \{ S \} R$ [7]:

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

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 wlp and sp in tandem.

2.2.3. sp vs. wp

Given a Hoare triple $Q \{ S \} R$, we note that wp is a backward rule, in that a derivation of a specification begins with R, and produces a predicate $wp(S, R)$. The predicate transformer 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 $sp(S, Q)$. The predicate transformer 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. Figure 2 gives a pictoral depiction of the differences between sp and wp, where the input to the predicate transformer produces the corresponding predicate. Figure 2(a) gives the case where the input to the predicate transformer is "S" and "R", and the output to the predicate transformer (given by the box and appropriately named "wp") is "wp(S,R)". The sp case (Figure 2(b)) is similar, where the input to the predicate transformer is "S" and "Q", and the output to the transformer is "sp(S,Q)".

The use of these predicate transformers for reverse engineering have different implications. Using wp implies that a postcondition R is known. However, with

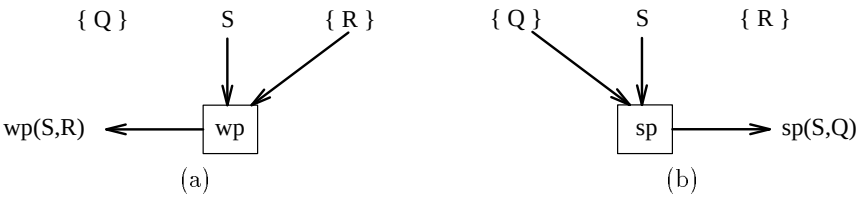


Figure 2. Black box representation and differences between *wp* and *sp*: (a) *wp* (b) *sp*

respect to reverse engineering, determining R is the objective, 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* is more applicable 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 guarded command language [6] 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 `x:= e;` where `x` is a variable, and `e` is an expression. The *wlp* of an assignment statement is expressed as $wlp(\texttt{x:=e}, R) = R^x_e$, 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. 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 follows [7]

$$sp(\texttt{x:=e}, Q) = (\exists v :: Q^x_v \wedge x = e^x_v), \tag{1}$$

where Q is the precondition, v is the quantified variable, and '::' indicates that the range of the quantified variable v is not relevant in the current context.

We 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, performing the textual substitution Q_v^x in Expression (1) is a redundant operation if, initially, Q has a conjunct of the form $x = v$. Refer to Appendix A where this case is described in more depth. Given the imposition of initial (or previous) values on variables, the Hoare triple formulation for assignment statements is as follows:

$\{Q\}$ `/* precondition */`
`x := e;`
$\{(x_{j+1} = e^x_{x_j}) \wedge Q\}$ `/* postcondition */`

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, "`x := a; x:= b; x:= c; x:= d; x:= e; x:= f; x:= g; x:= h;`" Despite its sim-

$\{ x = X \}$	$\{ x_0 = X \}$	$\{ x_0 = X \}$
`x := a;`	`x := a;`	`x := a;`
$\{ x = a \wedge X = X \}$	$\{ x_1 = a \}$	$\{ x_1 = a \wedge x_0 = X \}$
`x := b;`	`x := b;`	`x := b;`
$\{ x = b \wedge a = a \}$	$\{ x_2 = b \}$	$\{ x_2 = b \wedge x_1 = a \wedge \ldots \}$
`x := c;`	`x := c;`	`x := c;`
$\{ x = c \wedge b = b \}$	$\{ x_3 = c \}$	$\{ x_3 = c \wedge x_2 = b \wedge \ldots \}$
`x := d;`	`x := d;`	`x := d;`
$\{ x = d \wedge c = c \}$	$\{ x_4 = d \}$	$\{ x_4 = d \wedge x_3 = c \wedge \ldots \}$
`x := e;`	`x := e;`	`x := e;`
$\{ x = e \wedge d = d \}$	$\{ x_5 = e \}$	$\{ x_5 = e \wedge x_4 = d \wedge \ldots \}$
`x := f;`	`x := f;`	`x := f;`
$\{ x = f \wedge e = e \}$	$\{ x_6 = f \}$	$\{ x_6 = f \wedge x_5 = e \wedge \ldots \}$
`x := g;`	`x := g;`	`x := g;`
$\{ x = g \wedge f = f \}$	$\{ x_7 = g \}$	$\{ x_7 = g \wedge x_6 = f \wedge \ldots \}$
`x := h;`	`x := h;`	`x := h;`
$\{ x = h \wedge g = g \}$	$\{ x_8 = h \}$	$\{ x_8 = h \wedge x_7 = g \wedge \ldots \}$
$\vdots$	$\vdots$	$\vdots$
(a) Code with strict *sp* application	(b) Code with historical subscripts	(c) Code with historical subscripts and propagation

Figure 3. Different approaches to specifying the history of a variable

plicity, the example is useful in illustrating the different ways that the effects of

an assignment statement on a variable can be specified. For instance, Figure 3(a) 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^{th} 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 3(b). However, when using historical subscripts, special care must be taken to maintain the consistency of the specification with respect to the semantics of other programming constructs. That is, using the technique shown in Figure 3(b) is not sufficient. The precondition of a given statement must be propagated to the postcondition, as shown in Figure 3(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 the application of strongest postcondition, extra information is appended 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 guarded command language [6] is expressed as

$$
\begin{array}{l}
\texttt{if} \\
\qquad \texttt{B}_1 \rightarrow \texttt{S}_1; \\
\qquad \ldots \\
\qquad [\!] \; \texttt{B}_n \rightarrow \texttt{S}_n; \\
\texttt{fi};
\end{array}
$$

where $\texttt{B}_i \rightarrow \texttt{S}_i$ is a guarded command such that S_i is only executed if logical expression (guard) $\texttt{B}_i$ is true. The *wlp* for alternation statements is given by [7]:

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

where `IF` represents the alternation statement. The equation states that the necessary condition to satisfy R, if the alternation statement terminates, is that given $\texttt{B}_i$ is *true*, the *wlp* for each guarded statement S_i with respect to R holds. The *sp* for alternation has the form [7]

$$sp(\texttt{IF}, Q) \equiv (\exists i :: sp(\texttt{S}_i, \texttt{B}_i \wedge Q)). \tag{2}$$

The existential expression can be expanded into the following form

$$sp(\texttt{IF}, Q) \equiv (sp(\texttt{S}_1, \texttt{B}_1 \wedge Q) \vee \ldots \vee sp(\texttt{S}_n, \texttt{B}_n \wedge Q)). \tag{3}$$

Expression (3) 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 $\texttt{B}_i$ and $\texttt{S}_i$, respectively. This characterization follows the intuition that a statement $\texttt{S}_i$ is only executed if $\texttt{B}_i$ is true. The translation of alternation statements to specifications is based on the similarity of the semantics of Expression (3) and the execution behaviour for alternation statements. Using the Hoare triple notation, a specification is constructed as follows

$$\begin{array}{l}
\{\ Q\ \} \\
\texttt{if} \\
\qquad \texttt{B}_1 \rightarrow \texttt{S}_1; \\
\qquad \ldots \\
\qquad [\!] \ \texttt{B}_n \rightarrow \texttt{S}_n; \\
\texttt{fi}; \\
\{\ sp(\texttt{S}_1, B_1 \wedge Q) \vee \ldots \vee sp(\texttt{S}_n, B_n \wedge Q)\ \}
\end{array}$$

3.3. Sequence

For a given sequence of statements $\texttt{S}_1; \ldots; \texttt{S}_n$, it follows that the postcondition for some statement $\texttt{S}_i$ is the precondition for some subsequent statement $\texttt{S}_{i+1}$. The *wlp* and *sp* for sequences follow accordingly. The *wlp* for sequences is defined as follows [7]:

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

Likewise, the *sp* [7] is

$$sp(\texttt{S}_1;\texttt{S}_2, Q) \equiv sp(\texttt{S}_2, sp(\texttt{S}_1, Q)). \tag{4}$$

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

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

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 enables more compactness and abstraction in program development. 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 [7, 11] 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
        B_1 → S_1;
        ...
        || B_n → S_n;
od;
```

In more general terms, the iteration statement may contain any number of guarded commands of the form $\mathtt{B}_i \rightarrow \mathtt{S}_i$, such that the loop is executed as long as any guard B_i is true. A simplified form of repetition is given by "`do B → S od`".

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.

Using the abbreviated form of repetition "`do B → S od`", the semantics for iteration in terms of the weakest liberal precondition predicate transformer *wlp* is given by the following [7]:

$$wlp(\mathtt{DO}, R) \equiv (\forall i : 0 \leq i : wlp(\mathtt{IF}^i, B \vee R)), \tag{5}$$

where the notation "$\mathtt{IF}^i$" is used to indicate the execution of "`if B → S fi`" i times. Operationally, Expression (5) states that the weakest condition that must hold in order for the execution of an iteration statement to result with R true, provided that the iteration statement terminates, is equivalent to a conjunctive expression

where each conjunct is an expression describing the semantics of executing the loop i times, where $i \geq 0$.

The strongest postcondition semantics for repetition has a similar but notably distinct formulation [7]:

$$sp(\texttt{DO}, Q) \equiv \neg B \wedge (\exists i : 0 \leq i : sp(\texttt{IF}^i, Q)). \tag{6}$$

Expression (6) states that the strongest condition that holds after executing an iterative statement, given that condition Q holds, is equivalent to the condition where the loop guard is false ($\neg B$), and a disjunctive expression describing the effects of iterating the loop i times, where $i \geq 0$.

Although the semantics for repetition in terms of strongest postcondition and weakest liberal precondition are less complex than that of the weakest precondition [7], the recurrent nature of the closed forms make the application of such semantics difficult. For instance, consider the counter program "`do i < n` $\rightarrow$ `i := i + 1 od`". The application of the *sp* semantics for repetition leads to the following specification:

$$sp(\texttt{do i < n} \rightarrow \texttt{ i := i + 1 od}, Q) \equiv (i \geq n) \wedge (\exists j : 0 \leq j : sp(IF^j, Q)).$$

The closed form for iteration suggests that the loop be unrolled j times. If j is set to $n - start$, where $start$ is the initial value of variable i, then the unrolled version of the loop would have the following form:

```
1.      i:= start;
2.      if
3.          i < n --> i:= i + 1;
4.      fi
5.      if
6.          i < n --> i:= i + 1;
7.      fi
8.      ...
9.      if
10.         i < n --> i:= i + 1;
11.     fi
```

Application of the rule for alternation (Expression (2)) yields the sequence of annotated code shown in Figure 4, where the goal is to derive

$$sp(\texttt{do i < n} \rightarrow \texttt{ i := i + 1 od}, (start < n) \wedge (i = start)).$$

In the construction of specifications of iteration statements, knowledge must be introduced by a human specifier. For instance, in line 19 of Figure 4 the inductive assertion that "$i = start + (n - start - 1)$" is made. This assertion is based on a specifier providing the information that $(n - start - 1)$ additions have been performed

1. $\{\ (i = I) \wedge (start < n)\ \}$
2. `i:= start;`
3. $\{\ (i = start) \wedge (start < n)\ \}$
4. `if i < n -> i:= i + 1 fi`
5. $\{\ sp(i := i + 1, (i < n) \wedge (i = start) \wedge (start < n))$
6. $\quad \vee$
7. $\quad ((i >= n) \wedge (i = start) \wedge (start < n))$
8. $\quad \equiv$
9. $\quad ((i = start + 1) \wedge (start < n))\ \}$
10. `if i < n -> i:= i + 1 fi`
11. $\{\ sp(i := i + 1, (i < n) \wedge (i = start + 1) \wedge (start < n))$
12. $\quad \vee$
13. $\quad ((i >= n) \wedge (i = start + 1) \wedge (start < n))$
14. $\quad \equiv$
15. $\quad ((i = start + 2) \wedge (start + 1 < n))$
16. $\quad \vee$
17. $\quad ((i >= n) \wedge (i = start + 1) \wedge (start < n))\ \}$
18. . . .
19. $\{\ ((i = start + (n - start - 1)) \wedge (start + (n - start - 1) - 1 < n))$
20. $\quad \vee$
21. $\quad ((i >= n) \wedge (i = start + (n - start - 2)) \wedge (start + (n - start - 2) - 1 < n))$
22. $\quad \equiv$
23. $\quad ((i = n - 1) \wedge (n - 2 < n))\ \}$
24. `if i < n -> i:= i + 1 fi`
25. $\{\ sp(i := i + 1, (i < n) \wedge (i = n - 1) \wedge (n - 2 < n))$
26. $\quad \vee$
27. $\quad ((i >= n) \wedge (i = n - 1) \wedge (n - 2 < n))$
28. $\quad \equiv$
29. $\quad (i = n)\ \}$

Figure 4. Annotated Source Code for Unrolled Loop

if the loop were unrolled at least $(n - start - 1)$ times. As such, by using loop unrolling and induction, the derived specification for the code sequence is

$$((n - 1 < n) \wedge (i = n)).$$

For this simple example, we find that the solution is non-trivial when applying the formal definition of $sp(\texttt{DO}, Q)$. As such, the specification process must rely on a user-guided strategy for constructing a specification. A strategy for obtaining a specification of a repetition statement is given in Figure 5.

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

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

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

$$\texttt{proc}\ p : (input\_type)^* \rightarrow (output\_type)^* \tag{7}$$

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

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

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, sequence, and iteration as applied to the statements of the procedure body.

1. The following criteria are the main characteristics to be identified during the specification of the repetition statement:

   - *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).

2. Begin by introducing the assertion "$Q \wedge BB$" as the precondition to the body of the loop.

3. Query the user for modifications to the assertion made in step 2. This guided interaction allows the user to provide generalizations about arbitrary iterations of the loop. In order to verify that the modifications made by a user are valid, *wlp* can be applied to the assertion.

4. Apply the strongest postcondition to the loop body $\mathtt{S}_i$ using the precondition given by step 3.

5. Using the specification obtained from step 4 as a guideline, query the user for a loop invariant. Although this step is non-trivial, techniques exist that aid in the construction of loop invariants [15, 11].

6. Using the relationship stated above ($P \wedge \neg BB \rightarrow R$), construct the specification of the loop by taking the negation of the loop guard, and the loop invariant.

Figure 5. Strategy for constructing a specification for an iteration statement

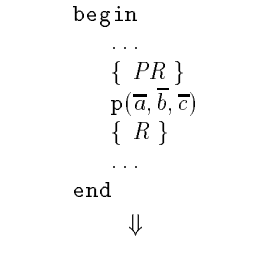
Gries defines a theorem for specifying the effects of a procedure call [11] using a total correctness model of execution. Given a procedure declaration of the above form, the following condition holds [11]

$$\{P\,RT : P^{\overline{x},\overline{y}}_{\overline{a},\overline{b}} \wedge (\forall \overline{u}, \overline{v} :: Q^{\overline{y},\overline{z}}_{\overline{u},\overline{v}} \Rightarrow R^{\overline{b},\overline{c}}_{\overline{u},\overline{v}})\}\ p(\overline{a},\overline{b},\overline{c})\ \{R\} \tag{8}$$

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 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 (8) 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 program block [11] found in Figure 6, where $\langle body \rangle$ comprises 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 enables 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 6 to appear as follows:

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

where $\overline{\alpha}$, $\overline{\beta}$, $\overline{\gamma}$, $\overline{\zeta}$, $\overline{\vartheta}$, and $\overline{\varphi}$ are the initial values of $\overline{\mathtt{x}}$, $\overline{\mathtt{y}}$ (before execution of the procedure body), $\overline{\mathtt{y}}$ (after execution of the procedure body), $\overline{\mathtt{z}}$, $\overline{\mathtt{b}}$, and $\overline{\mathtt{c}}$, 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




```
begin
   declare x̄, ȳ, z̄, ū, v̄;
   ...
   { PR }
   x̄,ȳ := ā,b̄;
   { P }
   ⟨body⟩
   { Q }
   ȳ,z̄ := ū,v̄;
   { QR }
   b̄,c̄ := ȳ,z̄;
   { R }
   ...
end
```

Figure 6. Removal of procedure call $p(\overline{a}, \overline{b}, \overline{c})$ abstraction

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 **value-result** parameters, the above sequence can be simplified using the semantics of *sp* for assignments to obtain the following annotated code sequence:

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

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 7, has four procedures, including three different implementations of "swap". AUTOSPEC [4, 9, 10] is a tool that we have developed to support the derivational approach to the reverse engineering of formal specifications from program code.

Figures 8, 9, and 10 depict the output of AUTOSPEC when applied to the program code given in Figure 7 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. 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 addition to the notations for variables, we use the notation '`|`' to denote a logical-or, '`&`' to denote a logical-and, and the symbols '`(* *)`' to delimit comments (i.e., specifications).

In Figure 8, 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 9 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 '`1`' 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 *)
```

```
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;
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 7. Example Pascal program

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`.

```
      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 *)
```

Figure 8. Output created by applying AUTOSPEC to example

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 10 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 10. 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

```
        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 *)

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

Figure 9. Output created by applying AUTOSPEC to example (cont.)

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 10. 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. Here, of interest are the final values of the variables that are local to the program `MaxMin` (i.e., `a`, `b`, and `c`). Thus, according to the rules for historical subscripts, the `a{0}3`, `b{0}3`,

```
     (* 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 10. Output created by applying AUTOSPEC to example (cont.)

and `c{0}1` are of interest. In addition, by propagating the preconditions for each statement, the logic that was used to obtain the values for the variables of interest can be analyzed.

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) [19]. 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 engineer COBOL involves 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 [12].

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* [7]). 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 and Future Investigations

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 specifications 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.

Since our technique to reverse engineering is based on the use of strongest postcondition for deriving formal specifications from program code, the application of the technique to other programming languages can be achieved by defining the for-

mal semantics of a programming language using strongest postcondition, and then applying those semantics to the programming constructs of a program. Our current investigations into the use of strongest postcondition for reverse engineering focus on three areas. First, we are extending our method to encompasses all major facets of imperative programming constructs, including iteration and recursion. To this end, we are in the process of defining the formal semantics of the ANSI C programming language using strongest postcondition and are applying our techniques to a NASA mission control application for unmanned spacecraft. Second, methods for constructing higher level abstractions from lower level abstractions are being investigated. Finally, a rigorous technique for re-engineering specifications from the imperative programming paradigm to the object-oriented programming paradigm is being developed [9]. Directly related to this work is the potential for applying the results to facilitate software reuse, where automated reasoning is applied to the specifications of existing components to determine reusability [14].

Acknowledgments

The authors greatly appreciate the comments and suggestions from the anonymous referees. Also, the authors wish to thank Linda Wills for her efforts in organizing this special issue. Finally, the authors would like to thank the participants of the IEEE 1995 Working Conference on Reverse Engineering for the feedback and comments on an earlier version of this paper.

Appendix A
Motivations for Notation and Removal of Quantification

Section 3.1 states a 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. This Appendix discusses this conjecture. Recall that

$$sp(\texttt{x:= e}, Q) = (\exists v :: Q_v^x \wedge x = e_v^x). \tag{A.1}$$

There are two goals that must be satisfied in order to use the definition of strongest postcondition for assignment. They are:

1. Elimination of the existential quantifier

2. Development and use of a traceable notation.

Eliminating the Quantifier. First, we address the elimination of the existential quantifier. Consider the RHS of definition A.1. Let y be a variable such that

$$(Q_y^x \wedge x = e_y^x) \Rightarrow (\exists v :: Q_v^x \wedge x = e_v^x). \tag{A.2}$$

Define $sp_\rho(\texttt{x:= e}, Q)$ (pronounced "s-p-rho") as the strongest postcondition for assignment with the quantifier removed. That is,

$$sp_\rho(\texttt{x:= e}, Q) = (Q^x_y \wedge x = e^x_y) \quad \text{for some } y. \tag{A.3}$$

Given the definition of sp_ρ, it follows that

$$sp_\rho(\texttt{x:= e}, Q) \Rightarrow sp(\texttt{x:= e}, Q). \tag{A.4}$$

As such, the specification of the assignment statement can be made more simple if y from equation (A.3) can either be identified explicitly or named implicitly. The choice of y must be made carefully. For instance, consider the following. Let $Q := P \wedge (x = z)$ such that P contains no free occurrences of x. Choosing an arbitrary α for y in (A.3) leads to the following derivation:

$$\begin{array}{ll} & sp_\rho(\texttt{x:= e}, Q) = Q^x_\alpha \wedge (x = e^x_\alpha) \\ \equiv & \langle Q := P \wedge (x = z) \rangle \\ & (P \wedge (x = z))^x_\alpha \wedge (x = e^x_\alpha) \\ \equiv & \langle \text{textual substitution} \rangle \\ & (P^x_\alpha \wedge (x = z)^x_\alpha \wedge (x = e^x_\alpha) \\ \equiv & \langle P \text{ has no free occurrences of x, Textual substitution} \rangle \\ & P \wedge (\alpha = z) \wedge (x = e^x_\alpha) \\ \equiv & \langle \alpha = z \rangle \\ & P \wedge (\alpha = z) \wedge (x = e^{x\alpha}_{\alpha z}) \\ \equiv & \langle \text{textual substitution} \rangle \\ & P \wedge (\alpha = z) \wedge (x = e^x_z). \end{array}$$

At first glance, this choice of y would seem to satisfy the first goal, namely removal of the quantification. However, this is not the case. Suppose P were replaced with $P' \wedge (\alpha \neq z)$. The derivation would lead to

$$sp_\rho(\texttt{x:= e}, Q) \equiv P' \wedge (\alpha \neq z) \wedge (\alpha = z) \wedge (x = e^x_z).$$

This is unacceptable because it leads to a contradiction, meaning that the specification of a program describes impossible behaviour. Ideally, it is desired that the specification of the assignment statement satisfy two requirements. It must:

1. Describe the behaviour of the assignment of the variable x, and

2. Adjust the precondition Q so that the free occurrences of x are replaced with the value of $\texttt{x}$ before the assignment is encountered.

It can be proven that through successive assignments to a variable x that the specification sp_ρ will have only one conjunct of the form $(x = \beta)$, where β is an expression. Informally, we note that each successive application of sp_ρ uses a textual substitution that eliminates free references to x in the precondition and introduces a conjunct of the form $(x = \beta)$.

The convention used by the approach described in this paper is to choose for y the expression β. If no β can be identified, use a place holder γ such that the precondition Q has no occurrence of γ. As an example, let y in equation (A.3) be z, and $Q := P \wedge (x = z)$. Then

$$sp_\rho(\texttt{x:= e}, Q) \equiv P \wedge (z = z) \wedge (x = e^x_z).$$

Notice that the last conjunct in each of the derivations is $(x = e^x_z)$ and that since P contains no free occurrences of x, P is an invariant.

Notation. Define $sp_{\rho\iota}$ (pronounced "s-p-rho-iota") as the strongest postcondition for assignment with the quantifier removed and indices. Formally, $sp_{\rho\iota}$ has the form

$$sp_{\rho\iota}(\texttt{x:= e}, Q) = (Q^x_y \wedge x_k = e^x_y) \quad \text{for some } y. \tag{A.5}$$

Again, an appropriate y must be chosen. Let $Q := P \wedge (x_i = y)$, where P has no occurrence of x other than i subscripted x's of form $(x_j = e_j), 0 \leq j < i$. Based on the previous discussion, choose y to be the RHS of the relation $(x_i = y)$. As such, the definition of $sp_{\rho\iota}$ can be modified to appear as

$$sp_{\rho\iota}(\texttt{x:= e}, Q) = ((P \wedge (x_i = y))^x_y \wedge x_{i+1} = e^x_y) \quad \text{for some } y. \tag{A.6}$$

Consider the following example where subscripts are used to show the effects of two consecutive assignments to the variable `x`. Let $Q := P \wedge (x_i = \alpha)$, and let the assignment statement be `x:= e`. Application of $sp_{\rho\iota}$ yields

$$\begin{array}{ll} & sp_{\rho\iota}(\texttt{x:= e}, Q) = (P \wedge (x_i = \alpha))^x_\alpha \wedge (x_{i+1} = e)^x_\alpha \\ \equiv & \langle \text{textual substitution} \rangle \\ & P^x_\alpha \wedge (x_i = \alpha)^x_\alpha \wedge (x_{i+1} = e)^x_\alpha \\ \equiv & \langle \text{textual substitution} \rangle \\ & P \wedge (x_i = \alpha) \wedge (x_{i+1} = e^x_\alpha) \end{array}$$

A subsequent application of $sp_{\rho\iota}$ on the statement `x:= f` subject to $Q' := Q \wedge (x_{i+1} = e^x_\alpha)$ has the following derivation:

$$\begin{array}{ll} & sp_{\rho\iota}(\texttt{x:= f}, Q') = (P \wedge (x_i = \alpha) \wedge (x_{i+1} = e^x_\alpha))^x_{e^x_\alpha} \wedge x_{i+2} = f^x_{e^x_\alpha} \\ \equiv & \langle \text{textual substitution} \rangle \\ & P^x_{e^x_\alpha} \wedge (x_i = \alpha)^x_{e^x_\alpha} \wedge (x_{i+1} = e^x_\alpha)^x_{e^x_\alpha} \wedge x_{i+2} = f^x_{e^x_\alpha} \\ \equiv & \langle P \text{ has no free } \texttt{x}, \text{ textual substitution} \rangle \\ & P \wedge (x_i = \alpha) \wedge (x_{i+1} = e^x_\alpha) \wedge x_{i+2} = f^x_{e^x_\alpha} \\ \equiv & \langle \text{definition of Q} \rangle \\ & Q \wedge (x_{i+1} = e^x_\alpha) \wedge x_{i+2} = f^x_{e^x_\alpha} \\ \equiv & \langle \text{definition of } Q' \rangle \\ & Q' \wedge x_{i+2} = f^x_{e^x_\alpha} \end{array}$$

Therefore, it is observed that by using historical subscripts, the construction of the specification of the assignment statements involves the propagation of the precondition Q as an invariant conjuncted with the specification of the effects of setting

a variable to a dependent value. This convention makes the evaluation of a specification annotation traceable by avoiding the elimination of descriptions of variables and their values at certain steps in the program. This is especially helpful in the case where choice statements (alternation and iteration) create alternative values for specific variable instances.

References

1. Eric Byrne. A Conceptual Foundation for Software Re-engineering. In *Proceedings for the Conference on Software Maintenance*, pages 226–235. IEEE, 1992.
2. Eric J. Byrne and David A. Gustafson. A Software Re-engineering Process Model. In *COMPSAC*. ACM, 1992.
3. Betty H. C. Cheng. Applying formal methods in automated software development. *Journal of Computer and Software Engineering*, 2(2):137–164, 1994.
4. Betty H.C. Cheng and Gerald C. Gannod. Abstraction of Formal Specifications from Program Code. In *Proceedings for the IEEE 3rd International Conference on Tools for Artificial Intelligence*, pages 125–128. IEEE, 1991.
5. Elliot J. Chikofsky and James H. Cross. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, January 1990.
6. Edsgar W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
7. Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.
8. Victoria Slid Flor. Ruling's Dicta Causes Uproar. *The National Law Journal*, July 1991.
9. Gerald C. Gannod and Betty H.C. Cheng. A Two Phase Approach to Reverse Engineering Using Formal Methods. *Lecture Notes in Computer Science: Formal Methods in Programming and Their Applications*, 735:335–348, July 1993.
10. Gerald C. Gannod and Betty H.C. Cheng. Facilitating the Maintenance of Safety-Critical Systems Using Formal Methods. *The International Journal of Software Engineering and Knowledge Engineering*, 4(2):183–204, 1994.
11. David Gries. *The Science of Programming*. Springer-Verlag, 1981.
12. H.P. Haughton and K. Lano. Objects Revisited. In *Proceedings for the Conference on Software Maintenance*, pages 152–161. IEEE, 1991.
13. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.
14. Jun-jang Jeng and Betty H. C. Cheng. Using Automated Reasoning to Determine Software Reuse. *International Journal of Software Engineering and Knowledge Engineering*, 2(4):523–546, December 1992.
15. Shmuel Katz and Zohar Manna. Logical Analysis of Programs. *Communications of the ACM*, 19(4):188–206, April 1976.
16. K. Lano and P.T. Breuer. From Programs to Z Specifications. In John E. Nicholls, editor, *Z User Workshop*, pages 46–70. Springer-Verlag, 1989.
17. Nancy G. Leveson and Clark S. Turner. An Investigation of the Therac-25 Accidents. *IEEE Computer*, pages 18–41, July 1993.
18. Wilma M. Osborne and Elliot J. Chikofsky. Fitting pieces to the maintenance puzzle. *IEEE Software*, 7(1):11–12, January 1990.
19. M. Ward, F.W. Calliss, and M. Munro. The Maintainer's Assistant. In *Proceedings for the Conference on Software Maintenance*. IEEE, 1989.
20. Jeannette M. Wing. A Specifier's Introduction to Formal Methods. *IEEE Computer*, 23(9):8–24, September 1990.
21. E. Yourdon and L Constantine. *Structured Analysis and Design: Fundamentals Discipline of Computer Programs and System Design*. Yourdon Press, 1978.

Received Date
Accepted Date
Final Manuscript Date