

A Formal Approach for Reverse Engineering: A Case Study*

Gerald C. Gannod[†]
Computer Science & Engineering
Arizona State University
Box 875406
Tempe, AZ 85287-5406
E-mail: gannod@asu.edu

Betty H.C. Cheng
Computer Science & Engineering
3115 Engineering Building
Michigan State University
East Lansing, MI 48824
E-mail: chengb@cse.msu.edu

Abstract

As a program evolves, it becomes increasingly difficult to understand and reason about changes in the source code. Eventually, if enough changes are made, reverse engineering and design recovery techniques must be used in order to understand the current behavior of a system. In this context, the effective use of complementary approaches can facilitate program and system understanding by taking advantage of the relative benefits of different approaches. This paper presents an approach to reverse engineering that combines the use of both informal and formal methods and describes a case study project involving the reverse engineering of a mission control system used by the NASA Jet Propulsion Laboratory to command unmanned spacecraft.

1 Introduction

On 13 June 1996, an inquiry board began a review of the failed maiden flight of the Ariane 5 launcher. One of the final recommendations of the review board was that the definition of critical components should be reconsidered and modified to include software [1]. As software continues to be used to control critical systems, it is increasingly likely that software will be considered critical in order to achieve mission goals. Consequently, ensuring the correctness of software becomes paramount in order to avoid catastrophic failures like that of the Ariane 5.

Formal methods are techniques that incorporate the use of formal specification languages, where a formal specification language has a well-defined syntax and semantics. In

addition, formal methods have associated calculation rules that can be used to analyze specifications in order to determine correctness and consistency. Since the notations have a formal mathematical basis, formal methods facilitate the use of automated processing [2]. Reverse engineering of program code is the process of examining the components and component interrelationships in order to construct a high-level abstraction of an implementation [3]. Re-engineering is the process of examination, understanding, and alteration of a system with the intent of implementing the system in a new form [3]. The benefits offered by re-engineering versus developing software from the original requirements are considered to be a better solution for handling legacy code, given that the actual code frequently differs from existing documentation. Since much of the functionality of the existing software has been achieved over a period of time, it must be preserved for many reasons, including providing continuity to current users of the software. The primary focus of our research is to apply the use of formal methods to the reverse engineering of program code in order to support maintenance and evolutionary activities, where the formal approach facilitates automated processing.

In previous investigations, we developed a formal technique for reverse engineering that is based on the use of the strongest postcondition predicate transformer [4, 5, 6]. In addition, we introduced an approach for constructing abstractions of specifications that are generated using the strongest postcondition [7]. In this paper, we describe a case study that applies a methodology that we have developed for using informal and formal techniques for reverse engineering. In the case study we apply the methodology to a mission control application used at the NASA Jet Propulsion Laboratory to control unmanned spacecraft.

The remainder of this paper is organized as follows. Section 2 discusses background material in the areas of formal methods and reverse engineering. The methodology for combining informal and formal techniques for reverse en-

*This work supported in part by the National Science Foundation grants CDA-9617310, CCR-9633391, CCR-9407318, CCR-9209873, and CDA-9312389, and NASA Training grant NGT-70376.

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

[‡]Contact author.

engineering is described in Section 3. Section 4 describes the case study application of the reverse engineering approach to a NASA mission control system. Section 5 discusses related work, and Section 6 draws conclusions and suggests further investigations.

2 Background

In this section we describe a formal reverse engineering technique that is based on the use of the *strongest postcondition* predicate transformer and order preserving transformations.

2.1 As-Built Specifications

The *strongest postcondition* (denoted sp) is defined as the strongest condition R that is true after the execution of a program S , when starting with condition Q true. Table 1 summarizes the strongest postcondition semantics of the Dijkstra guarded command language [8], where IF represents the n alternative conditional statement

$$\begin{array}{l} \text{if } B_1 \rightarrow S_1; \\ \quad \dots \\ \quad || B_n \rightarrow S_n; \\ \text{fi;} \end{array}$$

$B_i \rightarrow S_i$ represents a guarded command such that S_i is only executed if logical expression (guard) B_i is true. DO represents the loop statement “do $B \rightarrow S$ od” where S is executed iteratively until guard B is false.

Construct	sp Semantics
$sp(x := e, Q)$	$\equiv (\exists v :: Q_v^x \wedge x = e_v^x)$
$sp(\text{IF}, Q)$	$\equiv sp(S_1, B_1 \wedge Q) \vee \dots \vee sp(S_n, B_n \wedge Q)$
$sp(\text{DO}, Q)$	$\equiv \neg B \wedge (\exists i : 0 \leq i : sp(\text{IF}^i, Q))$
$sp(S_1; S_2, Q)$	$\equiv sp(S_2, sp(S_1, Q))$

Table 1. Strongest Postcondition Semantics

In the table, the semantics for $sp(x := e, Q)$ states that after the execution of “ $x := e$ ” there exists some value v such that every free occurrence of x in Q is replaced with v and $x = e_v^x$. The semantics for $sp(\text{IF}, Q)$ states that after execution of the if-f i statement, at least one of $sp(S_i, B_i \wedge Q)$ is true. In the case of iteration, denoted $sp(\text{DO}, Q)$, the semantics are that after execution of the loop, the loop guard is false ($\neg B$), and a disjunctive expression describing the effects of iterating the loop some number of times (approximated by the notation IF^k) is true, where $k \geq 0$. Finally, for sequences, $sp(S_1; S_2, Q)$ means that the postcondition for statement S_1 is the precondition for some subsequent statement S_2 .

In previous investigations, we have described the use of sp as the formal basis for reverse engineering [4]. In addition, we have addressed the use of sp to define the semantics of the C programming language in order to reverse engineer

C programs [5], as well as programs that contain the use of pointers and pointer operations [6], and procedure calls [4]. To show the applicability of the sp approach to real systems, we have applied the technique to a ground-based mission control system used by the NASA Jet Propulsion Laboratory to control spacecraft during planetary missions [5].

2.2 Deriving More Abstract Specifications

The specifications that are constructed using the approach described in Section 2.1 are considered to be “as-built” since they are derived from source code, and thus represent behavior based on the final product rather than the original design. As a result, the specifications contain a significant amount of algorithmic and implementation detail. We have defined an approach for generalizing as-built formal specifications with the abstraction match as the guiding principle, where an abstraction match is defined as follows [7]:

Definition 1 (Abstraction Match) *Let \mathcal{I} be a program with specification i such that the corresponding precondition and postcondition are i_{pre} and i_{post} , respectively, and let l be an axiomatic specification with precondition l_{pre} and postcondition l_{post} . A match is an **abstraction match** if $i \preceq l$, so that*

$$(l_{pre} \rightarrow i_{pre}) \wedge (i_{post} \rightarrow l_{post}).$$

One of the interesting properties of the abstraction match operator is that it forms a partial-order relation [7]. As such, high-level abstractions of pre and postcondition specifications can be derived as long as the abstraction match relationship is preserved. That is, given a pre and postcondition specification I that consists of precondition I_{pre} and postcondition I_{post} , we would like to identify an axiomatic specification A such that $I \preceq A$. We can identify such a specification by modifying I so that we have a specification I' that satisfies the relationship that $I \preceq I'$. If, for instance, \preceq is the abstraction match operator, then by either strengthening the precondition I_{pre} , weakening the postcondition I_{post} , or both, we produce a specification I' that satisfies the property that $I \preceq I'$. That is, the abstraction match operator is used as a constraint for *deriving* abstractions rather than by searching a library of specifications for appropriate matches. In order to address the computational complexities of deriving abstractions based on match preservation, we have developed a number of guidelines that focus on weakening the postcondition I_{post} and strengthening the precondition I_{pre} [7].

One technique for preserving an abstraction match relationship is by weakening the postcondition of a specification. For example, let I be a specification with precondition I_{pre} and postcondition I_{post} and let I' be a specification such that $I'_{pre} \leftrightarrow I_{pre}$ and $I_{post} \rightarrow I'_{post}$. As such, $I \preceq I'$,

since

$$\begin{aligned} & ((I'_{pre} \leftrightarrow I_{pre}) \wedge (I_{post} \rightarrow I'_{post})) \\ & \Rightarrow \\ & ((I'_{pre} \rightarrow I_{pre}) \wedge (I_{post} \rightarrow I'_{post})). \end{aligned} \quad (1)$$

Expression (1) provides a basis for deriving abstractions from a specification by weakening a postcondition I_{post} to produce a postcondition I'_{post} . Several options are available for weakening the postcondition including those listed in Table 2, which includes *delete a conjunct*, *add a disjunct*, transform \wedge to \vee , and transform \wedge to \rightarrow . In our previous investigations, we described several strategies for weakening postconditions based on module and specification characteristics [7].

Operation	I_{post}	I'_{post}
Delete a conjunct	$A \wedge B \wedge C$	$A \wedge C$
Add a disjunct	$A \wedge B$	$(A \wedge B) \vee C$
\wedge to \rightarrow	$A \wedge B$	$A \rightarrow B$
\wedge to \vee	$A \wedge B$	$A \vee B$

Table 2. Weakening the postcondition

3 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. Many semi-formal methods use notations that are graphical, thus facilitating ease of use in their application. The drawback to semi-formal methods is that the notations are typically imprecise and ambiguous. This section describes an approach to reverse engineering that combines the use of semi-formal and formal methods in order to benefit from the complementary advantages of both approaches.

3.1 Structured Analysis

Although the recent trend in software development has been to build systems using object-oriented technology, a majority of existing systems has 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) [9]. 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 yield 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 a low-level graphical description known as a call graph or structure chart. 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 as a guide for constructing formal specifications.

In general, the construction of the graphical descriptions proceeds in two phases. In the first phase, a high-level model is constructed that is based on information gathered from user manuals or high-level design descriptions. In the case that these documents do not exist, it is appropriate to incorporate user interviews, and if possible, empirical testing to determine high-level behavior. In the second phase, a low-level model in the form of call graphs and/or control-flow graphs is constructed.

3.2 Formal techniques and large systems

One of the limitations of our technique is that the specifications that are constructed can grow to be exponential in size with respect to the input program. Given this limitation, the appropriateness of using formal methods in the context of large systems must be well-understood. That is, the use of formal methods for reverse engineering, as is the case with all applications of formal methods, must be targeted to those contexts where it has the highest payoff; namely critical systems [10, 11]. However, in the reverse engineering of software, we can extend the context a bit further to include the parts of a software system that are deemed “critical”. In order to determine those critical portions of the software, several factors must be taken into account, including call graph and flow graph complexity.

3.3 Applying formal techniques

The motivation for using both semi-formal and formal methods is two-fold. First, it is clearly useful 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 (e.g., in critical systems or critical routines), 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.

In previous investigations, we described a three phase approach for deriving formal specifications from programs [5]. The three steps involve a *local analysis*, *use analysis*, and *global analysis*. During the *local analysis* phase, the calling hierarchy of a module is constructed and a skeletal formal specification is built using the strongest postcondition. 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 final step, 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. The following definition summarizes the method described above.

Definition 2 (Structure-Based Analysis Method)

Let M be a program with statements m_1, \dots, m_k , and $P = \{P_1 \dots P_n\}$ be the set of procedures called by M . In addition, let Q be the precondition for M . A statement m_i is in P if there is a procedure p_j in P such that program M calls p_j at line i of M .

Method $R(M, Q)$

1. *Local Analysis:*

- (a) Apply $sp(m_1; \dots ; m_k, Q)$
- (b) For each i such that $m_i \in \{P_1, \dots, P_n\}$
 set $sp(m_i, sp(m_1; \dots ; m_{i-1}, Q)) \quad :=$
 “ $sp(m_i, sp(m_1; \dots ; m_{i-1}, Q))$ ”.
 (We refer to the right hand side of the assignment as the skeleton. Also, note that the set $\{P_1, \dots, P_n\}$ can be derived using a call graph.)

2. *Use Analysis:* For each critical function $p \in \{P_1, \dots, P_n\}$, apply $R(p, Q_p)$, where Q_p is the precondition to the procedure p .

(For this step, a critical function is determined by several criteria including importance of the function with respect to required functionality, if known, and call graph connectivity.)

3. *Global Analysis:* Replace skeletons from Step 1b with results of Step 2.

3.4 Abstraction

After constructing an as-built formal specification using the process described in Section 3.3, an abstraction of the specification can be constructed using the approach described in Section 2.2. In addition, we have found it appropriate to apply the abstraction steps during intermediate steps of the method in Definition 2 in order to aid in reducing the logical complexity of the specifications. As such, we can modify the structure-based method to be as follows by adding a fourth step.

Definition 3 (Structure-Based Analysis Method (Ver. 2))

Let M be a program with statements m_1, \dots, m_k , and $P = \{P_1 \dots P_n\}$ be the set of procedures called by M . In addition, let Q be the precondition for M . A statement m_i is in P if there is a procedure p_j in P such that program M calls p_j at line i of M .

Method $R'(M, Q)$

1. *Local Analysis:*

- (a) Apply $sp(m_1; \dots ; m_k, Q)$. Abstraction method may be used after each application of *sp*.
- (b) For each i such that $m_i \in \{P_1, \dots, P_n\}$
 set $sp(m_i, sp(m_1; \dots ; m_{i-1}, Q)) \quad :=$
 “ $sp(m_i, sp(m_1; \dots ; m_{i-1}, Q))$ ”.
 (We refer to the right hand side of the assignment as the skeleton. Also, note that the set $\{P_1, \dots, P_n\}$ can be derived using a call graph.)

2. *Use Analysis:* For each critical function $p \in \{P_1, \dots, P_n\}$, apply $R(p, Q_p)$, where Q_p is the precondition to the procedure p .

(For this step, a critical function is determined by several criteria including importance of the function with respect to required functionality, if known, and call graph connectivity.)

3. *Global Analysis:* Replace skeletons from Step 1b with results of Step 2.

4. Apply the abstraction method to the final specification.

3.5 Process Summary

Given the definition in Section 3.4, a process for reverse engineering of programs that combines informal and formal techniques can be summarized as follows:

Definition 4 (Combined Method)

- 1. Construct an informal high-level model of the software
- 2. Construct an informal low-level model of the software, including a call graph
- 3. Apply R' to a module M , where M is chosen using some selection criteria.

One of the primary difficulties in the process is the determination of the criteria that can be used for Step 3. The criteria that we have used includes the identification of critical procedures by examining the call graph constructed in Step 2. Our selection of critical procedures is typically based on choosing those vertices in a call graph with a large difference between the in-degree and out-degree. Other criteria that can be used include keyword search and data structure usage.

4 Case Study

In this section, we apply our technique to a portion of the command translation system that is responsible for processing user mnemonics (messages). Failure in the translation system can have several impacts including erroneous messages being transmitted to spacecraft. Our objective is to investigate various properties of the system such as termination and translation failure.

4.1 Overview

In this section we provide an overview of the case study system and outline the objectives of the analysis.

4.1.1 System Overview

The Command Subsystem provides access and facilitates the command and control of spacecraft via a user interface. The system supports the control of multiple spacecraft and provides real-time feedback about the status of radiating commands at each operational point during the uploading of commands to the spacecraft. In addition, the Command subsystem supports command file reformatting and direct access to project databases.

The overall Command process is a six-step sequence as follows:

1. A user accesses the Command Subsystem and prepares a mnemonic command file.
2. The system translates the mnemonic file to binary format.
3. The binary file is converted into a format required by the Deep Space Network (DSN) for radiation (i.e., transmission).
4. The Command file is transferred to the DSN for radiation to the spacecraft.
5. The user can then control and monitor the command file from the workstation.
6. Exit.

The *command translation* module of the command subsystem is responsible for two of the items in the sequence, namely items 2 and 3. The overall size of the command

translation subsystem is approximately five thousand lines of code while the overall command system is approximately fifty thousand lines of code. The command translation system has an interesting history that motivates the analysis of the software. The system was originally developed to support the control of a specific set of spacecraft. Every time a new mission is developed (for example, the 1997 Cassini mission to Saturn), the software is updated to handle the translation of spacecraft-specific mnemonics. Given that the system is in a constant change of flux, analysis of the command translation software using reverse engineering techniques is well-justified.

4.1.2 Analysis Objectives

In the remainder of this section, we analyze the command translation subsystem in order to demonstrate the use of a combined informal and formal technique for reverse engineering. The primary objective of the case study is to illustrate how a formal method can be used along with informal methods to derive information about the functionality of a software system.

The application of formal methods to large systems has yet to be effectively demonstrated. However, formal methods have been shown to yield the highest payoff when applied to systems that are critical in nature. In the area of reverse engineering, the highest payoff for formal methods occurs not when applied to an entire system, but rather when applied to a *critical* part of a system.

4.2 Project-Specific Process

In this case study, the high-level informal analysis was facilitated by the existence of documentation that was written by the original developers of the software.

Besides the demonstration of the utility of the combined informal and formal approach described in Section 3, the goal of the case study was to examine the source code in order to continually refine our knowledge about various properties of the system. The focus of the analysis was to identify termination conditions which we later refined once the code was studied informally.

One of the assumptions that was made concerning the existing documents was that constant modifications to the software were not reflected in the documentation. Due to this lack of document maintenance, it was assumed that only a certain level of detail from the documents could be determined to be reliable.

The low-level informal analysis was based on the construction of source models (i.e., call graphs) in order to recover structural information about the system. Using some standard visualization tools, the source models were used to determine potential *points of failure*. In this context, we use the phrase *point of failure* to mean those parts of the source model where there is a significant difference between the in-degree and out-degree of a vertex in the call graph, or where

the cardinality of the in- or out-degrees is greater than certain threshold values. The reason that these vertices of the graph are interesting is that the high out-degree means that a procedure invokes many other procedures and thus is potentially a critical procedure. High in-degree vertices in a graph indicate that a procedure is called often and thus is also a potentially critical procedure.

The formal analysis follows a top-down, bottom-up approach as described in Section 2. In the analysis we focused primarily on issues of mnemonic translation and spacecraft message construction. Our intent was to examine properties of *process termination* and *process failure*. With process termination, we were interested in determining what conditions were required for ensuring that the translation process terminates and for process failure, we were interested in determining what conditions force the translation process to fail. The remainder of this section focuses on the low-level analysis of the command and command translation subsystems. For a description of the high-level analysis, see [12].

4.3 Low-Level Analysis

The software for the command system, written in C, was organized into several directories that were partitioned by subsystem. Accordingly, the command translation system resided in a single directory. We began our analysis by first using a combination of tools ranging from call graph browsers to source file editors. In addition we used the unix command “*grep*” to perform keyword searches.

The first step involved the construction of the call graph for the *main* procedure for the command translation system. One of the cues that was used for identifying procedures to be analyzed was procedure names. In the case of the command translation system, we were interested in analyzing the *translation* subsystem.

Figure 1 shows the source code for the `translate` routine of the command translation subsystem. The corresponding call graph is given in Figure 2. During the informal analysis, the source code and call graph were used in tandem in order to help identify procedures that required further study. For instance, the source code in Figure 1 consists of a switch statement with three cases: (1) `INIT`, (2) `XLT`, and (3) `CARG`. These cases correspond to different operating modes for the software for initialization, translation, and command file copying, respectively. In our analysis we were interested in the `XLT` or *translation* mode and so two functions, `process_mnemonic_input` and `process_binary_output` were tagged as requiring further study.

The next step in the process involved an analysis of Figure 2 which led to the observation that the `process_msg` procedure controls a majority of the mnemonic input processing. Specifically, given that the `process_msg` has a large difference between the out-degree and in-degree,

```

struct msg *translate(op, args)
    int op;
    char *args;
{
    extern int dontoutput;
    static struct project_parameters *pp;
    struct msg *mp = NULL;
    switch (op){
        case INIT:
            pp = initialize_interpreter();
            break;
        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;
        case CARG:
            process_carg(&args, pp);
            break;
        default:
            inform_user(
                "internal error: bad op in translate");
            end_cmdxlt(CMD_ERROR);
    }
    /* only translation returns a value;
       return NULL on error or no value */
    return(mp);
}

```

Figure 1. Translate source code

with the out-degree dominating, it led us to identify `process_msg` as a critical procedure.

At this point in the study of the command translation system we were able to formulate questions to be answered by the formal specification phase of the analysis. For instance, a quick analysis of the `process_msg` procedure, shown in Figure 3, revealed that a loop is executed until the value of the variable `sp->msg_complete = 1`. Using this information, we were interested in determining when the value of `sp->msg_complete` changes from 0 to 1. In addition, given that the return value of the `process_msg` procedure is the negation of `sp->failed`, we were also interested in determining what conditions needed to be present in order for `sp->msg_complete = 1` and `sp->failed = 1` or `sp->failed = 1`. These cases would indicate that the message was syntactically correct and that the processing either failed or succeeded. Specifically, we were interested in the case where the message was constructed correctly but the processing still failed.

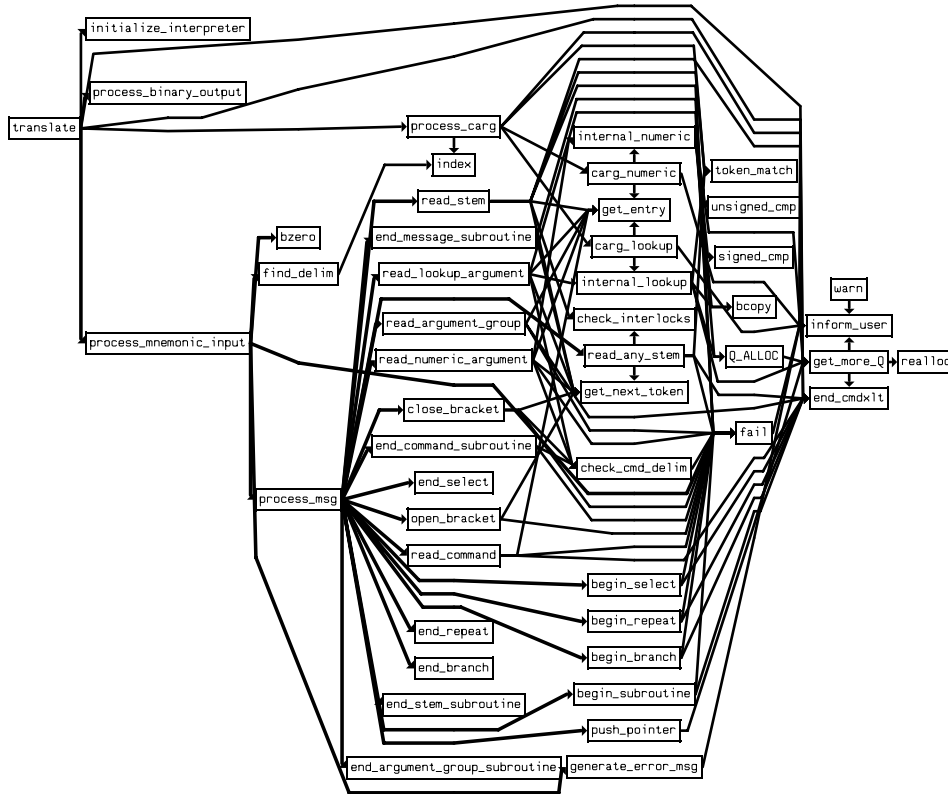


Figure 2. Translate source model

4.4 Formal Analysis

Prior to the formal analysis of the command translation system, the following details about the functionality had been determined via the informal analysis of the source code:

- The sequence of calls originating from the `translate` procedure and proceeding to `process_mnemonic_input` and finally to `process_msg` constitute a “critical path” of execution.
- The `process_msg` routine terminates only when the variable `sp->msg_complete` variable is set to the value 1.
- The routine `end_cmdxlt` is invoked by every one of the `begin_*` routines (among others) as shown in Figure 2. This led to the conjecture that `end_cmdxlt` is a critical procedure.

Using this information, we formulated the following questions to be answered by the formal analysis:

- What are the conditions for terminating the translation process.

- What are the routines that exhibit representative behavior for successful and unsuccessful translation? That is, given a known termination condition for the routine `process_msg`, what routines establish `sp->msg_complete = 1`?
- Are there other terminating paths that bypass `process_msg`?

In an attempt to answer these questions, we analyzed several procedures that potentially had an impact on the issues outlined above. That is, we analyzed the `process_mnemonic_input`, `process_msg`, and `end_cmdxlt` procedures, as well as the procedure named `end_message_subroutine`. We identified `end_message_subroutine` as a routine of interest after using the `grep` command to locate the places in the code where the variable `sp->msg_complete` was assigned a value of 1.

4.4.1 Analysis of `process_mnemonic_input`

Figure 4 shows a sequence of code from the `process_mnemonic_input` procedure. In lines 2-11 the `do-while` loop contains several assignment statements and a

```

0.  int process_msg(ep, tp, sp, parms)
1.      U16 *ep;
2.      struct tokens *tp;
3.      struct interp_state *sp;
4.      struct project_parameters *parms;
5.  {
6.      U16 code;
7.
8.
9.      if (ep[1]&SITE_BIT)
10.     {
11.         sp->failed = 1;
12.         fail(EXCLUDED_MSG, tp, sp);
13.         return(0);
14.     }
15.
16.     P = ep + 3 + ep[2];
17.     sp->msg_level = 1;
18.     sp->msg_complete = 0;
19.
20.
21.     while (!sp->msg_complete)
22.     {
23.         /* on failure, a new value for P
24.            will be on top of the stack */
25.         if (sp->failed)
26.             P = (U16 *)STACK(0);
27.         code = *P++;
28.         (*(rtn[code]))(tp, sp, parms);
29.     }
30.
31.     return(!sp->failed);
32. }

```

Figure 3. process_msg source code

call to the process_msg procedure. The call is used to guard a break statement that, in essence, provides another termination condition for the loop. As such, the terminating condition for this loop is the following:

$$(process_msg(ep, tp, sp, parms) = 1) \vee (ep = collection[249]). \quad (2)$$

Expression 3 states that either the process_msg routines returns 1, or the ep pointer takes the value of the 250th element of the collection array. The significance of the number 249 is that the ep pointer is used as a cursor to refer to the current position in a message. When the entire input has been processed, the ep pointer is moved to the adjacent memory locations until, finally, it refers to the next element in the collection array. The more interesting aspect of the terminating condition in Expression 3 is the term $process_msg(ep, tp, sp, parms) = 1$. In this case, we need to analyze the process_msg procedure in order to determine when process_msg returns 1.

4.4.2 Analysis of process_msg

Consider again the source code in Figure 3. At line 21, the statement `return(!sp->failed)` indicates that the program returns the negated value of the `sp->failed` variable. Since line 6 of the program in Figure 4 states that `sp->failed = 0`, it is reasonable for us to infer that

```

0.  ep = get_first_entry(248);
1.
2.  do {
3.      tp->token_index = tp->t;
4.      Q = control_list;
5.      sp->num_of_commands = 0;
6.      sp->failed = 0;
7.      sp->cmd_delimiter_deferred = 0;
8.
9.      if (process_msg(ep, tp, sp, parms))
10.         break;
11.  } while ((ep = get_next_entry(ep)) !=
12.          collection[249]);
13.
14.  if (sp->failed)
15.      generate_error_msg(sp, tp);
16.
17.  *strp = s;
18.  stem_entry = sp->stem_name;
19.
20.  return(!sp->failed);

```

Figure 4. Source code sequence for process_mnemonic_input

$(coset(sp).failed = 0)$ is a precondition for the process_msg procedure, where the coset function is used to associate a pointer variable to a set of equivalent reference objects [6]. Given this precondition, consider lines 26 - 39. Line 28 establishes the condition that `sp->msg_complete = 0`, so in the initial iteration of the loop, $(coset(sp).failed = 0) \wedge (coset(sp).msg_complete = 0)$. Using strongest postcondition, then, to formally specify the loop, we obtain the following postcondition (as generated by AUTOSPEC):

```

/* AutoSpec:
"(((coset(sp).msg_complete.V == 1) /\
  (((((R_i-1 /\ (coset(sp).failed.V != 0)) /\
  (as_const8 = S[0])) /\
  ((R_i-1 /\ (!(coset(sp).failed.V != 0))) /\
  (suif_tmp0 .> coset(P)) /\
  (P.V = ((2 * 1) + suif_tmp0.V)) /\
  (code.V = coset(suif_tmp0.V))) /\
  sp(rtn[(int)code](tp, sp, parms), R_i)))" */

```

where the term R_i is used to represent the i th iteration of the loop. The specification states that message processing is complete and that either the message processing failed or it was successful. The term (last line) `sp(rtn[(int)code](tp, sp, parms), R_i)` is the specification of the various calls to the procedures listed in lines 1-7 in Figure 3. In order to determine if after the loop is executed that indeed $(coset(sp).msg_complete.V == 1)$, we must analyze the various procedures.

4.4.3 Analysis of end_message_subroutine

Figure 5 contains the annotated source code for the end_message_subroutine procedure. After performing a *grep* search for the references to the variable

sp->msg_complete, it was determined that in only one location throughout the code is the value of sp->msg_complete set to 1.

```
extern void end_message_subroutine(tp, sp, parms)
struct tokens *tp;
struct interp_state *sp;
struct project_parameters *parms;
{
    S = (unsigned int *)((char *)S + 4 * 1);
    sp->msg_complete = 1;

/* AutoSpec:
R_1: ((((((parms .> _param5) /\
(_param5.V == _pVal6)) /\
((sp .> _param4) /\
(_param4.V == _pVal5)) /\
(tp .> _param3) /\
(_param3.V == _pVal4)))) /\
(S.V = ((4 * 1) + as_const4))) /\
(coset(sp).msg_complete.V = 1)) */

    if (sp->failed != 0) {
        return;
    }

/* AutoSpec:
"((R_1 /\ (coset(sp).failed.V != 0)) /\
(R_1 /\ (!(coset(sp).failed.V != 0))))" */

    if (get_next_token(tp) != (void *)0) {
        sp->failed = 1;
        fail("End of message expected", tp, sp);
    }

/* AutoSpec:
"(((R_1 /\ (!(as_const6 != 0))) /\
(get_next_token(tp.V) != 0)) /\
(coset(sp).failed.V = 1)) /\
((R_1 /\ (!(coset(sp).failed.V != 0))) /\
(!(get_next_token(tp.V) != 0))))" */

    return;

/* AutoSpec:
"R_1 /\ (((!(as_const6 != 0)) /\
(get_next_token(tp.V) != 0)) /\
(coset(sp).failed.V = 1)) /\
(!(coset(sp).failed.V != 0)) /\
(!(get_next_token(tp.V) != 0))))" */

}

/* AutoSpec:
"(coset(sp).msg_complete.V = 1) /\
(((!(as_const6 != 0)) /\
(get_next_token(tp.V) != 0)) /\
(coset(sp).failed.V = 1)) /\
(!(coset(sp).failed.V != 0)) /\
(!(get_next_token(tp.V) != 0))))" */
```

Figure 5. Annotated source code for end_message_subroutine

The original specification for the end_message_subroutine procedure as generated by AUTOSPEC is as follows:

```
/* AutoSpec:
"(((((((parms .> _param5) /\
(_param5.V == _pVal6)) /\
((sp .> _param4) /\
(_param4.V == _pVal5)) /\
(tp .> _param3) /\
(_param3.V == _pVal4)))) /\
(S.V = ((4 * 1) + as_const4))) /\
(coset(sp).msg_complete.V = 1)) /\
(!(as_const6 != 0)) /\
(get_next_token(tp.V) != 0)) /\
(coset(sp).failed.V = 1)) /\
((R_1 /\ (!(coset(sp).failed.V != 0))) /\
(!(get_next_token(tp.V) != 0))))" */
```

Since we were interested in conditions related to message processing completion and failure, we were able to use the SPECGEN system to derive an abstraction based on deleting conjuncts. The resulting postcondition specification of the end_message_subroutine procedure is as follows:

```
/* AutoSpec:
"(coset(sp).msg_complete.V = 1) /\
(((!(as_const6 != 0)) /\
(get_next_token(tp.V) != 0)) /\
(coset(sp).failed.V = 1)) /\
((coset(sp).failed.V = 0) /\
(!(get_next_token(tp.V) != 0))))" */
```

This specification states that after executing this procedure, $(coset(sp).msg_complete.V = 1)$ and that either $(coset(sp).failed.V = 1)$ or $(coset(sp).failed.V = 0)$. In the case that $(coset(sp).failed.V = 1)$, the *get_next_token* procedure returned a non-zero value, indicating the message stream buffer was not empty. Conversely, in the case that $(coset(sp).failed.V = 0)$, the message processing was successfully completed.

The completion of the above specification allowed us to answer the question concerning the conditions for the termination of the translation process. In doing so, it was determined that in the event the *end_message_subroutine* never appears on the message stack, the *process_msg* procedure can potentially run forever (or at least until there is a message stack overflow).

4.4.4 Analysis of end_cmdx1t

Given our earlier observation about the termination of *process_msg*, we proceeded to analyze whether or not other conditions can cause the *process_msg* to terminate.

In the command translation source code there are several macros that are used to access the message stack. One such macro is given in Figure 6. The code contained in this macro, upon accessing the stack, will generate a failure condition and terminate the entire program if the stack overflows. The importance of this macro is that several routines called by *process_msg* utilize this stack macro. As such, if the failure conditions are met, then the procedure *end_cmdx1t* will be called.

The formal specification of the *end_cmdx1t* is shown in Figure 7. The most important aspect of this routine is that it terminates the entire program if invoked. As such, the final specification of the program is “false”, indicating that the routine will never reach the *return* statement at the end of the code. Given this fact, the command translation system,

```

#define POPM(m) S+=(int)(m); \
if (((U16 *)S<W) || (S>min_S)) \
{ \
fail("stack overflow", NULL, NULL); \
end_cmdxlt(-1); \
}

```

Figure 6. The POPM Macro

specifically the `process_msg` procedure and subsequently the, `process_mnemonic_input` procedure, will terminate either by a successful (or partially successful) completion of a message translation, or by an eventual termination via the `end_cmdxlt` procedure.

4.5 Discussion

In the process of performing the case study, several discoveries concerning the structure and functionality of the command translation system were gathered. In addition to revealing functional properties of the system software, the case study allowed us to discover several non-functional properties regarding the code. In this section, we summarize the case study analysis.

Command translation. The command translation system provides two types of command file interpretation: user mnemonic translation and command file conversion. In addition, it was determined that the command translation system relies heavily upon communicating with other Command subsystems via the use of system files. From a low-level perspective, the `process_mnemonic_input` and `process_msg` procedures are two of the most critical procedures in the system. These procedures either directly or indirectly control the command translation process and they constitute a critical path of execution.

Termination. The termination of the command translation process depends heavily upon the termination of the `process_msg` procedure. The `process_msg` procedure terminates in one of two ways: gracefully or by fault.

Global Variables and Macros. The command translation system relies heavily upon the use of global variables and macros. While the source code is visually compact, the functional complexity seemed to increase with each encounter of one of these constructs.

4.6 Lessons Learned

Several lessons about our reverse engineering approach were learned while performing the case study described in this paper. This section summarizes these lessons.

4.6.1 Combined Analysis Technique

The utilization of a combined informal and formal process enhanced the usefulness of both the informal and formal techniques. The informal analysis provided a structured method for early discovery and organization of the functionality of the system. During the low-level analysis, the informal techniques provided valuable information and cues regarding where to focus the formal analysis. The formal analysis facilitated the functional understanding of the underlying logic embedded in many of the structural models derived during the low-level analysis. In addition, given many of the questions that arose after the informal analysis, the formal technique provided a method for understanding certain properties of the code.

4.6.2 Tools

To facilitate the investigations described in this paper, we have constructed a number of tools that support formal specification construction, specification editing, call graph derivation, and basic theorem proving [13]. The availability of these tools greatly facilitated the analysis process both during the informal and the formal phases of analysis. However, while the tools were invaluable, they need to mature in regards to the functionality that they provide, especially in regards to user interface concerns. In addition, the use of other existing tools that replace, for instance, the `grep` tool, are needed to improve the ability to effectively analyze software source code.

5 Related Work

Previously, formal approaches to reverse engineering have used the semantics of the weakest precondition predicate transformer *wp* as the underlying formalism for the respective techniques. 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) [14]. 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 [15] (Restructuring, Maintenance, Validation and Documentation of Software Systems) is an Esprit II project whose objective is to improve applications by making them more maintainable through the use of reverse engineering techniques. The approach used to reverse engineer COBOL involves the development of general guidelines for the process of deriving objects and specifications from program code as well as providing a framework for formally reasoning about objects [16].

The "Loop ANalysis Tool for Recognizing Natural concepts" or LANTeRN [17] is an approach that uses a

```

extern void end_cmdxlt(int n) {
if (params->cmdcntl != 0) {
    inform_user(1);
    comm_tbl_ptr->alloc[comm_index].xlt_pid = 0;
    xlt_dir->new = 0;
    if (smclose("directive") == DTS_ERROR) {
        fprintf(stderr,
            "translate: closing directives shared memory failed\n");
    }
}
/* AutoSpec:
"(((n.V = _param0) /\ (coset(params).cmdcntl.V != 0)) /\
(coset(comm_tbl_ptr).alloc[comm_index].V = 0)) /\
(coset(xlt_dir).new.V = 0)) */
if (dereg_appl("SFOC CMD", "com_ws", SHM_ALLOC) == RES_ERROR) {
    fprintf(stderr,
        "translate: deregistration with SMC failed: %s\n",
        smc_errlist[smc_errno]);
}
/* AutoSpec:
"(((n.V = _param0) /\ (coset(params).cmdcntl.V != 0)) /\
(coset(comm_tbl_ptr).alloc[comm_index].V = 0)) /\ (coset(xlt_dir).new.V = 0)) */
if (master_detach_proj(-1) == -1) {
    fprintf(stderr,
        "translate: cannot detach from masterfile: %s\n",
        master_strerror(master_errno));
}
/* AutoSpec:
"(((n.V = _param0) /\ (coset(params).cmdcntl.V != 0)) /\
(coset(comm_tbl_ptr).alloc[comm_index].V = 0)) /\ (coset(xlt_dir).new.V = 0)) */
}
/* AutoSpec:
"(((n.V = _param0) /\ (coset(params).cmdcntl.V != 0)) /\
(coset(comm_tbl_ptr).alloc[comm_index].V = 0)) /\ (coset(xlt_dir).new.V = 0)) \/\
(n.V = _param0) /\ (!(coset(params).cmdcntl.V != 0)))" */

exit(n);
/* AutoSpec: false */

return;
}

```

Figure 7. Annotated source code for end_cmdxlt

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 are in the form of *basic events* and *augmentation events*.

In the REDO and *Maintainer's Assistant* approaches, the applied formalisms are based on the semantics of the *weakest precondition* predicate transformer *wp*. Some differences in applying *wp* and *sp* are that *wp* is a backward rule for program semantics and assumes a total correctness model of execution. However, the total correctness interpretation has no forward rule (i.e., no *strongest total postcondition stp* [8]). By using a partial correctness model of execution, both a forward rule (*sp*) and backward rule (*wlp*) can be used to verify and refine formal specifications generated by program understanding and reverse engineering tasks. The main difference between the two approaches is that the strongest postcondition predicate transformer can be applied to code to derive formal specifications, whereas the weakest precondition predicate transformer can only be

used as a guideline for constructing formal specifications. That is, *wp* cannot be used to derive postconditions for program code.

6 Conclusions and Future Investigations

Many different approaches for reverse engineering have been suggested in the literature [18, 19, 20, 21, 22]. The availability of such approaches provides the opportunity to study the effect of their combination. In our previous investigations, we have developed a formal approach for reverse engineering [4, 5, 6]. When applied as a standalone technique, the complexity of this approach is unmanageable. However, as demonstrated in this paper, when applied alongside simple structured approaches, the approach can be used to obtain meaningful results.

In order to further study the impact of integrating our formal approach with more informal approaches, we are investigating the relationship between structured abstraction and plan-based techniques and how they might benefit from the semantic information that is recoverable via our technique. In addition, we are also investigating how the approach can be used to support other activities such as soft-

ware reuse [23]. Finally, we are continuing to improve the tool suite that we have constructed to support the activities described in this paper [13].

Acknowledgments

The authors would like to thank Ron Slusser and John Kelly at the Jet Propulsion Laboratory for their assistance throughout these investigations. The authors would also like to thank the anonymous reviewers who provided invaluable comments on an earlier version of this paper.

References

- [1] Report by the Inquiry Board. ARIANE 5 Flight 501 Failure. Technical report, European Space Agency, 1996. J.L. Lions, Chairman of the Board.
- [2] B. H. C. Cheng. Applying formal methods in automated software engineering. *Journal of Computer and Software Engineering*, 2(2):137–164, 1994.
- [3] Elliot J. Chikofsky and James H. Cross. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, January 1990.
- [4] Gerald C. Gannod and Betty H. C. Cheng. Strongest Post-condition as the Formal Basis for Reverse Engineering. *Journal of Automated Software Engineering*, 3(1/2):139–164, June 1996. A preliminary version appeared in the *Proceedings for the IEEE Second Working Conference on Reverse Engineering*, July 1995.
- [5] Gerald C. Gannod and Betty H. C. Cheng. Using Informal and Formal Methods for the Reverse Engineering of C Programs. In *Proceedings of the 1996 International Conference on Software Maintenance*, pages 265–274. IEEE, 1996. Also appears in the Proceedings for the Third IEEE Working Conference on Reverse Engineering.
- [6] Gerald C. Gannod and Betty H. C. Cheng. A Formal Automated Approach for Reverse Engineering Programs with Pointers. In *Proceedings of the Twelfth IEEE International Automated Software Engineering Conference*, pages 219–226. IEEE, 1997.
- [7] Gerald C. Gannod and Betty H. C. Cheng. A Specification Matching Based Approach to Reverse Engineering. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 389–398. ACM, 1999.
- [8] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.
- [9] E. Yourdon and L Constantine. *Structured Analysis and Design: Fundamentals Discipline of Computer Programs and System Design*. Yourdon Press, 1978.
- [10] Rick Covington, editor. *Formal Methods Specification and Verification Guidebook for Software and Computer Systems; Volume 1: Planning and Technology Insertion*, volume NASA-GB-002-95. National Aeronautics and Space Administration, July 1995.
- [11] Judith Crow, editor. *Formal Methods Specification and Analysis Guidebook for the Verification of Software and Computer Systems; Volume 2: A Practitioner's Companion*, volume NASA-GB-001-97. National Aeronautics and Space Administration, May 1997.
- [12] Gerald C. Gannod. *Integrating Informal and Formal Techniques to Reverse Engineer Imperative Programs*. PhD thesis, Michigan State University, 1998.
- [13] Gerald C. Gannod and Betty H. C. Cheng. A suite of tools for facilitating reverse engineering using formal methods. Technical Report ASUCSE-TR99-02, Arizona State University, May 1999.
- [14] M. Ward, F.W. Calliss, and M. Munro. The Maintainer's Assistant. In *Proceedings for the Conference on Software Maintenance*. IEEE, 1989.
- [15] K. Lano and P.T. Breuer. From Programs to Z Specifications. In John E. Nicholls, editor, *Z User Workshop*, pages 46–70. Springer-Verlag, 1989.
- [16] H.P. Haughton and K. Lano. Objects Revisited. In *Proceedings for the Conference on Software Maintenance*, pages 152–161. IEEE, 1991.
- [17] S. K. Abd-El-Hafiz and V. R. Basili. Documenting Programs Using a Library of Tree Structured Plans. In *Proceedings of the Conference on Software Maintenance*, pages 152–161, 1993.
- [18] Alex Quilici. A Memory-Based Approach to Recognizing Program Plans. *Communications of the ACM*, 37(5):84–93, May 1994.
- [19] J.P. Bowen, P.T. Breuer, and K. Lano. The REDO Project: Final Report. Technical Report PRG-TR-23-91, Oxford University, 1991.
- [20] Martin Ward. Abstracting a Specification from Code. *Journal of Software Maintenance: Research and Practice*, 5:101–122, 1993.
- [21] Ira D. Baxter and Michael Mehlich. Reverse Engineering is Reverse Forward Engineering. In *Proceedings of the Fourth IEEE Working Conference on Reverse Engineering*. IEEE, October 1997.
- [22] Scott R. Tilley, Kenney Wong, Margaret-Anne Storey, and Hausi A. Müller. Programmable Reverse Engineering. *The International Journal of Software Engineering and Knowledge Engineering*, 4(4):501–520, 1994.
- [23] Gerald C. Gannod, Yonghao Chen, and Betty H. C. Cheng. An Automated Approach to Supporting Software Reuse via Reverse Engineering. In *Proceedings of the 13th Automated Software Engineering Conference*. IEEE, 1998.