Int'l. Conf. on Software Maintenance, Nov. 1996, pp. 265-274; Appears as joint paper in WCRE '96

Using Informal and Formal Techniques for the Reverse Engineering of C Programs*

Gerald C. Gannod† and Betty H. C. Cheng‡
Department of Computer Science
Michigan State University
East Lansing, Michigan 48824
tel: (517) 355-8344; fax: (517) 432-1061
{gannod,chengb}@cps.msu.edu

Abstract

Reverse engineering of program code is the process of constructing a higher level abstraction of an implementation in order to facilitate the understanding of a system that may be in a "legacy" or "geriatric" state. Changing architectures and improvements in programming methods, including formal methods in software development and object-oriented programming, have prompted a need to reverse engineer and re-engineer program code. At the same time, there is a need to preserve the functionality of existing systems as well as reason about the correctness of changed code, each of which is facilitated by the existence of formal specifications. This paper describes an approach that incorporates the use of semi-formal analysis and formal program semantics to reverse engineer C *programs. The reverse engineering techniques are applied to a portion of a ground-based command system for unmanned flight systems.*

1. Introduction

Software maintenance has long been a problem faced by software professionals, where the average age of software is between 10 to 15 years old [11]. 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 [13]. One of the advantages of using formal methods in software development is that the formal notations are precise, verifiable, and facilitate automated processing [2]. *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 [5, 8, 12].

This paper presents an approach for combining the use of informal methods, such as structured analysis, with formal techniques in order to reverse engineer imperative programs written in the C programming language. The formal approach is based on the formal semantics of the *strongest postcondition* predicate transformer *sp* [4], and the partial correctness model of program semantics introduced by Hoare [9]. The objective of this approach is to take advantage of the benefits of graphical notations while providing a rigorous underlying formalism in order to make reverse engineering a more manageable task. The approach is applied to source code taken from an existing NASA application involving unmanned flight systems.

The remainder of this paper is organized as follows. Section 2 provides background material for software maintenance and formal methods. The semantics of the C programming language using the strongest postcondition predicate transformer is described in Section 3. The issues related to integrating informal and formal methods for reverse engineering are discussed in Section 4, and this approach is applied to a NASA ground-based system for controlling unmanned spacecraft in Section 5. Related work is described in Section 6. Finally, Section 7 draws conclusions and suggests future investigations.

*This work is supported in part by the National Science Foundation grants CCR-9407318, CCR-9209873, and CDA-9312389 and NASA Grant NGT-70376

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

‡Please address all correspondences to this author.

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

Figure 1 contains a graphical depiction of a process model for reverse and re-engineering [1]. The process model is captured by 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*. 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 configured into a new form at a different level of abstraction. Finally, *Refinement* of the new form into an implementation can be performed to create system *B*.

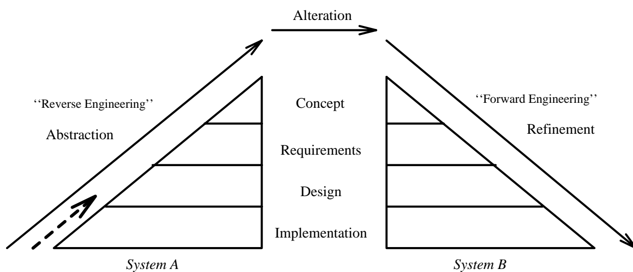


Figure 1. Re-engineering Process Model

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, current development teams must be able to understand the relationship between the source code and the specifications before taking advantage of the benefits offered by formal methods [3].

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 [14]) 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 [13]. 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 [13]. A benefit of formal methods is that their notations are well-defined and thus, are amenable to automated processing [2].

The notation $Q \{ S \} R$ [9] 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. 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 since the termination properties of these systems are known *a priori*.

The *strongest postcondition* $sp(S, Q)$ predicate transformer [4] is defined as the set of all states in which *there exists* a computation of S that begins with precondition 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. The *weakest precondition* predicate transformer $wp(S, R)$ is defined as the set of all states in which the statement S can begin execution and terminate with postcondition R true. Given a Hoare triple $Q \{ S \} R$, we note that wp is a "backward" rule, in that a derivation of a specification begins with postcondition 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 $wp(S, R)$, then 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 $sp(S, Q)$.

3. Semantics of C Programs

Our previous investigations [7] involved the use of the strongest postcondition predicate transformer as applied to the Dijkstra guarded command language [4]. This section defines the *sp* semantics of the C programming language [10]. Due to space constraints, only a subset of the programming language used in the application example is presented. A more complete description of the *sp* semantics of the C programming language may be found in [6].

3.1. Assignment

Let v be a variable or an assignable expression and e be an expression. An assignment in the C programming language has the form $\mathtt{v} \cong \mathtt{e}$, where $\cong$ is an assignment operator (i.e., =, +=, *=). There are two roles that an assignment statement can have. The first is the traditional assignment of a variable with the value of an expression. The second role is as a side-effect boolean expression.

In order to handle the dual role of an assignment statement, two functions are defined. First, in order to describe the semantics of the traditional use of assignment, an *evaluation* function $\mathcal{A}$: $S \rightarrow \mathcal{T}$ is defined, where S is the set of syntactically valid expressions, and $\mathcal{T}$ is the range of the result given by evaluating the expression e. If $s \in S$ is a non-assignment expression, then in general $\mathcal{A}(s) = s$. If, however, s is an assignment statement such as "x *= n", the function $\mathcal{A}$ would be evaluated as $\mathcal{A}(\mathtt{x\ *=\ n}) = \mathtt{x} \times \mathtt{n}$. Table 1 defines the semantics of the function $\mathcal{A}$ on a few sample assignment operators. A more general form of the function $\mathcal{A}$, can be defined as $\mathcal{A}(\mathtt{b}) = \mathtt{b}$, where b is an expression. The interpretation is that the evaluation $\mathcal{A}$ on any expression has the value of the expression. For example, consider "$\mathcal{A}(\mathtt{x\ +\ y\ +\ z})$". The expression "x + y + z" is a non-assignment expression, therefore $\mathcal{A}(\mathtt{x\ +\ y\ +\ z}) = \mathtt{x} + \mathtt{y} + \mathtt{z}$. Due to space constraints, we focus primarily on the assignment expressions. Using the definition of $\mathcal{A}$, we can define the strongest postcondition of an assignment in the following manner:

$$sp(\mathtt{x} \cong \mathtt{e}, Q) = (\exists v :: Q_v^x \wedge x = \mathcal{A}(\mathtt{x} \cong \mathtt{e}_v^x)), \qquad (1)$$

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. This specification states that after the execution of an assignment statement, there exists some value v such that the textual substitution of every free occurrence of x with v in Q keeps Q true, and x takes the value of the evaluation $\mathcal{A}$ on $\mathtt{x} \cong \mathtt{e}_v^x$. This means that after the execution of an assignment statement, the precondition Q must still be true with respect to the value that the variable x had before the assignment, and that the assignment must be valid.

The second function that is used to define the effects of an assignment statement is the *logical valuation* function $\mathcal{V} : S \rightarrow \mathcal{B}$, where S is the set of valid expressions, and $\mathcal{B}$ is the Boolean type. Note that S includes general expressions and assignment expressions. In this paper, we focus on assignment expressions. The purpose of $\mathcal{V}$ is best motivated by an example. Consider the sequence of code in Figure 2. Informally, the semantics of this code sequence is that if the guard is true, execute S1, otherwise execute S2. However, the guard is worth noting since the expression is not a logical one, but rather an assignment expression. The semantics in this case are dependent on the side-effect of executing the statement v = e. Using the function $\mathcal{A}$, function $\mathcal{V}$ is defined as:

$$\mathcal{V}(v \ \cong \ e) = \begin{cases} T & \text{if } \mathcal{A}(v \cong e) \neq 0 \\ F & \text{if } \mathcal{A}(v \cong e) = 0 \end{cases},$$

where T and F are Boolean constants *true* and *false*, respectively. In general, for some arbitrary expression b, $\mathcal{V}$ is defined as:

$$\mathcal{V}(b) = \begin{cases} T & \text{if } \mathcal{A}(b) \neq 0 \\ F & \text{if } \mathcal{A}(b) = 0\,. \end{cases}$$

Although the side-effects of an assignment statement have no effect on the assignment itself, the side-effects do impact other operations as was shown in the short example above. The use of $\mathcal{V}$ will be important for defining the semantics of alternation statements with side-effects.

Operation $v \cong e$	Evaluation $\mathcal{A}$
=	e
*=	$v \times e$
/=	$\frac{v}{e}$
+=	$v + e$
-=	$v - e$
%=	v mod e

Table 1. Evaluation of $\mathcal{A}$ on sample C assignment operators

```
if (v = e) {
    S1
} else {
    S2
}
```

Figure 2. Assignment statement as a guard

In this paper, discussion has been limited to a small subset of the available assignment operators. The semantics of the shift and bitwise assignment operators can also be defined using the functions $\mathcal{A}$ and $\mathcal{V}$. In addition, the semantics of other expressions can be defined using these functions [6].

3.2. Alternation

Alternation is a term that was originally introduced by Dijkstra to describe conditional statements in his guarded command language [4]. In this paper we use the same convention. The alternation statement for C programs can take two forms:

```
if B {              if B {
  S                   S1
}         and       } else
                      S2
```

We refer to these statements as C-IF1 and C-IF2, respectively.

When the guard of an alternation statement has no side-effects, the semantics of the alternation statement is as follows, respectively:

$$\begin{aligned} sp(\texttt{C-IF1}, Q) &\equiv sp(\texttt{S}, \texttt{B} \wedge Q) \vee sp(\texttt{skip}, \neg\texttt{B} \wedge Q) \\ &\equiv sp(\texttt{S}, \texttt{B} \wedge Q) \vee (\neg\texttt{B} \wedge Q) \end{aligned}$$

$$sp(\texttt{C-IF2}, Q) \equiv sp(\texttt{S}_1, \texttt{B} \wedge Q) \vee sp(\texttt{S}_2, \neg\texttt{B} \wedge Q),$$

where Q is the precondition for the alternation statement. The specification of $sp(\texttt{C-IF1}, Q)$ states that after execution of C-IF1 either $sp(\texttt{S}, \texttt{B} \wedge Q)$ is true (i.e., S was executed) or $(\neg\texttt{B} \wedge Q)$ is true (guard B was false). Similarly, the specification of $sp(\texttt{C-IF1}, Q)$ states that after execution of C-IF2 either $sp(\texttt{S}_1, \texttt{B} \wedge Q)$ is true (i.e., $\texttt{S}_1$ was executed) or $sp(\texttt{S}_2, \neg\texttt{B} \wedge Q)$ is true (guard B was false and statement $\texttt{S}_2$ was executed).

If the restriction of having alternation statements without side-effects in the guards is removed, then the semantics of the alternation statement has a different meaning. Informally, if there is a side-effect in the guard B, then the execution of an alternation is analogous to "executing" B, then executing the alternation using the evaluation of B. More formally, let B be a guard of an alternation statement (C-IF1 for instance) such that the evaluation of B causes a side-effect, and let $\mathcal{V}(B)$ represent the truth value of B. Execution of the alternation statement is equivalent to the execution of the following, respectively:

```
B;                  B;
if V(B) {           if V(B) {
  S                   S1
}                   } else
                      S2
```

We refer to the alternation statements (the if statement with the replacement of B by $\mathcal{V}(B)$) as $\texttt{C-IF1}_s$ and $\texttt{C-IF2}_s$, respectively. The semantics of $\texttt{C-IF}_s$ are as follows:

$$\begin{aligned} sp(\texttt{C-IF1}_s, Q) &\equiv sp(\texttt{C-IF1}, sp(\texttt{B}, Q)) \\ &\equiv sp(\texttt{S}, \mathcal{V}(\texttt{B}) \wedge sp(\texttt{B}, Q)) \vee \\ &\quad (\neg\mathcal{V}(\texttt{B}) \wedge sp(\texttt{B}, Q)) \end{aligned}$$

$$\begin{aligned} sp(\texttt{C-IF2}_s, Q) &\equiv sp(\texttt{C-IF2}, sp(\texttt{B}, Q)) \\ &\equiv sp(\texttt{S}_1, \mathcal{V}(\texttt{B}) \wedge sp(\texttt{B}, Q)) \vee \\ &\quad sp(\texttt{S}_2, \neg\mathcal{V}(\texttt{B}) \wedge sp(\texttt{B}, Q)). \end{aligned}$$

The specification of $sp(\texttt{C-IF1}_s, Q)$ states that after execution of $\texttt{C-IF1}_s$ either $sp(\texttt{S}, \texttt{B} \wedge Q)$ is true (i.e., S was executed) or the valuation of $(\neg\texttt{B} \wedge Q)$ is true (the valuation of the guard $\mathcal{V}(\texttt{B})$ was false). Similarly, the specification of $sp(\texttt{C-IF2}_s, Q)$ states that after execution of C-IF2 either $sp(\texttt{S}_1, \texttt{B} \wedge Q)$ is true (i.e., $\texttt{S}_1$ was executed) or $sp(\texttt{S}_2, \neg\texttt{B} \wedge Q)$ is true (the valuation of B was false and statement $\texttt{S}_2$ was executed).

3.3. Sequence

Sequences of statements in the C programming language have the form $\texttt{S}_1\texttt{;}\ \texttt{...;}\texttt{S}_n$. The appropriate semantics using *sp* is as follows:

$$sp(\texttt{S}_1\texttt{;}\texttt{S}_2, Q) \equiv sp(\texttt{S}_2, sp(\texttt{S}_1, Q)). \qquad (2)$$

Since the impact of side-effects are specified by the corresponding sp formalisms for assignment, alternation, and iteration, this characterization of the semantics of sequence is sufficient.

3.4. Iteration

This section describes the strongest postcondition semantics for the while iteration construct of the C programming language. For the do-while and for constructs, appropriate transformations using the while semantics are provided.

3.5 while

The while statement in the C programming language has the form `while (B) { S; }`, where B is a guard and S is a statement. When no side-effects are present, the while iteration construct has the following semantics:

$$sp(\texttt{while}, Q) = \neg\texttt{B} \wedge (\exists i : 0 \leq i : sp(\texttt{C-IF1}^i, Q)), \qquad (3)$$

where Q is the precondition for the iteration statement. Expression (3) states that if the execution of the while statement terminates then the guard B is false and the result of applying the rule $sp(\texttt{C-IF1}, Q)$ i times is true. This construction comes from the fact that an iteration statement can be thought of as a series of alternation statements, where the guard for the alternation is given by the guard of the iteration and the number of alternation statements that are included in the series is determined by the guard. Obviously, there is no decidable way to determine how many alternation statements to include in the series. Notationally, $sp(\texttt{C-IF1}^i, Q)$, where i is the number of iterations, means that *sp* is recursively applied to the result of $sp(\texttt{C-IF1}, Q)$. For instance, $sp(\texttt{C-IF1}^3, Q)$ has the following derivation:

$$\begin{aligned} sp(\texttt{C-IF1}^3, Q) &\equiv sp(\texttt{C-IF1}, sp(\texttt{C-IF1}^2, Q)) \\ &\equiv sp(\texttt{C-IF1}, sp(\texttt{C-IF1}, sp(\texttt{C-IF1}, Q))). \end{aligned}$$

In the case when the guard of the while statement has a side-effect, the semantics are similar to executing the following:

```
B;
while (V(B)) {
    S;
    B;
}
```

where $\mathcal{V}$ is the valuation function described previously. The corresponding sp semantics of the `while` statement with side-effects (denoted $sp(\texttt{while}_s, Q)$) is

$$\neg\mathcal{V}(\texttt{B}) \wedge (\exists i : 0 \leq i : sp(\texttt{C-IF1}^i, sp(\texttt{B}, Q))), \qquad (4)$$

where the body of the statement `C-IF1` consists of "`S`; `B`;" as given by the transformation of the `while` statement to account for the side-effect. Expression (4) states that if the execution of the `while` statement terminates then the valuation of the guard `B` is false and the result of applying the rule $sp(\texttt{C-IF1}, Q)$ i times is true.

3.6 `do while`

The `do while` statement in the C programming language has the form `do { S; } while (B)`, where `B` is a guard and `S` is a statement. The semantics of the `do while` statement are similar to the `while` statement, where the guarding condition appears after the loop body. Using the `while` construct, `do while` can be written as the following:

```
S;
B;
while (V(B)) {
    S;
    B;
}
```

The corresponding formal specification of the semantics of the `do while` statement is

$$sp(\texttt{while}_s, sp(S, Q)) = \qquad (5)$$
$$\neg\mathcal{V}(\texttt{B}) \wedge (\exists i : 0 \leq i : sp(\texttt{C-IF1}^i, sp(\texttt{B}, sp(S, Q)))),$$

where the body of the statement `C-IF1` consists of "`S`; `B`;", and the effects of executing "`S; B`" before entering the loop is given by the precondition argument of $sp(\texttt{C-IF1}^i, sp(\texttt{B}, sp(S, Q)))$. This specification states that after the execution of a *do while* statement, the valuation of `B` is false, and the body of the loop is executed *i* times.

3.7 `for`

Recall that the `for` construct in C has the form

```
for (expr1;expr2;expr3) {
    S;
}
```

The semantics of the `for` iteration statement is that the first expression (*expr1*) is executed (evaluated) once, the second expression (*expr2*) is evaluated before each iteration, and the third expression (*expr3*) is evaluated after each iteration. These semantics, using the `while` construct, are represented by the following:

```
expr1;
expr2;
while (V(expr2)) {
    S;
    expr3;
    expr2;
}
```

The resulting formal specification of the semantics of `for` using the *sp* for `while` is defined as

$$sp(\texttt{while}_s, sp(expr1, Q)) = \qquad (6)$$
$$\neg\mathcal{V}(expr2) \wedge (\exists i : 0 \leq i : sp(\texttt{C-IF1}^i, sp(expr2, Q))),$$

where the body of the statement `C-IF1` consists of "`S`; *expr3*; *expr2*;". This specification states that after the execution of the `for` loop, the logical valuation of *expr2* is false, and the loop body is executed *i* times where the initial precondition to the loop is given by $sp(expr2, Q)$.

3.8. Function Calls

Functions in the C programming language can serve two basic purposes. A function can be a *pure value function*, where the purpose is to compute some value based on the parameters. Alternatively, a function can be a *procedure*, where the purpose is to perform a number of encapsulated tasks. Our previous investigations [7] describe an approach for defining the semantics of functions that serve a procedural role. Due to space constraints, this discussion is not repeated here.

Table 2 contains a taxonomy of functions based on the properties of *variables*, *side-effects*, *values returned*, and *parameters*. The *variables* property describes the kinds of variables that are used by a function. The *side-effects* property is used to indicate whether the class of functions produces side-effects. The types of parameters and the number of values that are returned by a function are described by the *parameter* and *values returned* properties, respectively. *Pure Valued* functions are characterized by the use of local variables, the functions produce no side-effects, the parameters are value parameters, and the functions return a single value. Note that a procedural function can effectively serve the role of a pure valued function if it can be ensured that the functions produce no side-effects. This implies that the number of values must be singular. In this context, we assume that the modification of a value-result parameter or result parameter produces a side-effect.

	Function Class	
Property	Procedural	Pure Valued
variables	global, local	local
side-effects	yes	no
parameters	value, value-result, result	value
values returned	multiple	single

Table 2. A Taxonomy of Programming Language Functions

A function in the C programming language has a signature (or prototype) of the form $\mathcal{R}\ \ \mathtt{f}(\mathcal{D})$ where $\mathcal{R}$ is the return type, and $\mathcal{D}$ is the input type of function `f`. For example, a function `max` could have a signature "`int max(int,int);`". Given a variable "`x`" of type $\mathcal{R}$, a parameter "`a`" of type $\mathcal{D}$, and an assignment operator $\cong$, a call to the function f has the form "$\mathtt{x} \cong \mathtt{f(a)}$".

Let `f` be a pure valued function. The effect of calling the function is that a value is returned and assigned to the variable `x`. The corresponding *sp* semantics for the function call is

$$sp(\mathtt{x} \cong \mathtt{f(a)}, Q) = (\exists v :: Q_v^x \wedge x = \mathcal{A}(\mathtt{x} \cong \mathtt{f(a}_v^x\mathtt{)})). \quad (7)$$

This specification states that after the execution of an assignment statement using a function call, there exists some value *v* such that the textual substitution of every free occurrence of *x* with *v* in *Q* keeps *Q* true, and *x* takes the value of the evaluation $\mathcal{A}$ on $\mathtt{x} \cong \mathtt{f(a}_v^x\mathtt{)}$. Note that in the case where a pure valued function is called but not assigned that $sp(\mathtt{f(a)}, Q) = Q$.

4. Approach

Due to the mathematical nature of formal specification languages, formal methods have been perceived as time consuming and tedious. However, since the languages are well-defined, formal methods have been found to be amenable to automated processing. Semi-formal methods are techniques for specifying system requirements and design using hierarchical decomposition. Most semi-formal methods have the property that the notations are graphical, facilitating ease of use in their application. The drawback of semi-formal methods is that the notations are imprecise and ambiguous. This section describes an approach to reverse engineering that combines the use of semi-formal methods and formal methods in order to utilize the complementary advantages of the notations.

4.1. Structured Analysis

Although the recent trend in software development has been to build systems using object-oriented technology, a majority of existing systems have been developed using imperative programming languages, such as C, FORTRAN, and COBOL. The procedural structure of these languages makes them amenable to the techniques offered by the *Structured Analysis and Design Technique* (SADT) [14]. In SADT, the focal point is the procedure or function. The analysis stage centers around high-level descriptions of the functionality of the system. During the design phase, the refinement and decomposition of the high-level descriptions of functions yields more detailed descriptions of functions and procedures that incorporate implementation details. Finally, during the implementation phase, functions and procedures identified during design are decomposed into more specific functions.

When using SADT for reverse engineering activities, the structure of an implementation is abstracted into high-level graphical descriptions functions known as call graphs or structure charts. These graphs depict the calling hierarchy of functions within a system. Further analysis of source code involves analyzing the data that flows to and from various functions by constructing data flow diagrams. Our approach is to construct various graphical descriptions of a program, in most cases automatically, and then use those descriptions to guide the construction of formal specifications from the different parts identified by the graphical descriptions.

4.2. Applying formal techniques

The purpose of combining the use of formal methods and semi-formal methods is two-fold. First, it is desirable to take advantage of the benefits of the complementary techniques. Second, by using a semi-formal technique to guide the formal technique, organization of the formal specifications will be based on the structure of an implementation. As such, in the case where formal specifications are warranted, the specifications can be directly associated with a graphical entity, while those parts of a module that do not require rigorous descriptions can be left unspecified (formally), with the descriptions of these modules being left to the semi-formalisms.

There are three guidelines that are followed when formally specifying a module. That is, the process of formally specifying a module consists of three steps or phases: 1) Local Analysis, 2) Use Analysis, and 3) Global Analysis. During the *local analysis* phase, the calling hierarchy of a module is constructed and a skeletal formal specification is built, with the *sp* predicates left as parameterized transforms, that is, the transformations for *sp* are unevaluated. The objective at this stage is to gain a high-level understanding of the logical complexity of the given code. The second step, *use analysis*, is a recursive step where the three phases are applied to the functions and procedures *used* by the original module. This phase is characterized by the fact that the

semantics of the *used* functions and procedures are determined before they are used by the original module. However, in many cases, where the semantics are either well-defined or the semantics are not critical, an unevaluated *sp* predicate can be used. For example, given a statement **S** and a precondition *Q* where the semantics of *S* are well-defined, instead of evaluating the transformation, we use $sp(S, Q)$ to represent the logical expression describing the semantics. In the *global analysis* phase, the *use analysis* information is combined with the *local analysis* information to obtain a global description of the original module. The global description, an expanded form of the skeleton formal specification constructed during the first phase, elaborates upon the semantics of a module by integrating the specifications constructed during the *use* analysis into the skeleton. This activity corresponds to removing the encapsulation provided by a procedure or function call.

In order to support our approach, we have been developing a system called AUTOSPEC. Currently, AUTOSPEC supports the construction of graphical specifications from C programs and is being extended to support the construction of formal specifications using the three step approach described above.

5. An Example

In this section, we demonstrate the use of the integrated approach to modules from a mission control ground-based system at the NASA Jet Propulsion Laboratory. The purpose of the code is to translate user commands into spacecraft mnemonics.

5.1. Local Analysis

Figure 3 gives the code for the translate procedure. An initial semi-formal analysis of the translate code yields a calling graph as depicted in Figure 4, where the rectangles indicate functions, and the labels correspond to the function names given by the index to the right of the graph. From this initial analysis, we find that the `translate` function uses five functions including `initialize_interpreter`, `inform_user`, `end_cmdxlt`, `process_carg`, `process_binary_output`, and `process_mnemonic_input`. The `translate` function has four different modes: initialize, translate, control argument assignment, and error. For this analysis, we focus on the translate function in the translate mode (`XLT`). Thus, we are ignoring the initialization, control argument, and default modes in this analysis which correspond to the `INIT`, `CARG`, and `default` cases of the `switch` statement,[1] respectively. Therefore, we are left with specifying the `while` statement depicted in Figure 5, where

```
struct msg *translate (int op, char *args)
{
    extern int dontoutput;
    static struct project_parameters *pp;
    struct msg *mp = NULL;

    switch (op)
    {   /* initialize the interpreter */
        case INIT:
           pp = initialize_interpreter();
           break;
        /* interpret a message */
        case XLT:
           while (args[0] != '\0')
           {
              if (process_mnemonic_input(&args, pp))
              {
                 if (mp == NULL)
                    mp = process_binary_output(pp);
                 else
                 {
                    mp->next = process_binary_output(pp);
                    mp = mp->next;
                 }
              }
              else
                 dontoutput = 1;
           }
           break;
        /* set a value for a control argument */
        case CARG:
           process_carg(&args, pp);
           break;
        default:
           inform_user(
              "internal error: bad op in translate");
           end_cmdxlt(CMD_ERROR);
    }

    return(mp);
}
```

Figure 3. Translate Source Code

labels have been attached to the programming constructs for convenience in the following discussion. Informally, the `translate` function in the translate mode is responsible for building a list of spacecraft instructions corresponding to interpreted commands by calling a function called `process_binary_output`.

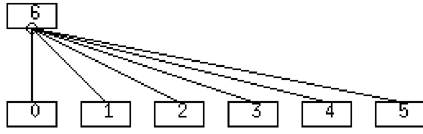
An analysis of the code in Figure 5 using the *sp* rule for the `while` statement yields the following specification:

$$\neg(\texttt{args[0] != '0'}) \wedge (\exists i : 0 \leq i : sp(\mathbf{S0}^i, Q)), \quad (8)$$

where the expression $(\texttt{args[0] != '0'})$ has no side-effects, and *Q* is the precondition to the statement **S0**. This specification states that after the `while` statement has been executed, the `args` array has a `'0'` as the first entry, and the statement **S0** has been executed some number of iterations. Unfortunately, the specification in (8) is not very informative outside of identifying that the program uses an iterative construct. As such, an expansion of $sp(\mathbf{S0}, Q)$ is warranted.

Using the labels shown in Figure 5, a specification of

[1] Although in the context of this paper we have not defined the semantics of the `switch` statement, our investigations have included the construct.



0. initialize_interpreter
1. process_binary_output
2. inform_user
3. process_mnemonic_input
4. end_cmdxlt
5. process_carg
6. translate

Figure 4. Translate

$sp(\mathbf{S0}, Q)$ is given by

$$sp(\mathbf{S0}, Q) \;=\; sp(\mathbf{S1}, \mathcal{V}(B) \wedge sp(B,Q)) \vee sp(\mathbf{S2}, \neg\mathcal{V}(B) \wedge sp(B,Q)) \quad (9)$$

where B = `process_mnemonic_input(&args,pp)`. This specification states that after executing the statement **S0**, it will be true that either **S1** was executed or **S2** was executed, where the semantics are determined by the preconditions $\mathcal{V}(B) \wedge sp(B,Q)$ and $\neg\mathcal{V}(B) \wedge sp(B,Q)$, respectively. So, in this case, either the `if` statement (**S1**) was executed or the assignment statement (**S2**) was executed. Since the valuation function $\mathcal{V}$ is used in this specification the specification explicitly states that the precondition $sp($`process_mnemonic_input(&args,pp)`$, Q)$ to the statement **S0** may contain a side-effect. Note that if the function `process_mnemonic_input` has no side-effect then

$sp($`process_mnemonic_input(&args,pp)`$, Q) = Q$.

Further expansion of $sp(\mathbf{S1}, \mathcal{V}(B) \wedge sp(B,Q))$, and $sp(\mathbf{S2}, \neg\mathcal{V}(B) \wedge sp(B,Q))$ yield

$$\begin{array}{c} sp(\mathbf{S1}, \mathcal{V}(B) \wedge sp(B,Q)) = \\ sp(\mathbf{S1a}, (mp = NULL) \wedge \mathcal{V}(B) \wedge sp(B,Q)) \vee \\ sp(\mathbf{S1b}, (mp \neq NULL) \wedge \mathcal{V}(B) \wedge sp(B,Q)), \end{array} \quad (10)$$

and

$$\begin{array}{c} sp(\mathbf{S2}, \neg\mathcal{V}(B) \wedge sp(B,Q)) = \\ sp(\texttt{dontoutput = 1}, \neg\mathcal{V}(B) \wedge sp(B,Q)) = \\ (dontoutput = 1) \wedge (\neg\mathcal{V}(B) \wedge sp(B,Q))_v^{dontoutput} \end{array} \quad (11)$$

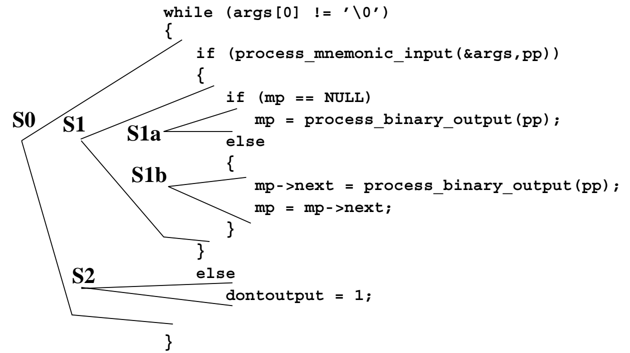


Figure 5. A Sequence of Translate Source Code

respectively, where v is the value of `dontoutput` before executing **S2**. Expression (10) states that given that the expression '$\mathcal{V}(B) \wedge sp(B,Q)$' is true, either **S1a** has been executed or **S1b** has been executed, each depending on the added condition that either $(mp = NULL)$, or $(mp \neq NULL)$, respectively. On the other hand, Expression (11) states that given that the expression '$\neg\mathcal{V}(B) \wedge sp(B,Q)$' is true, execution of **S2** results in the assignment of the variable '`dontoutput`' to be '`1`'.

The preliminary skeleton of the logical specification of the translation module can be constructed by substituting the Expressions (10) and (11) back into the original Expression (9) such that

$$\begin{array}{c} sp(\mathbf{S0}, Q) = \\ sp(\mathbf{S1a}, (mp = NULL) \wedge \mathcal{V}(B) \wedge sp(B,Q)) \vee \\ sp(\mathbf{S1b}, \neg(mp = NULL) \wedge \mathcal{V}(B) \wedge sp(B,Q)) \vee \\ (dontoutput = 1) \wedge (\neg\mathcal{V}(B) \wedge sp(B,Q))_v^{dontoutput} \end{array} \quad (12)$$

which states that in every iteration, one of three actions is executed, namely one of **S1a**, **S1b**, or **S2** (`dontoutput = 1`).

At this point in the analysis, since **S1a** and **S1b** are statements that depend on the specification of functions and procedures that are used by `translate`, it is appropriate to begin a *use analysis* for the `translate` function, where in this case, the function `process_binary_output` is analyzed.

In summary, during the local analysis phase for `translate`, a graphical representation of the function was created with the intention of determining the calling hierarchy for the function. Next, a logical analysis was performed using a top-down approach that uses encapsulation with the intention of determining the logical complexity.

5.2. Use Analysis

Use analysis involves the specification of functions that are used by a given object of study. In our example, given that the object of study is the `translate` function, use analysis involves specifying the functions used by `translate`. In this section we describe the function `process_binary_output`.

Figure 6 contains the source code for `process_binary_output`. The *use* analysis for this function involves three steps, each corresponding to the steps followed for `translate`. That is, we perform *local*, *use*, and *global* analyses on `process_binary_output`. The remaining analysis of `process_binary_output` is similar to the process used to analyze `translate`. However, in the interest of simplifying the analysis we shall ignore many of the details involved with analyzing `process_binary_output` and focus primarily on the output characteristics. Note that the strict application of the rules for *sp* require a line by line construction of a specification. Here, we informally construct the specification with the understanding that all of the information can and should be constructed rigorously. Our main objective in this example analysis is to provide enough information about `process_binary_output` to be able to describe `translate` in a sufficient manner without having to perform a full analysis of the entire command translation system. Again, we note that the code used in this paper is taken out of context. Therefore, it is unreasonable to specify this code without these constraints. Therefore, the specification given in this section is used primarily to show how a true specification of `process_binary_output` might be used to describe the `translate` function.

```
  struct msg *process_binary_output
           (struct project_parameters *pp) {
      extern U16 *stem_entry;
      U16 code;
      U16 *ep;
      struct msg *mp;
      Q = control_list;
      W = (U16 *)stack_base;
      S = (U32 *)min_S;
      mp = (struct msg *)malloc(sizeof(struct msg) +
                                  MAX_MSG_BYTES);
I:    if (mp == NULL) {
          warn("process_binary_output:
                out of memory (malloc failed)\n");
          end_cmdxlt(-1);
      }
      /* -1 for length field, written over later */
      PUSHL(mp->msg_bits - 1);
      ep = get_entry(get_U32_Q());
      P = ep + 1;
      do {
         code = *P++;
         if ((code < 1) || (code > 32)){
              warn("bad code");
              end_cmdxlt(-1);
         }
         (*output_rtn[code])();
      } while (code != RFMS);
      mp->next = NULL;
      mp->msg_len = *(mp->msg_bits - 1);
J:    if (mp->msg_len > pp->max_msg_bits) {
          fail(TOO_MANY_BITS, NULL, NULL);
          free(mp);
          return(NULL);
      }
      mp->msg_num = 0;
      copy_space_filled("", mp->start, sizeof(mp->start));
      copy_space_filled("", mp->open, sizeof(mp->open));
      copy_space_filled("", mp->close, sizeof(mp->close));
      copy_space_filled(get_stem_and_title(stem_entry),
                        mp->comment, sizeof(mp->comment));
      mp->chksum = chksum(mp->msg_bits,
                           FLD_LEN_OF(mp->msg_len)*2);
K:    return(mp);
   }
```

Figure 6. Process Binary Output Source

Consider the code given in Figure 6 for the function `process_binary_output`. There are three statements that determine whether or not the output of the function is defined. These are indicated by the line numbers `I`, `J`, and `K`, respectively. Line `I`, for instance, has the interpretation that if space could not be allocated for the return object, then the routine aborts, while line `J` forces the routine to return a `NULL` object due to some other error. Finally, the line `K` indicates a successful return of an object. Therefore, we can construct the following specification for `process_binary_output`:

$$\begin{aligned} & sp(warn;\, end_cmdxlt, (mp = NULL) \wedge Q) \;\vee \\ & \quad sp(fail;\, free;\, return(NULL), \\ & \quad (\texttt{mp->msg_len} > \texttt{pp->max_msg_bits}) \wedge \\ & \quad (mp \neq NULL) \wedge Q)) \;\vee\; sp(return(mp), \\ & \quad (\texttt{mp->msg_len} \leq \texttt{pp->max_msg_bits}) \wedge \\ & \quad (mp \neq NULL) \wedge Q) \end{aligned} \tag{13}$$

which states that after executing the function `process_binary_output` either `warn` and `end_cmdxlt` were executed, the routine returned a `NULL` object, or the routine returned a valid object. Again, we stress that this specification is incomplete and only specifies a small slice of the functionality of the routine.

5.3. Global Analysis

The final step in the analysis is to take the specification of Expression (14) and integrate it back into the skeleton specification of Expression (13). This specification is as follows

$$\begin{aligned} sp(\mathbf{S0}, Q) = & ((mp = NULL) \vee (mp = u)) \vee \\ & (((mp\text{-}{>}next = NULL) \vee (mp\text{-}{>}next = u)) \wedge \\ & (mp = mp\text{-}{>}next)) \vee (dontoutput = 1) \wedge (\neg \mathcal{V}(B) \\ & \wedge sp(B, Q))_v^{dontoutput} \end{aligned} \tag{14}$$

where *u* is some new object. This specification states that after executing **S0**, the variable `mp` has either the value

NULL or points to some new object, or `mp->next` has the value NULL or points to some new object with `mp` pointing to `mp->next`. Finally, if neither of those cases holds, it must be that `dontoutput = 1`. In the context of the specification of Expression (8), this specification means that after each iteration, a chain of messages is constructed or the `dontoutput` flag is set to 1. Note that in this specification we make the assumption that the pointer assignment behaves like a variable assignment. In this case there is no impact on the specification by making this assumption since no reference is made to the allocated data objects outside of simple assignments.

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) [12]. 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 (Restructuring, Maintenance, Validation and Documentation of Software Systems) was an Espirit II project whose objective was to improve applications by making them more maintainable through the use of reverse engineering techniques. In order to reverse engineer COBOL programs, this approach uses a set of general guidelines to derive objects and specifications from original code, where a framework is developed to reason about objects.

The "Loop ANalysis Tool for Recognizing Natural concepts" or LANTRN is an approach that uses a multi-step process to construct predicate logic annotations for loops. The analysis process involves the translation and normalization of loop programs into forms that are amenable to matching of various components of loops. A knowledge-base or *plan library* is used to identify stereotypical loop *events*, where events come in the form of *basic events* and *augmentation events*.

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.

Currently we are developing a system to support all of the techniques described in this paper called AUTOSPEC. In addition, we have been applying our techniques to a ground-based mission control system for controlling unmanned spacecraft at the NASA Jet Propulsion Laboratory. Our future investigations include the development of an approach to introducing abstraction into the "as-built" specifications to obtain design level specifications.

References

[1] E. Byrne. A Conceptual Foundation for Software Re-engineering. In *Proceedings for the Conference on Software Maintenance*, pages 226–235. IEEE, 1992.

[2] B. H. C. Cheng. Applying formal methods in automated software engineering. *Journal of Computer and Software Engineering*, 2(2):137–164, 1994.

[3] B. H. C. Cheng and B. Auernheimer. Applying Formal Methods and Object-Oriented Analysis to Existing Flight Software. In *Proceedings of the 18th Annual NASA Software Engineering Workshop*, pages 274–282. NASA, December 1993.

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

[5] G. C. Gannod and B. H. C. Cheng. Facilitating the Maintenance of Safety-Critical Systems Using Formal Methods. *The International Journal of Software Engineering and Knowledge Engineering*, 4(2), 1994.

[6] G. C. Gannod and B. H. C. Cheng. Formal Methods Applied to Reverse Engineering: Summer Report. Technical Report MSU-CPS-95-44, Michigan State University, December 1995.

[7] G. C. Gannod and B. H. C. Cheng. Strongest Postcondition as the Formal Basis for Reverse Engineering. *To appear in Journal of Automated Software Engineering. An earlier version of this paper appeared in the Proceedings for the Second Working Conference on Reverse Engineering*, 1996.

[8] H. Haughton and K. Lano. Objects Revisited. In *Proceedings for the Conference on Software Maintenance*, pages 152–161. IEEE, 1991.

[9] C. A. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[10] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[11] W. M. Osborne and E. J. Chikofsky. Fitting pieces to the maintenance puzzle. *IEEE Software*, 7(1):11–12, January 1990.

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

[13] J. M. Wing. A Specifier's Introduction to Formal Methods. *IEEE Computer*, 23(9):8–24, September 1990.

[14] E. Yourdon and L. Constantine. *Structured Analysis and Design: Fundamentals Discipline of Computer Programs and System Design*. Yourdon Press, 1978.