

## Facilitating the Maintenance of Safety-Critical Systems\*

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

Department of Computer Science  
Michigan State University  
A714 Wells Hall  
East Lansing, Michigan 48824  
tel: (517) 355-8344; fax: (517) 336-1061  
{gannod,chengb}@cps.msu.edu

---

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

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

## **Abstract**

As software is increasingly used to control safety-critical systems, correctness becomes paramount. Formal methods in software development provide many benefits in the forward engineering aspect of software development. Reverse Engineering is the process of constructing a high level representation of a system from existing lower level instantiations of that system. Reverse engineering of program code into formal specifications facilitates the utilization of the benefits of formal methods in projects where formal methods may not have previously been used, thus facilitating the maintenance of safety-critical systems.

**Keywords:** formal methods, formal specifications, reverse engineering, maintenance, safety-critical systems

# 1 Introduction

As software is increasingly used to control safety-critical systems, correctness becomes paramount. 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 caused by software increases.

Formal methods in software development provide many benefits in the forward engineering aspect of software development [3, 4, 5, 6, 7]. One of the advantages of using formal methods in software development is that the formal notations are precise, verifiable, and facilitate automated processing [8]. 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. Considering the high price of re-implementation and, even worse, the failure of software, reverse engineering of program code into formal specifications provides a good alternative approach to traditional methods for maintaining safety-critical systems.

This paper presents an approach to reverse engineering that focuses on representing of block structured programming language constructs in a form that facilitates the translation of programming statements into formal specifications. One of the difficulties in automating the abstraction of a formal specification from program code is that the specification can often be too tightly bound to the implementation. Ultimately, this coupling necessitates user interaction in order to correctly obtain a high level specification that is free of implementation bias. By taking full advantage of the logical properties of programming constructs, a precise determination of a program's purpose can be represented at a higher level of abstraction, as compared to program code. The corresponding formal specification, along with information provided by a domain expert (someone who is

knowledgeable about the specific domain, implementation details, and functionality requirements), facilitates determining program correctness using automated reasoning techniques. Furthermore, any implementation changes due to efficiency considerations, platform differences, or new requirements can be formally verified to determine whether the critical properties of the original system are preserved.

The remainder of this paper is organized as follows. Section 2 discusses basic properties associated with block structured languages. Section 3 describes the methods used to represent the major features of block structured languages, including a discussion of how those representations are translated into formal specifications. A few examples that emphasize the major facets of the abstraction process are described in Section 4. Related work in the area of reverse engineering is presented in Section 5. Conclusions as well as future work is outlined in Section 6.

## **2 Properties of Block Structured Languages**

The fundamental concepts of block structured languages originated with the development of ALGOL 60 [9] and have since been incorporated into the design of many programming languages, such as Ada [10] and Pascal [11]. Programs written using block structured languages are organized into nested blocks, where each block introduces a new local referencing environment. Because of their widespread and longstanding use, block structured languages have become a common object of study by maintenance engineers [12]. In order to analyze and maintain programs written using block structured languages, an understanding of the rules that govern block structured languages is necessary. The remainder of this section describes the fundamental concepts needed to understand block structured languages, where Pascal is used as a model for imperative (procedural) languages.

### **2.1 Static Scope Rules**

Static scope rules provide the definition of the context of a declaration within a program. In order to determine the intended behavior of a subject system, it is important that the rules that define

the scope of variables within a system are understood. When abstracting a formal specification from program code, the scope of a variable and its potential values play a major role in expressing the effects of statements. The static scope rules for block structured languages are as follows [9]:

1. The declarations at the beginning of any block define the local identifiers for a block.
2. Any identifiers referenced within a block for which no local declaration exists refer to the immediate parental block for a declaration. If no declaration is located in the parent block then the next ancestor is referenced. This identification process continues until the declaration is found (success) or no declaration is found (i.e., the top-most environment is reached and no declaration is present).
3. Declarations in nested child blocks are completely hidden from parent blocks and cannot be referenced by parents or ancestors.
4. Named subblocks in the form of subprograms are members of the parent's local referencing environment.

## 2.2 Statements

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

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

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

## 2.3 Subprograms

At issue in the use of subprograms is parameter association. Knowing the methods for binding formal and actual parameters as well as determining the parameter transmission schemes (i.e.,

value, value-result) of a language are necessary prerequisite tasks for the correct abstraction of formal specifications from program code. The rules for parameter association in Pascal are as follows [13]:

1. The number of formal and actual parameters for a given function or subprogram must be identical.
2. The types of the parameters must be the same. Actual parameters of type `integer` can be coerced to type `real`.
3. The actual parameters associated with `var` formal parameters must be variables. They cannot be constants or expressions.

### 3 Representation and Construction

Reverse engineering of program code into formal specifications facilitates the utilization of the benefits of formal methods in projects where formal methods may not have previously been used. Translation has been shown to provide an effective method for reverse engineering programs into formal specifications [14]. In order to perform a translation of program code into abstract specifications, a representation of the constructs that define a programming language must exist. These representations, in the form of rules and models, can then be used to translate the programs into specifications. In earlier investigations, an approach to reverse engineering of imperative programs was developed [15, 16]. The approach, based on the semantics of programming constructs as defined by the weakest precondition [17], provides the foundation for the methods presented in this paper for constructing formal specifications from program code in the form of preconditions and postconditions.

A *precondition* describes the initial state of a program, and a *postcondition* describes the final state. For a given postcondition  $R$  and a statement  $S$ , the *weakest precondition*  $wp(S, R)$  describes the set of all states in which the statement  $S$  can begin execution and terminate with  $R$  true [18]. A weakest precondition having the value *false* represents the empty set of states; *true* represents the set of all states. In terms of Hoare logic [19], the *wp* relationship is expressed as

$$\{ wp(S, R) \} S \{ R \},$$

where assertions are enclosed in braces. The  $wp$  is called a predicate transformer because it takes predicate  $R$  and, using the properties listed in Table 1, produces a new predicate. The general

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

Table 1: Properties of the  $wp$  predicate transformer

approach to constructing formal specifications from program code uses the definition of  $wp$  in the following manner. Given that some condition  $Q$  is known, then for some statement  $S$ , we derive postcondition  $R$ . Upon completion of the construction process, the rules of  $wp$  can then be used in the verification step to determine whether  $wp(S, R) = Q$ .

The remainder of this section presents the rules and representations that have been developed to encode the Pascal language and describes how these representations are used to construct formal specifications. For ease in understanding the representations, two complementary description mechanisms are used. First, a diagramming technique, known as the *Object-Modeling Technique* (OMT) [20], is used to represent the object-oriented relationships of programming constructs.<sup>1</sup> Second, the formal specification language Larch Shared Language (LSL) [21] is used to formally specify the abstract data types that support the automated construction of formal specifications from program code.

### 3.1 Variables, Symbols, and Types

Section 2 discussed properties (including static scope rules) relevant to the reverse engineering of program code written using imperative block structured languages. Static scope rules define the methods for determining the context of an identifier within a program. As is common in

---

<sup>1</sup>Note that the OMT approach includes the use of *object models*, *data flow diagrams*, and *statecharts*. In this discussion, OMT refers exclusively to the use of object models.

compiler construction [22], a hierarchical approach has been developed for both determining scope and recording histories of identifiers within a program. This approach is implemented in the form of an abstract data type (ADT) called **SymbolTable**.

A **SymbolTable** is an ADT containing a set of **SymbolTableElements** (referred to as the *symbols* set) and a link to a parent referencing environment. Each **SymbolTableElement** object in the *symbols* set represents an identifier, a table of the values of the instances for that identifier (*history*), and an index used to reference the last instance of the identifier in the *history* table. The **SymbolTableElement** objects contained in the *symbols* set are uniquely identified within the set by the name given to each identifier. The notation convention for referring to the name, history table, and index of the last instance added to the history table for a given identifier  $i$  will be  $i.name$ ,  $i.history$ , and  $i.index$ , respectively. The formal specification of the **SymbolTableElement** ADT, written in the Larch Shared Language, is given in Figure 1 and is used to define a method for determining the effects of certain programming constructs. In the specification, *SymTabElement* is the name of the **SymbolTableElement** ADT, the **includes** keyword indicates that the definition for the *Integer* type is used, the **introduces** keyword delimits the signature of the operators of the ADT, and the **asserts** keyword introduces the semantics of the operators, including the types of the arguments for the operators. The formal definitions for the operations of **SymbolTableElement** are used in the next section in the discussion of the abstraction process.

Figure 2 contains a graphical depiction of the **SymbolTable** ADT using the OMT notation that shows **SymbolTable** as an *aggregate of one or more SymbolTableElements* and *zero or one SymbolTables*, where aggregation is symbolically represented by a diamond, the “one or more” relation is represented by the filled circle, and the “zero or one” relation is represented by the hollow circle.

### 3.2 Statements, Rules, and Formal Specifications

Section 2.2 defined the *nesting property* of programming statements. For abstraction purposes, it becomes quite useful to recognize that nested statements are contained, to some extent, within a

---

```

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

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

```

Figure 1: Formal Specification of **SymbolTableElement** using LSL

---

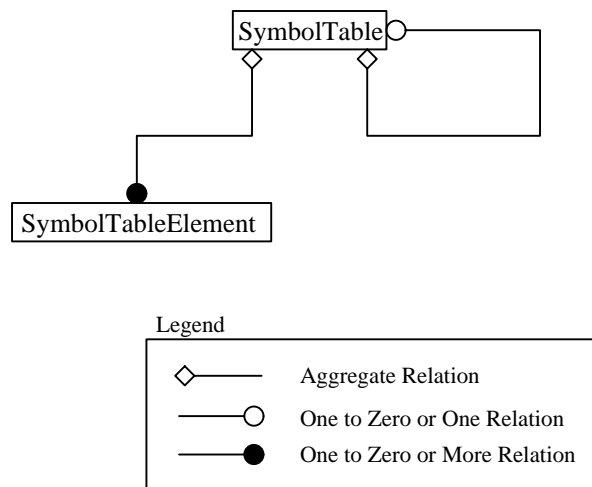


Figure 2: OMT Object Model of **SymbolTable**

---

single statement. For instance, consider the sequence of code given in Figure 3. This sequence has eleven separate statements including the **begin-end** sequences. At the highest level of abstraction this sequence has one statement, the outer **begin-end** statement. The next level of granularity contains three statements, the assignment statement at line 2, the **if** statement beginning at line 3, and the assignment statement at line 14. The **if** statement beginning at line 4 contains two statements in the form of an assignment statement and a **while** statement.

---

```

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

Figure 3: Example Sequence of Pascal Code

---

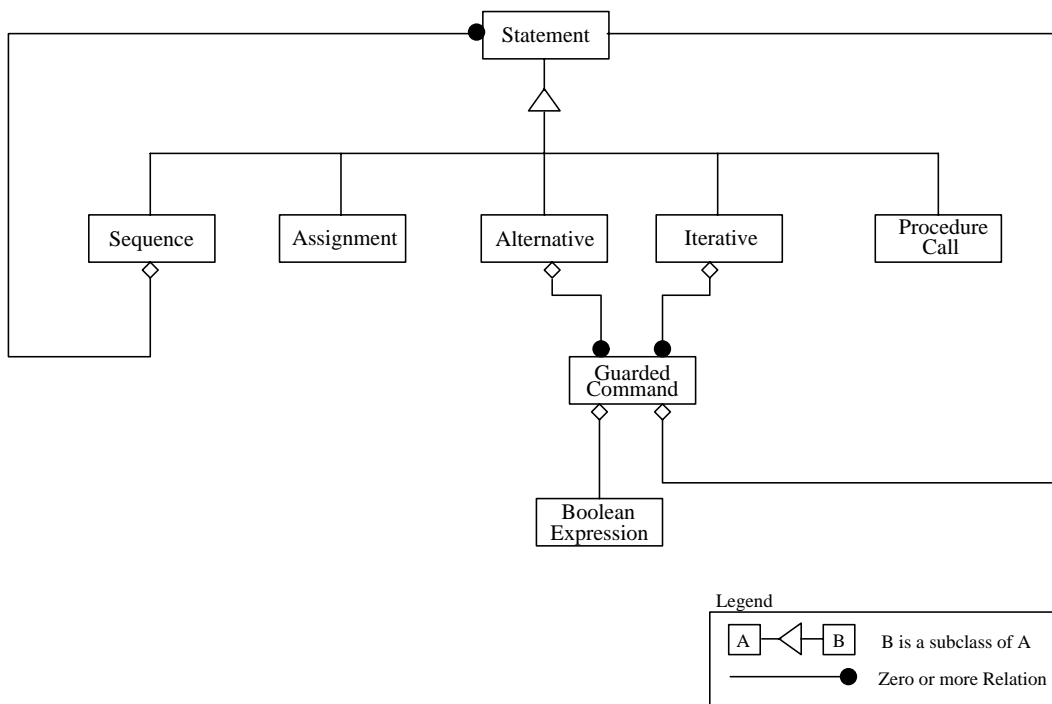


Figure 4: Graphical Representation of Statements

Figure 4 contains the object model of the **Statement** ADT, where the triangle symbol denotes inheritance. As such, the diagram explicitly states that **Sequence**, **Assignment**, **Alternation**, **Iteration**, **GuardedCommand** (a statement that is only executed when a given boolean expression called the *guard* is true), and **ProcedureCall** are **Statements**. Furthermore, the diagram depicts the notion of *nesting* through the use of the aggregation symbol. Notice that **Sequences** can contain zero or more **Statements**, and **Alternation** and **Iteration** statements can contain one or more **GuardedCommands**.

The rules for constructing formal specifications rely heavily on the assumption that nested programming statements can be abstracted into single statements with autonomous contexts that depend only on sequence and subprogram calls. For example, the **if** statement of line 3 in Figure 3 can be considered to be a function called **function-if**, whose arguments are the identifiers accessible via the current referencing environment. The output of the **function-if** is a conditional expression defined in terms of the identifiers referenced by **function-if**.

The *functional* approach to programming constructs has many benefits, including the natural correspondence to annotating program code with formal preconditions and postconditions. In order to support the functional view of programming constructs, the notion of a referencing environment local to a programming statement is introduced. It is important to note that in order to abstract away the details underlying an implementation, it becomes necessary to treat programming statements as “mini-programs” with the assumption that programming statements are single entry, single exit. Currently, this assumption effectively excludes **goto** statements but facilitates the hierarchical management of the abstraction process using **SymbolTable** objects.

### 3.2.1 Assignment

Given an assignment statement of the form  $\mathbf{x} := \mathbf{e};$  and a precondition  $U$ , where  $U$  is a logical expression, the following annotated code is constructed

```

{(x0 = X) ∧ U}           /* precondition */
x := e;
{(x0 = X) ∧ (x1 = e) ∧ U} /* postcondition */

```

where  $x_i = e$  is the specification of the  $i^{th}$  instance of the variable  $x$ , and the instances form the history of  $x$  in the corresponding **SymbolTableElement**. Expression  $U$ , propagated from the precondition to the postcondition, specifies the statements preceding the current statement. In propagating  $U$ , context is provided to the specification of the statement. For instance, the propagation of  $U$  in the previous example provides context to the specification of the assignment of  $x$  to the expression  $e$ . The remainder of this paper uses expression  $U$  in a similar way. The *wp* of an assignment statement is expressed as  $wp(\mathbf{x}:=\mathbf{expr}, R) = R_{e\mathbf{expr}}^x$ , which represents the postcondition  $R$  with every occurrence of  $x$  replaced by the expression  $\mathbf{expr}$ . This type of replacement is termed a textual substitution of  $x$  by  $\mathbf{expr}$  in expression  $R$  [18]. If  $x$  corresponds to a vector  $\bar{y}$  of variables and  $\mathbf{expr}$  represents a vector  $\bar{E}$  of expressions, then the *wp* of the assignment is of the form  $R_{\bar{E}}^{\bar{y}}$ , where each  $y_i$  is replaced by  $E_i$ , respectively, in expression  $R$ . In the above example, the *wp* of the assignment is  $U$ , which is satisfied by the original precondition.

One of the main purposes of the **SymbolTable** ADT is to support the construction of specifications. The history capabilities of a **SymbolTable** directly supports the construction of specifications by providing information about the values of various instances of a given identifier and by providing a means for storing the effects of an assignment statement. Expressions are resolved using textual substitution of the value of the last instance of an identifier contained in an expression and found in a **SymbolTable** object. For example, an expression typically found in a program might be as follows

$$q + r - s + t. \tag{1}$$

An expression can be processed such that the identifiers in the expression are replaced with the representation of the last instance of each identifier. Expression (1), for example, can be translated into an internal representation such that each identifier in (1) is replaced with the corresponding internal representation of the last instance for the identifier. If it is assumed that the last instance

for  $q$ ,  $r$ ,  $s$ , and  $t$  are  $q_1$ ,  $r_3$ ,  $s_0$ , and  $t_1$ , respectively, then Expression (1) would be translated into the following

$$q_1 + r_3 - s_0 + t_1, \tag{2}$$

where the subscripts represent the  $i^{th}$  instance of an identifier. For example, in Expression (2),  $r_3$  represents the third instance of identifier  $r$ . Once the initial substitution of the last instance for each identifier is completed, the representation of an instance of an identifier is replaced with the actual value of the instance. For illustration purposes, assume that the values of instances  $q_1$ ,  $r_3$ ,  $s_0$ , and  $t_1$  are 5, 9,  $s_0$ , and  $x_1$ , respectively. Then Expression (2) would be translated into the following

$$5 + 9 - s_0 + x_1. \tag{3}$$

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

### 3.2.2 Alternation

An alternation statement takes the form

```

if B then
    S1;
else
    S2;

```

where  $B$  is an expression and  $S1$  and  $S2$  are statements. An equivalent representation using the Dijkstra language [23] can be expressed as

```

if
    B → S1;
    ¬B → S2;
fi;

```

where  $B \rightarrow S_1$  and  $\neg B \rightarrow S_2$  are guarded commands such that  $S_1$  is only executed if logical expression (guard)  $B$  is true and  $S_2$  is executed if guard  $\neg B$  is true. Given a precondition  $U$ , a postcondition of an alternation statement of the form

$$((B_1 \wedge \text{post}(S_1)) \vee \dots \vee (B_n \wedge \text{post}(S_n))) \wedge U,$$

can be constructed such that the *wp* is satisfied. The *wp* of the alternation statement is: at least one guard  $B_i$  must be *true* and every guard must logically imply the *wp* of its corresponding statement list  $S_i$  with respect to the postcondition  $R$  [18]. Symbolically, the *wp* is expressed as

$$(\exists i :: B_i) \wedge (\forall i :: B_i \rightarrow \text{wp}(S_i, R)),$$

where ‘ $::$ ’ indicates that the range of the quantified variable  $i$  is not used in the current context.

Consider the code sequence in Figure 5(a). Using a bottom-up approach to specify this sequence

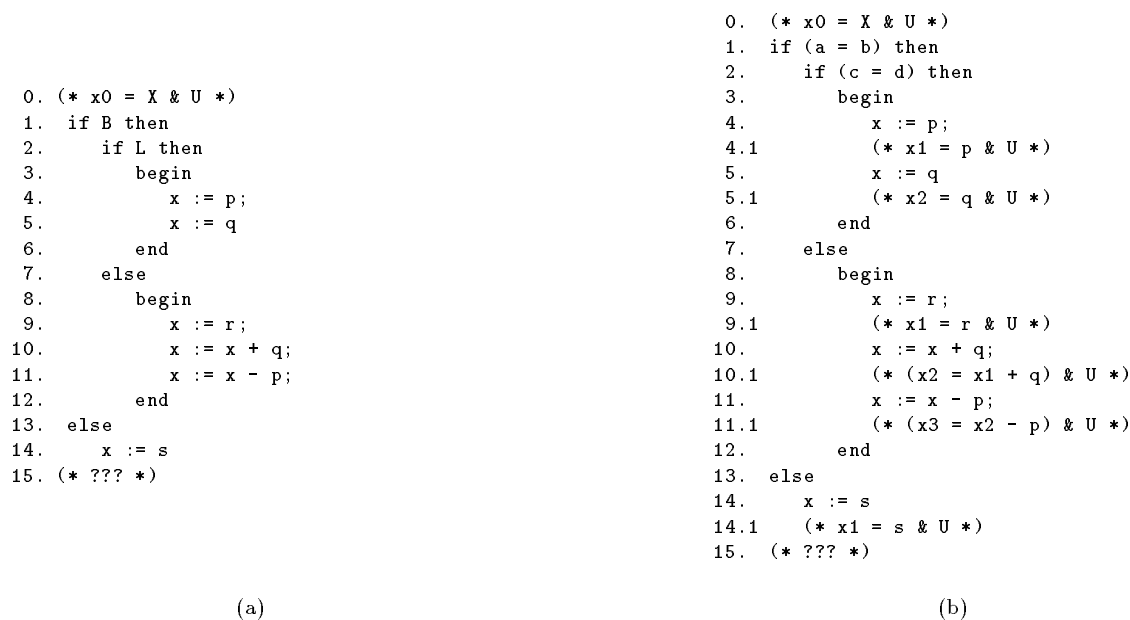


Figure 5: Code sequences partially annotated with specifications

---

of code requires that the deepest level of nesting be specified first, followed by the next to deepest level, and so on. This process yields the annotated code shown in Figure 5(b), where specifications are delimited with Pascal comment tokens. In our notation we use ‘&’ and ‘|’ to denote logical ‘and’ and ‘or’, respectively. Combining the specifications of lines 4.1, 5.1, 9.1, 10.1, 11.1, and 14.1 can often lead to an incoherent specification for line 15 due to the multiple instance subscripts

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

**Definition 1 (Last Instance)**

*Assuming that a begin-end sequence is single entry, single exit, then a last instance is the most recent value of any identifier accessible during the context of the sequence. In some cases, the last instance may be the same as the initial instance, while in other cases the last instance is different. Given that a **SymbolTable** object, called `symtab`, is used to manage an arbitrary level of nesting, we define the `last_instance` of an identifier `id` to be*

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

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

**Definition 2 (Dirty Set)**

*A dirty set is a construct that can be used to determine whether an assignment statement has been performed on an identifier in a nested sequence of statements. In addition to being useful for combining specifications of nested statements, a dirty set also aids in simplifying specifications. Formally, a dirty set is defined in the following manner*

$$dirty\_set(orig, target) = \{y \in orig \mid last\_instance(y, orig) \neq last\_instance(y, target)\},$$

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

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

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

where *main* represents the main **SymbolTable**, and *aux<sub>1</sub>* and *aux<sub>2</sub>* refer to the **SymbolTables** used to manage the guarded commands of the alternation statement. Upon determining the identifiers that fit this criterion, the main **SymbolTable** is updated in order to satisfy the following condition:

$$(\forall i : i \in I : (\exists j : j \in aux_1 : i.name = j.name \wedge last\_instance(j, aux_1) = last\_instance(i, orig)) \vee (\exists j : j \in aux_2 : i.name = j.name \wedge last\_instance(j, aux_2) = last\_instance(i, orig)))$$

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

Finally, with knowledge about the values of last instances of identifiers that would result from the execution of nested subblocks, a formal specification can be completed. Consider again the sequence of code given in Figure 5. The final version of the code with annotated specifications of the alternation statements using the notation `id{nest}instance` for identifiers, where `id` is the identifier in question, `nest` is the level of nesting, and `instance` is the instance number within the current context, is given in Figure 6.

### 3.2.3 Iteration

A Pascal iteration statement and an equivalent Dijkstra version take the forms, respectively.

---

```

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

```

Figure 6: Example with specifications annotated by the AUTOSPEC [16] system

---

<pre> while B   S; </pre>	<pre> do   B → S; od; </pre>
---------------------------	------------------------------

In more general terms, the iterative statement can 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. Gries defines guidelines for developing loops through the identification of loop invariants [18]. 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 7 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.

When no automated strategy can be applied to a loop or the constructed specification is incorrect, the domain expert is prompted for the proper specification of the statement. The following items are then identified in order to confirm that the specification of the loop is complete:

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

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

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

- *bound function (t)*: an integer expression representing the bound on the number of iterations. If at least one of the guards is true and the invariant is true, then the number of iterations is bounded below by  $t$  ( $P \wedge BB \rightarrow (t > 0)$ ).
- *statements that make progress towards termination (S<sub>i</sub>)*: these statements must decrease the bound function after each iteration. Each loop iteration is guaranteed to decrease the bound function. Formally, this condition is:

$$\{P \wedge B_i\}t_1 := t; S_i\{t < t_1\}, \text{ for } 1 \leq i \leq n,$$

where  $P \wedge B_i$  indicates that the invariant  $P$  and guard  $B_i$  are *true*, the assignment to  $t_1$  preserves the value of  $t$  before the iteration, and  $S_i$  represents the statements guarded by  $B_i$ .

---

1. The abstraction algorithm begins with the template for a quantified expression of the form
 
$$(Q i : range(i) : expression(i)),$$
 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 7: Steps for abstracting the effect of iteration statements

---

### 3.3 Procedure Calls, Rules, and Formal Specifications

This section describes the construction of formal specifications from code involving the use of procedural abstractions. Pascal **procedure** and **function** declarations can be represented using the following notation

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

where  $\bar{x}$ ,  $\bar{y}$ , and  $\bar{z}$  represent all the **value**, **value-result**, and **result** parameters for the procedure, respectively. The notation  $\langle \text{body} \rangle$  represents one or more statements making up the “procedure”, while  $\{P\}$  and  $\{Q\}$  are the precondition and postcondition, respectively. The syntactic *signature* of a procedure appears as

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

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 of the form

$$\begin{aligned} & \{ \text{pre: } U \} \\ & \text{proc } p : E_0 \rightarrow E_1 \\ & \{ \text{post: } \text{post}(\text{body}) \wedge U \} \end{aligned}$$

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,  $\text{post}(\text{body})$ , 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 [18]. Given a procedure declaration of the above form, the following condition holds

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

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  $PR$  must hold before the execution of procedure *p* in order to satisfy  $R$ . In addition,  $PR$  states that the precondition to procedure *p* must hold for the parameters passed to the procedure and that the postcondition for procedure *p* implies  $R$  for each **value-result** and **result** parameter. 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.

A procedure call  $p(\bar{a}, \bar{b}, \bar{c})$  can be represented by the following Pascal-like block [18]

```

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

```

where  $\langle body \rangle$  comprises the statements of the procedure declaration for  $p$ . By representing a procedure call in this manner, parameter binding can be achieved through a number of assignments and the postcondition  $R$  can be verified using the  $wp$  for assignment.

## 4 Example

The following example demonstrates the use of three major programming constructs described in this paper along with the application of the translation rules for abstracting formal specifications from code. The program, shown in Figures 8(a) and 8(b), has four procedures, including three different implementations of “swap”. AUTOSPEC [16] is a tool that we have developed to support the translational approach to the reverse engineering of formal specifications from program code. Figures 9 and 10 depict the output of AUTOSPEC when applied to the program code given in Figures 8(a) and 8(b), respectively, where the notation `id{procedure name}instance` is used to indicate an identifier `id` with scope defined by the referencing environment for `procedure name`.

Particular attention is directed to the specifications for the swap procedures given in Figure 9 named `swapa` and `swapb`. Although each implementation of the swap operation is different, the code in each procedure effectively produces the same results, a property appropriately captured by the respective specifications for `swapa` and `swapb`. In addition, Figure 10 shows the formal specification of the `funnyswap` procedure. The feature emphasized in this procedure is the different way that parameters are passed to the procedure, a property exhibited by the specification of the effects of the call to `funnyswap` in Figure 10. The procedure `FindMaxMin` provides another example of the

specification of alternative statements, with the specification of the procedure shown in Figure 9, and the effect of the call to the procedure given in Figure 10.

---

```

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.

```

(a)
(b)

Figure 8: Example Pascal program

---

## 5 Related Work

Lano et al. have investigated the translation of COBOL to UNIFORM to the formal notation  $Z$  [14, 24, 25, 26]. The approach is based on monad and category theory and relies on the notions of *phasing* and *slicing* for the identification of object-oriented constructs embedded in code. Ward

---

```

program MaxMin( input, output );

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

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

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

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

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

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

var
  temp : integer;

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

```

Figure 9: Output created by applying AUTOSPEC to example

---

```

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

var
  temp : integer;

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

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

```

Figure 10: Output created by applying AUTOSPEC to example (cont.)

---

et al. use a knowledge-based transformational approach to the reverse engineering of program code into formal specifications [27, 28, 29]. A knowledge-base manages the correctness preserving transformations of concrete, implementation constructs in a Wide Spectrum Language (WSL) to abstract specification constructs in the same WSL. Finally, Linger et al. describe a high level approach to reverse engineering using function abstraction and the logical properties of program flowcharts to derive descriptions of programs [30].

## 6 Conclusion

As software is increasingly used to control safety-critical systems, correctness becomes paramount. 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 approach described in this paper to reverse engineering of program code into formal specifications is a two step process involving the architectural (object) modeling of a programming language in order to capture the information pertinent to determining the semantics of a program and translations of the representations of particular instances of programs into formal specifications. Formal specifications constructed using the described approach facilitate the maintenance of programs and aid in preserving the safety-critical properties when code is modified.

Our previous investigations led to the development of AUTOSPEC, a tool that abstracts formal specifications from program code using a translational approach to determining the effects of basic programming constructs [16]. The prototype has since been modified to incorporate the modeling and translation approach described in this paper. Future investigations include the increased diversity in the methods used to determine the function of iteration statements including the use of *slicing* [30] as well as increasing the scope of the system to handle recursive procedures. We will also

be investigating the application of the remaining OMT modeling techniques to the representation of a programming language, that is, capturing the functional and behavior models.

## 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] J. M. Wing, "A Specifier's Introduction to Formal Methods," *IEEE Computer*, September 1990.
- [4] S. L. Gerhart, "Applications of formal methods: Developing virtuoso software," *IEEE Software*, vol. 7, pp. 7–10, September 1990.
- [5] N. G. Leveson, "Formal Methods in Software Engineering," *IEEE Transactions on Software Engineering*, vol. 16, pp. 929–930, September 1990.
- [6] R. A. Kemmerer, "Integrating Formal Methods into the Development Process," *IEEE Software*, vol. 7, pp. 37–50, September 1990.
- [7] A. Hall, "Seven myths of formal methods," *IEEE Software*, vol. 7, pp. 11–19, September 1990.
- [8] B. H. Cheng, "Synthesis of Procedural Abstractions from Formal Specifications," in *Proc. of COMPSAC'91*, pp. 149–154, September 1991.
- [9] T. Pratt, *Programming Languages: Design and Implementation*. Prentice-Hall, 1984.
- [10] J. Ichbiah, "Rationale for the Design of the Ada Programming Language," *SIGPLAN Notices*, vol. 14, 1979.
- [11] N. Wirth, "The programming language Pascal," *Acta Informatica*, vol. 1, pp. 35–63, 1971.
- [12] P. Benedusi, A. Cimitile, and U. DeCarlini, "A Reverse Engineering Methodology to Reconstruct Hierarchical Data Flow Diagrams for Software Maintenance," in *Proceedings for the Conference on Software Maintenance*, pp. 180–189, IEEE, 1989.
- [13] S. Leestma and L. Nyhoff, *Pascal Programming and Problem Solving*. Macmillan, second ed., 1987.
- [14] K. Lano and P. Breuer, "From Programs to Z Specifications," in *Z User Workshop* (J. E. Nicholls, ed.), pp. 46–70, Springer-Verlag, 1989.
- [15] G. C. Gannod and B. H. Cheng, "A Two Phase Approach to Reverse Engineering Using Formal Methods," *Lecture Notes in Computer Science: Formal Methods in Programming and Their Applications*, July 1993.

- [16] B. H. Cheng and G. C. Gannod, "Abstraction of Formal Specifications from Program Code," in *Proceedings for the IEEE 3rd International Conference on Tools for Artificial Intelligence*, pp. 125–128, IEEE, 1991.
- [17] E. W. Dijkstra, *A Discipline of Programming*. Prentice Hall, 1976.
- [18] D. Gries, *The Science of Programming*. Springer-Verlag, 1981.
- [19] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, pp. 576–580, October 1969.
- [20] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*. Englewood Cliffs, New Jersey: Prentice Hall, 1991.
- [21] J. Guttag and J. Horning, *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.
- [22] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.
- [23] E. W. Dijkstra, *A Discipline of Programming*. Englewood Cliffs, New Jersey: Prentice Hall, 1976.
- [24] K. Lano, "Formal Approaches to Reverse-Engineering," Tech. Rep. 2487-TN-PRG-1067, Oxford University, March 1991.
- [25] K. Lano, "Test Results for the Reverse-Engineering Tool Set," Tech. Rep. 2487-TN-PRG-1074, Oxford University, September 1991.
- [26] H. Haughton and K. Lano, "Objects Revisited," in *Proceedings for the Conference on Software Maintenance*, pp. 152–161, IEEE, 1991.
- [27] M. Ward, F. Calliss, and M. Munro, "The maintainer's assistant," in *Proceedings Conference on Software Maintenance*, (Miami, Florida), pp. 307–315, IEEE, October 1989.
- [28] T. Bull, "An Introduction to the WSL Program Transformer," in *Proceedings for the Conference on Software Maintenance*, pp. 242–250, IEEE, 1990.
- [29] F. Calliss, M. Khalil, M. Munro, and M. Ward, "A Knowledge-Based System for Software Maintenance," in *Proceedings for the Conference on Software Maintenance*, pp. 319–324, IEEE, 1988.
- [30] P. Hausler, M. Pleszkock, R. Linger, and A. Hevner, "Using Function Abstraction to Understand Program Behavior," *IEEE Software*, vol. 7, pp. 55–63, January 1990.