

A Suite of Tools for Facilitating Reverse Engineering Using Formal Methods*

Gerald C. Gannod^{†‡}
Computer Science and Engineering
Arizona State University
Box 875406
Tempe, AZ 85287-5406
E-mail: gannod@asu.edu

Betty H. C. Cheng
Computer Science and Engineering
Michigan State University
3115 Engineering Building
East Lansing, MI 48824-1226
E-mail: chengb@cse.msu.edu

Abstract

As a program evolves, it becomes increasingly difficult to understand and reason about changes in source code. Eventually, if enough changes are made without a corresponding modification of the software documentation, reverse engineering and design recovery techniques must be used in order to understand the current behavior of a system. In our previous investigations, we described a formal technique for reverse engineering. One of the benefits of formal techniques is that they are amenable to automated processing. In this paper we describe an integrated suite of tools that we have developed to support reverse engineering and analysis of C programs.

1 Introduction

It is well-known that the maintenance of software is one of the most costly aspects of software development [1]. As a program evolves, it becomes increasingly difficult to understand and reason about changes to source code. Eventually, if several changes to software are made without a corresponding modification of the software documentation, *reverse engineering* and design recovery techniques must be used in order to understand the current behavior of a system. Software reverse engineering is defined as the process of analyzing a subject system in order to identify components and component interrelationships and to create representations of that system in other forms or at other levels of abstraction [2]. Several approaches to reverse engineering and design recovery have been suggested and include

techniques for deriving structural abstractions [3] and identifying plans embedded in code [4].

In our previous investigations, we described a formal technique for reverse engineering that involves two phases [5, 6, 7]. The first phase constructs low-level, *as-built* specifications from program code [5]. The specifications recovered from this phase are considered to be *as-built* since they describe a system as it was implemented rather than how the system was designed. The second phase of our investigations into reverse engineering involves the derivation of high-level abstractions from *as-built* specifications [7]. By constructing high-level abstractions, several activities are facilitated, including high-level reasoning, program understanding, and software reuse [8].

One of the attractive properties of formal methods is that formal languages with well-defined syntax and semantics facilitate the construction and use of automated support tools. In addition, since manual application of formal techniques can be prone to errors, automated support is desirable. The size of non-trivial programs ranges from the tens of thousands to perhaps even millions of lines of code. Given the combined context of formal methods and program analysis, the construction of automated support tools is well-motivated. In this paper, we describe the development of several tools that have been designed to support a formal approach for reverse engineering. Specifically, the suite consists of four tools:

- AUTOSPEC: supports the construction of specifications using the semantics of the strongest postcondition predicate transformer
- SPECGEN: derives abstract specifications from *as-built* specifications
- SPECEDIT: a specification editor with a graphical user interface (GUI) front-end that supports the construction of syntactically correct specifications

*This work has been supported in part by the National Science Foundation grants CCR-9633391, CDA-9617310, 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.

- **TPROVER**: a tableau theorem prover that verifies the consistency of specifications that are modified by a user

The remainder of this paper is organized as follows. Section 2 describes background material regarding the software reverse engineering approach supported by the toolset. Each of the tools, including a class library for supporting the interchange of specifications between the tools, is presented in Section 3. Section 4 briefly describes results from applying the toolset to a software system. Related work is presented in Section 5, 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 [9], where \mathbb{IF} represents the n alternative conditional statement $\text{if } B_1 \rightarrow S_1; \dots \parallel B_n \rightarrow S_n; \text{fi}$; In the conditional statement, $B_i \rightarrow S_i$ represents a guarded command such that S_i is only executed if logical expression (guard) B_i is true. DO represents the loop statement “ $\text{do } B \rightarrow S \text{ od}$ ” where S is executed iteratively until guard B is false.

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

Table 1. Strongest Postcondition Semantics for the Dijkstra Guarded Command Language

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 $sp(\mathbb{IF}, Q)$ states that after execution of the if-fi 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 \mathbb{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 [5]. 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 [10], as well as programs that contain the use of pointers and pointer operations [6]. 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 [10].

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 $I \preceq I'$. 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 Tool Suite

The reverse engineering approach presented in Section 2 describes several well-defined translations and transformations that, although tedious and error prone when applied manually, can be easily applied when using automated support tools. In addition, given the mere size of existing programs, the tools facilitate the use of a formal reverse engineering technique for analyzing non-trivial systems in a repeatable manner.

Figure 1 contains a data flow diagram that shows the inter-relationships that exist between the various tools in the suite that we have developed. In this notation, circles are used to represent processes, parallel lines represent data stores, rectangles represent actors, and arrows represent flow of data. Figure 1 shows the relationship between the tools in the suite and external tools that we have used to aid in the analysis of software during the reverse engineering process. In the diagram, externally developed tools are shown as actors.

3.1 Overview

The primary objective of using the tool suite described in this paper is to apply the techniques described in Section 2. That is, to parse and analyze program code, and to generate

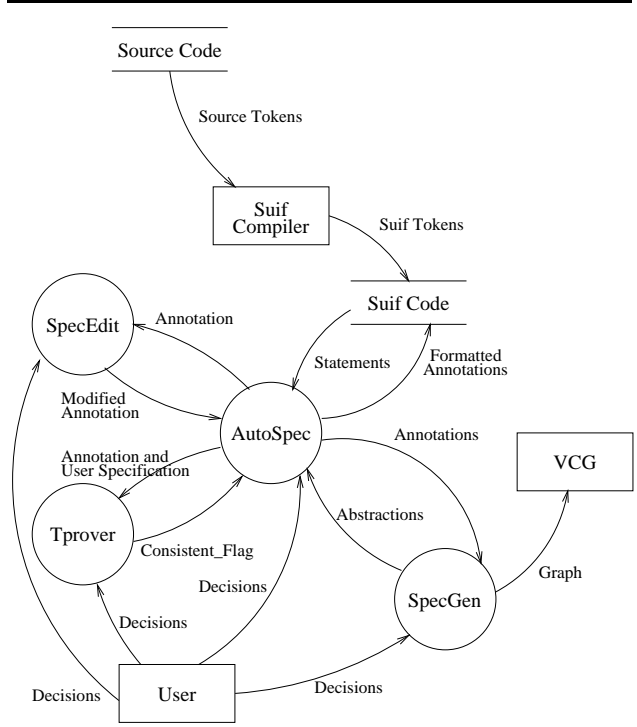


Figure 1. Tool Suite

formal specifications that can be subsequently manipulated either via user input or by applying the abstraction technique described in Section 2.2. As a result, specifications that formally describe behavior of existing program code can be annotated to source code and eventually analyzed for a wide variety of applications including the population of specification libraries in the context of software reuse [8].

The overall process for using the tools begins with a pre-processing step whereby the SUIF Compiler [11] is used to generate an intermediate format based on the C programming language. The AUTOSPEC system then takes the SUIF generated code, and based on user input, generates source code annotations based on the strongest postcondition. During the analysis, the user can provide assistance to the AUTOSPEC system via the use of the SPECEDIT specification editor. Primarily, this assistance comes in the form of simplifications to the annotations. When a user makes modifications to annotations the TPROVER theorem prover is used to verify that the modifications are consistent with the specification generated by the AUTOSPEC system. Finally, after as-built specifications have been constructed using the AUTOSPEC tool, the SPECGEN tool can be used to generate abstractions that can be visualized as Hasse diagrams using the Visualization of Compiler Graphs (VCG) tool [12].

3.2 AUTOSPEC

The (Semi-)Automated *Specification* and generation system, or AUTOSPEC, was originally developed in order to demonstrate the feasibility of our initial investigations into reverse engineering [13]. Written in an object-oriented variant of Prolog, the original prototype facilitated the application of user-directed heuristics to construct predicate logic specifications from Dijkstra guarded command language programs [13]. Since then, several different variations and refinements of the AUTOSPEC system have been developed, each with the intention of investigating new dimensions of our research investigations and validating the amenability of our techniques to automation, including the analysis of programs using strongest postcondition semantics [5], and the analysis of pointer semantics [6]. In this section, we describe the most recent version of AUTOSPEC that has been refined from previous versions in order to handle more complex languages and a wider variety of programs.

3.2.1 Design

The high-level design of the AUTOSPEC system is shown in Figure 2. The AUTOSPEC system interacts with three different environmental entities: the *User*, a specification editor called SPECEDIT, and a theorem prover called TPROVER. The specification editor and theorem prover are described in Sections 3.4 and 3.5, respectively. The AUTOSPEC system reads a file, and based on various interactions with the user and external tools, generates formal specifications based on the use of strongest postcondition, and annotates the original source code with those specifications. Direct user interaction with the AUTOSPEC system comes in the form of decisions about how a source file analysis should proceed. The user also interacts with the AUTOSPEC system indirectly via the use of the SPECEDIT and TPROVER tools.

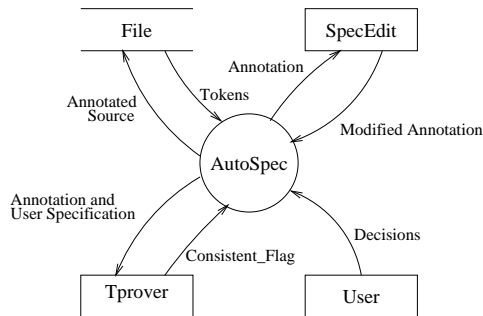


Figure 2. Level 0 AUTO SPEC Model

The design of the AUTOSPEC system follows the same general architecture of many compiler and static analysis systems [14]. That is, the design of the AUTOSPEC system consists of a *parsing* component that reads a source file and creates an *abstract syntax tree*, an *analysis component*

that is used to construct specifications from the program, and an *output* component that writes the results to an appropriate output file. Figure 3 contains the level 1 data flow diagram of the AUTOSPEC system. The *Parse*, *SP*, and *Output* processes correspond to the *parsing*, *analysis* and *output* components, respectively. In addition to the standard compiler-oriented components, the AUTOSPEC system has a component for interacting with the user (i.e., a *user interface*). Data in the AUTOSPEC system is centered primarily around the *Abstract Syntax Tree* and *program statements*. In addition to statements, flow of data in the AUTOSPEC system consists of *specifications* and *annotations*, where annotations are specifications that are tied to specific program statements.

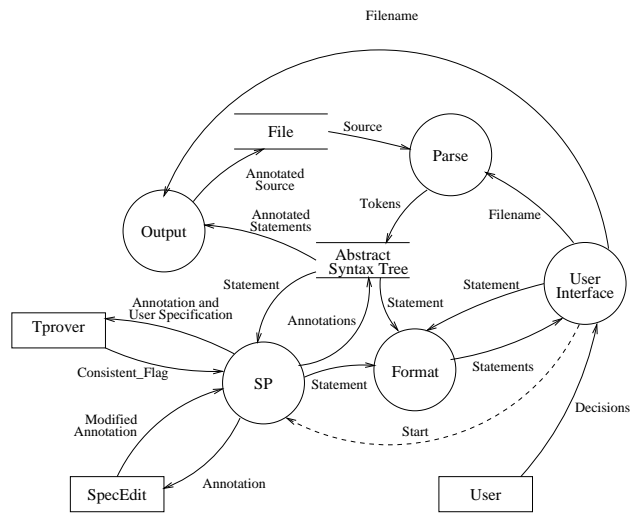


Figure 3. Level 1 Data Flow Diagram of AUTO SPEC

The primary component of the AUTOSPEC system is the analysis, or *SP* component. The *SP* component consists of several procedures that are responsible for constructing specifications from programming constructs. The formal specifications that are generated by the *SP* component correspond to the semantic definitions summarized in Section 2. In addition to constructing specifications, the *SP* component is responsible for launching the TPROVER and SPECEDIT applications when user input is required.

3.2.2 Implementation

The AUTOSPEC system was developed primarily as a means for supporting the analysis of source code using the strongest postcondition. The AUTOSPEC system was originally developed to support the Dijkstra guarded command language. Since that language is used primarily for theoretical development and analysis, it was decided that in order to show the applicability of our approaches to real systems,

support for a commonly-used language must be included in the subsequent implementations of AUTOSPEC. The remainder of this section describes the implementation of a C variant of the AUTOSPEC system.

SUIF Compiler. The Stanford University Intermediate Format (SUIF) library is a suite of routines that were developed to support research for optimizing and parallelizing compilers [11]. Developed by the Stanford University Compiler Group, the objective of the SUIF compiler is to provide an extensible support system for a wide variety of compiler-oriented investigations [11].

Several factors motivated the use of the SUIF compiler suite of tools. First, the SUIF compiler provides a library of routines for parsing and accessing source code information via the use of an abstract syntax tree representation of SUIF code. Second, by using the SUIF compiler, we are able to take advantage of several built-in features including the use of *annotations* to document programs, and source code iterators for traversing the abstract syntax trees. In addition, the SUIF library has extensive support for symbol tables. Finally, by using the SUIF compiler we are able to leverage the experience of an established community of users.

The SUIF library focuses on the organization of input files into abstract syntax trees based on the structure of the C programming language. SUIF also supports the analysis of Fortran programs and contains support for programming constructs, such as loops, conditionals, and assignments. These constructs are translated into an intermediate format that is decomposed into semantically equivalent SUIF constructs. In addition, the SUIF libraries support the use of symbol tables as well as convenience functions that can be used to traverse source code.

SUIF also supports the use of source code annotations. In the context of the AUTOSPEC system, these SUIF code annotations are used to attach strongest postcondition specifications to particular programming statements. After analyzing the source code, these annotations can be translated into comments and added as annotations to the original source code.

In the AUTOSPEC system we use the SUIF tools as follows. First, the SUIF compiler *scc* is used to generate a SUIF intermediate file from the original source code. Second, the SUIF library of tools is used to manipulate the SUIF intermediate file. Third, the symbol table and source code traversal functions are used to access the abstract syntax trees for the input source code in order to generate formal specifications based on the semantics of the strongest postcondition. Fourth, the annotation facility is used to associate formal specifications with specific source statements. Finally, a tool called *s2c* is used to translate the SUIF source code into equivalent C source code with annotations.

Interface. The interface for the AUTOSPEC system was constructed using the Tcl/Tk language [15]. In addition, we used the C++ language to provide the interconnection between Tcl/Tk and the SUIF libraries. The interface is organized to provide convenient access to and manipulation of the input source code. The main window, as shown in Figure 4, is used to display the source code for a user-selected procedure.

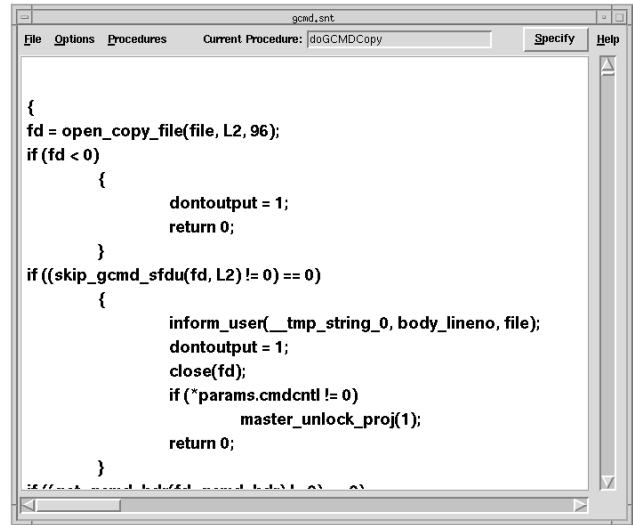


Figure 4. AUTOSPEC Main Window

The analysis of the source code includes three distinct phases. The first phase focuses on allowing a user to select *analysis breakpoints*, thus providing the user with a means for indicating where they prefer to provide input to the analysis. Selecting (i.e., “double-clicking”) a particular programming statement indicates that the analysis of the current procedure should be interrupted just before processing of the selected statement. Figure 5 shows the interface with a selected statement shown in italics.

The second phase of the analysis is the *specification phase*. In this phase, AUTOSPEC constructs a formal specification of the procedure by using the strongest postcondition semantics. As each analysis breakpoint is encountered, AUTOSPEC pauses, as depicted in Figure 6, and allows the user to modify the current annotation (i.e., the precondition for the next statement). The modification of the specification is performed by using the SPECEDIT specification editor, which is described in Section 3.4. Using the SPECEDIT system, the user modifies the precondition and can optionally launch a theorem prover that can be used to check the consistency between the user-defined specification and the system-defined specification. In Section 3.5 we describe the design and implementation of the TPROVER theorem prover.

The final phase of the analysis is the *post-specification*

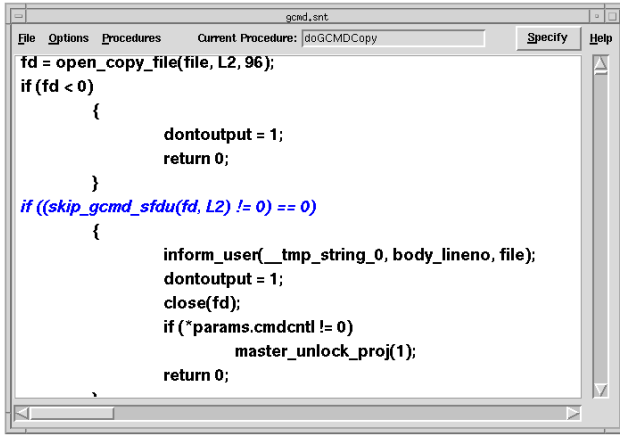


Figure 5. AUTOSPEC A Selection in the Main Window

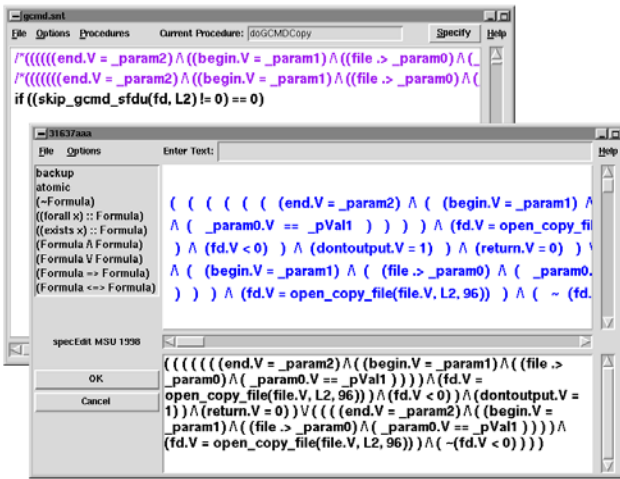


Figure 6. Launching SPECEDIT from AUTOSPEC

phase. During this phase, the user is free to select and modify any annotation that is displayed in the main window. In addition, we are incorporating a feature that will allow a user to modify a specification, replay the analysis, and incorporate the changes into the analysis.

3.3 SPECGEN

Once the specifications have been constructed using the AUTOSPEC system, the specifications need to be generalized into higher-levels of abstraction. The *Specification Generalization* system, or SPECGEN, was developed in order to aid in the construction of abstract specifications from as-built specifications.

3.3.1 Design

The high-level design for the SPECGEN system is shown in Figure 7. The SPECGEN system interacts with two environmental entities: the *User*, and the visualization tool *VCG* [12]. Upon launching, the SPECGEN system reads a specification from a file (*SpecFile* in Figure 7), and based on user decisions, constructs abstractions that can be visualized in the form of a partial-order diagram. The partial-order diagram, depicted in Figure 7 as the *GraphFile*, is then displayed using the *VCG* tool.

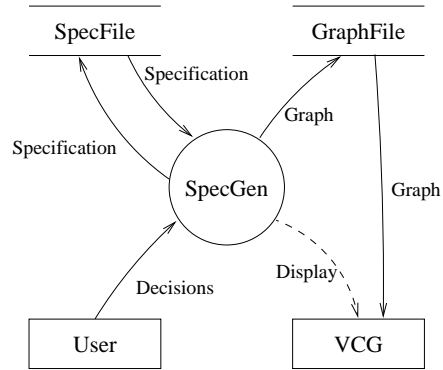


Figure 7. Level 0 SPECGEN Model

Figure 8 shows the level 1 data flow diagram for the SPECGEN system. The internal structure of the SPECGEN system consists of three major components: a *parser*, a *user interface*, and an *abstraction engine*. The parser is a standard *Lex* and *Yacc* [16] parser that has been constructed for checking the syntax of first-order logic specifications. The user interface is the primary mechanism for mediating interaction between the user and the abstraction engine. The abstraction engine is responsible for constructing high-level specification generalizations based on the techniques described in Section 2.2.

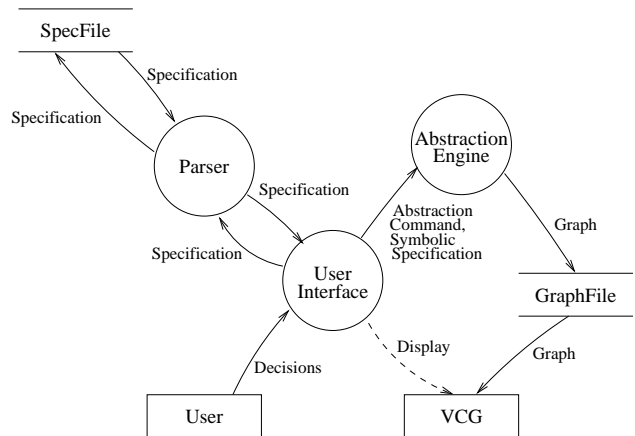


Figure 8. Level 1 SPECGEN Model

3.3.2 Implementation

The implementation of the SPECGEN system has two major components. The first component is the *abstraction engine*, which is responsible for deriving symbolic abstractions from an input specification. The second component is the *user interface*, which provides a mechanism for facilitating user-driven specification generalization. In this section we describe each of these components in detail.

Prolog. The abstraction engine was written using SWI-Prolog [17] interconnected with a number of C routines. The primary motivation for using Prolog was to take advantage its inferencing and backtracking capabilities. The small number of routines (approximately 20) and their size (10-15 lines each), easily justified our choice of language in this case.

The Prolog routines are primarily responsible for deriving partial-orderings of specifications as well as pruning and expanding the orderings based on user input. By using a library of C routines that support integration of C and Prolog, it became a straightforward process to build the system along with a graphical user interface that facilitated visualization and manipulation of the specification generalizations by a user.

Interface. The user interface for SPECGEN includes two different components. The first component is the direct user manipulation component that allows a user to load, manipulate, and analyze specifications using the abstraction engine. This component, written in Tcl/Tk [15], facilitates specification generalization by allowing a user to select specification components of interest, to exclude or *mask* out certain specification components, and to generate all possible abstractions. For instance, Figure 9(a) depicts the SPECGEN system with the top portion containing the conjuncts of a conjunctive normal form expression. The buttons labeled “Delete Conjuncts”, “Preserve Conjuncts”, “Focus”, and “Generate All” allow a user to derive different levels of abstraction with differing levels of detail from a specification. Figure 9(b), shows the results of an analysis, where a specification has been generalized using SPECGEN and the resulting partial-ordering visualized using a tool called VCG.

The second component of the user interface is the visualization component. In this component we take advantage of an existing system called VCG [12], a system that supports the visualization of graphs. VCG provides many functions for graph layout and placement, issues that are well beyond the scope of our investigations. Currently, the SPECGEN system produces a VCG formatted input file and generates an event that is trapped by the VCG tool. As a consequence, each step of the abstraction process is visually represented and displayed to the user with a Hasse diagram. One of

the shortcomings of using VCG, however, is that it lacks a mechanism for providing feedback to external systems such as the main SPECGEN system. As such, as part of our future investigations, we plan on extending the functionality of SPECGEN to incorporate an internal visualization facility.

3.4 SPECEDIT

One of the advantages of using formal methods is that their notations are mathematically based. As such, these notations are amenable to automated processing and reasoning. One of the tools that can be constructed to support any particular formal specification language is a syntactic checker or parser. An associated tool along the same lines of a checker is a syntactic editor [18]. In this section we describe the design and implementation of a specification editor that we have developed in order to facilitate user interaction with the AUTOSPEC and SPECGEN systems.

3.4.1 Design

Figure 10 shows the high-level data flow diagram model for the SPECEDIT system. The SPECEDIT system, based on a configurable library for specification editors [18], interacts primarily with the user to construct or modify specifications, and allows the user to save the specifications to files for later modification or for incorporation into other tools that use first-order logic as an input language.

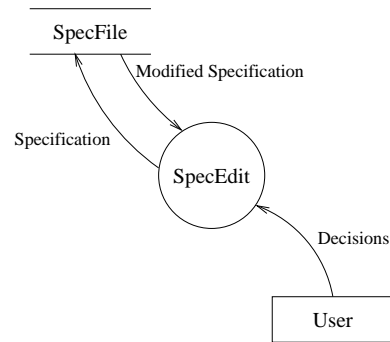
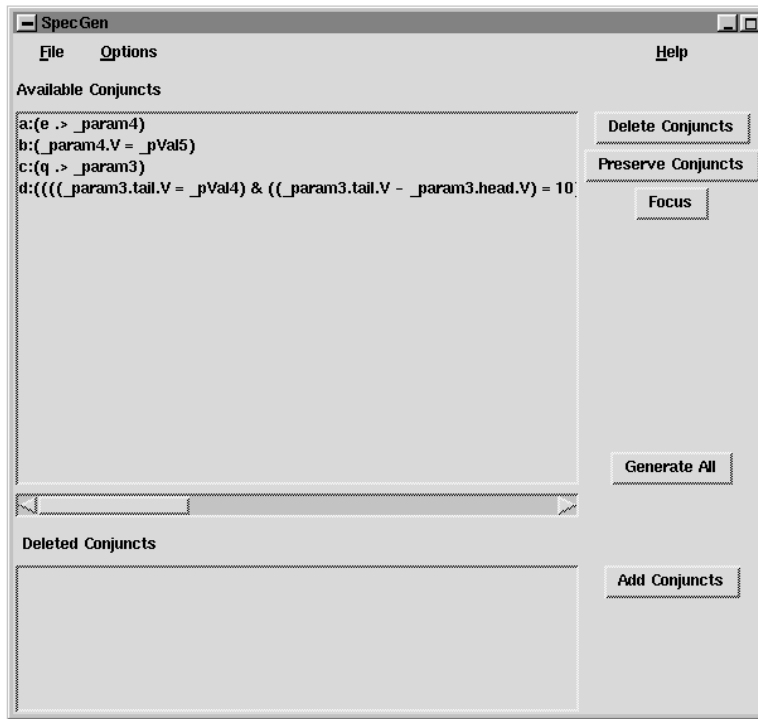
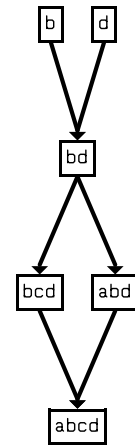


Figure 10. Level 0 SPECEDIT Model

Figure 11 contains the data flow diagram for the SPECEDIT system. The internal design of the SPECEDIT system has two primary components: a *parser* and a *user interface*. The user interface facilitates the construction of a syntactically correct specification in two ways. First, the user interface has a graphical interface that allows users to construct specifications using a point and click method. The user interface also has a text-based interface that allows a user to type in a specification. The parsing component is responsible for two different activities. First, the parser is responsible for checking the syntax of pre-existing specifications that are contained in input files. The parser is also



(a) SpecGen



(b) Focus Graph

Figure 9. SPECGEN Interface and Output

responsible for checking the syntax of user modifications that are made using the text-based interface for the system.

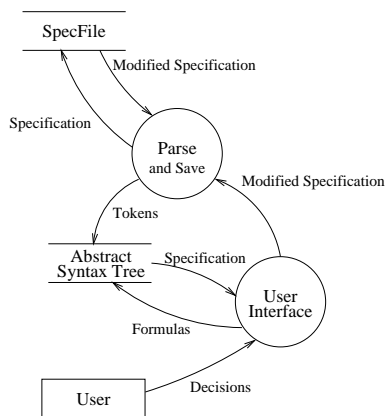


Figure 11. Level 1 SPECEDIT Model

3.4.2 Implementation

The *Specification Editing* system, or SPECEDIT, was developed in order to facilitate specification modification during the analysis phase of the AUTOSPEC system. The main objective in constructing the SPECEDIT system was to

provide a way of ensuring syntactic correctness during the modification of a specification. This correctness is ensured in two ways: by construction or by verification. Correctness by construction is facilitated by providing a mechanism where a user clicks on various syntactic elements and replaces them with valid substitutions. For instance, Figure 12, shows the conjunctive formula " $p(x) \wedge \text{Formula}$ ". The italicized font for $p(x)$ indicates that the term has been selected using a mouse click. By double-clicking on the "Formula" conjunct in the upper window of SPECEDIT, the user can choose to substitute the conjunct with either an atomic formula, a conjunction, disjunction, implication, or quantification. Since substitutions can only be made with valid substitutions, the final specification is syntactically correct by construction.

SPECEDIT can also verify correctness using a syntactic checker or parser. The user may enter a new specification in the lower window of the SPECEDIT interface. By clicking on the "OK" button, a user directs the SPECEDIT system to run the syntactic checker on the specification in the lower window. If the specification is syntactically correct, the specification is loaded into the upper window and the user is free to modify the specification in either window.

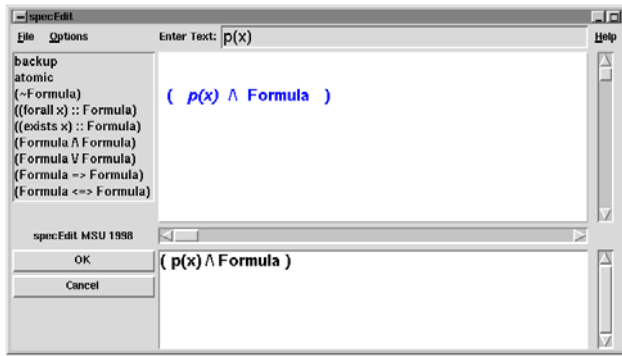


Figure 12. SPECEDIT

3.5 Theorem Prover

Many of the interactions between a user and the AUTOSPEC system involves the modification of a specification by a user and the incorporation of the modified specifications into the current analysis or program annotation. In order to verify that the specification modifications made by a user are logically consistent with the system-generated specifications, we have constructed a simple theorem prover. In this section we describe the design and implementation of the theorem prover TPROVER.

3.5.1 Design

Figure 13 depicts the high-level design of the TPROVER system. Except for file interactions and user guidance, the TPROVER system is entirely self-contained. The TPROVER system takes as input a source file containing a first-order logic specification to be proved. Using guidance provided by a user for reasoning about quantified expressions, the TPROVER system determines whether or not the specification is valid.

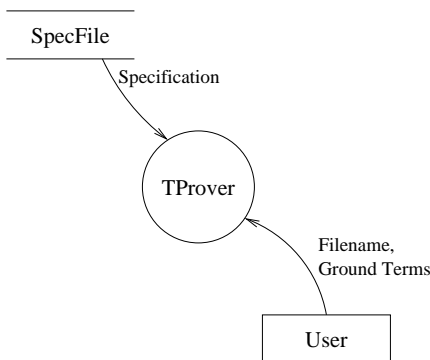


Figure 13. Level 0 TPROVER Model

Figure 14 shows the internal structure of the TPROVER system. The primary components of the system are: the *parse* component, the *prover* component, and the user *con-*

sult component. The *parse* component is a *Lex* and *Yacc* generated parser that is used to translate first-order logic specifications into an *abstract syntax tree*. The *prover* component uses the information in the abstract syntax tree to generate a proof tree based on the tableau proof technique. For certain proof rules in the tableau method, ground terms must be identified in order to continue processing. In these instances, the *consult* component is used to interact with the user in order to identify an appropriate ground term for the proof method.

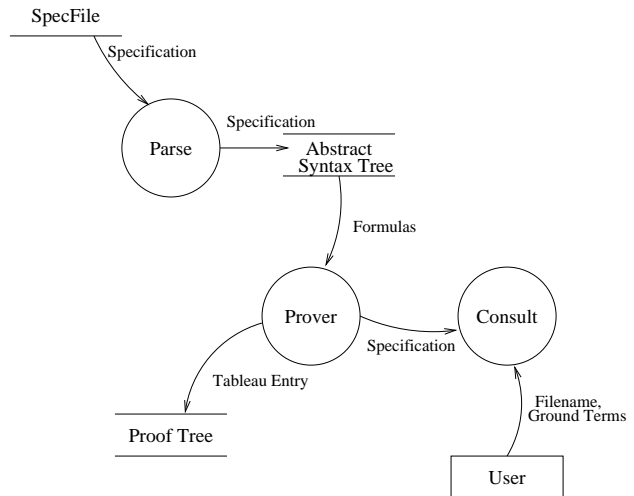


Figure 14. Level 1 TPROVER Model

3.5.2 Implementation

The main proof engine in the TPROVER system was constructed using C++. The proof engine construction was facilitated by the existence of the formula class library that we developed to support all the tools. The graphical user interface was constructed using Tcl/Tk interconnected with C++.

The TPROVER system takes as input the name of a file containing a logical expression. Based on the rules for constructing tableau proofs, the TPROVER system will generate a proof tree. For propositional logic and certain expressions of first order logic, the theorem prover is automatic. For the remaining classes of valid input, user direction is required. In the latter case, the TPROVER system acts as a tableau proof editor. In the former case, the TPROVER system acts as a theorem prover.

Figure 15 shows the main window of the TPROVER system. The upper sub-window is the main prover window. In this window the proof tree is constructed and displayed to the user. The lower sub-window is the proof information window. In this window the user is provided information about the current proof. Specifically, during times of user interaction, the proof information window contains

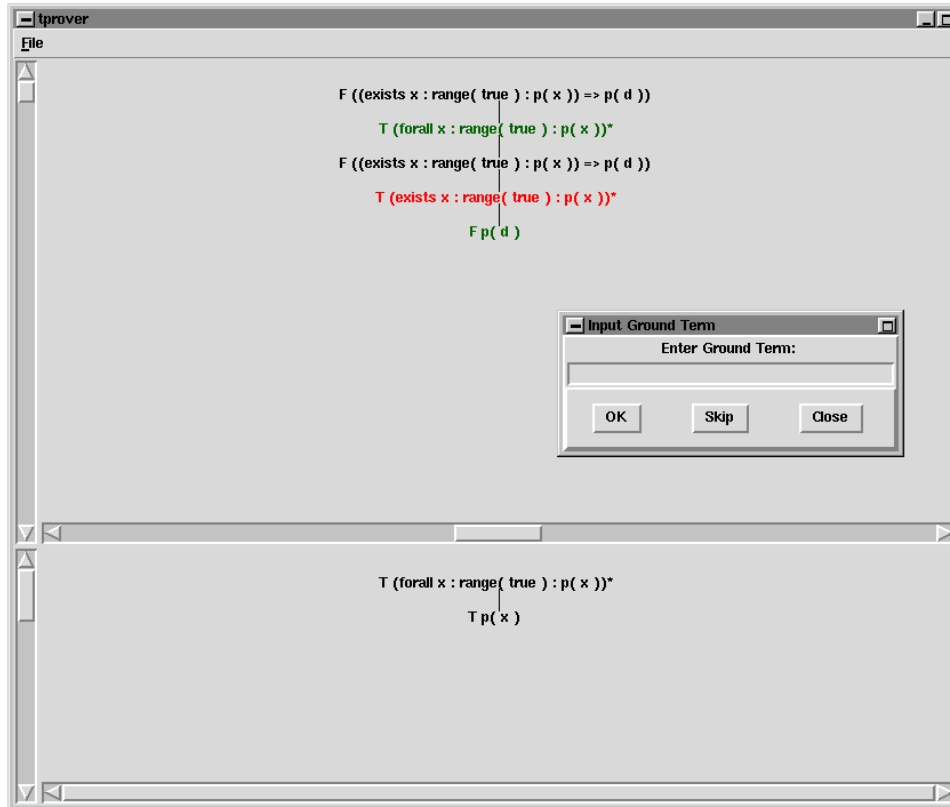


Figure 15. Example TPROVER Session

data about the current entry in the proof tree. To aid in proof comprehension, the vertices in the proof trees are displayed with different colors, each indicating the different states that an entry in a proof tableau may take including *open* (green), *closed* (black), *pending* (red), and *contradictory* (blue).

4 Case Study Results

Previously, we performed a case study whereby we applied the use of the tools described in this paper [19]. In this section we describe some of the results and the impact of tools upon the case study.

4.1 Process

The process used to analyze software using the formal analysis tools that we have developed is a three step method that involves the construction of high-level models that depict overall structure, construction of low-level models that identify calling relationships, and, finally, construction of formal models using formal techniques.

Figure 16 shows a formal specification that was derived using the tools described in this paper. Due to space constraints, we refer the reader to the full discussion of its derivation which is contained elsewhere [19]. The software shown here is part of a larger system used for ground-based mission control for robotic spacecraft.

4.2 Lessons Learned

Several lessons about our reverse engineering approach were learned while performing the case study [19]. This section summarizes these lessons.

4.2.1 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.

4.2.2 Tools

The availability of the tools described in this paper greatly facilitated the analysis process both during the informal and the formal phases of analysis. Given the amount

```

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 16. Annotated source code for end-message_subroutine

of “bookkeeping” required to derive the formal specifications, the tools were invaluable. However, the tools 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. While it would be advantageous to evaluate user application of these tools, we find that the potential audience for the techniques is rather limited, and thus we have been unable to do widespread end-user evaluations.

5 Related Work

Approaches to reverse engineering focus on the construction of specifications, both informal and formal, and are based on the construction of structural abstractions [3], the identification of *plans* [4], use of formal methods [20, 21], and transformation of programs into specifications [22]. Rigi [3] is a tool that has been used to derive structural abstractions (e.g., design diagrams) from program code. Such abstractions provide a view of programs that is complementary to the by-products of the tools described in this paper. As such, they can be used as a cognitive guide to a user during the program understanding activity [10].

Baxter and Mehlich [22] suggest an approach to reverse engineering using “backward transformation” where a series of transformations (semantic preserving rewrite rules), similar to those used in forward transformation, are used in an inverse manner. The use of a library is extensive in this approach where the contents of the library are semantic preserving transformations. Ward [21] also advocates a transformational approach while the tools from the REDO project [20] focused on translation and transformation of programs into specifications. In contrast, the AUTOSPEC tool is based primarily on the application of the strongest postcondition predicate transformer. As such, specifications are constructed based on the use of a relatively small number of rules.

Several theorem proving tools have been constructed and subsequently described in the literature [23]. By comparison, the theorem prover described in this paper is much simpler, which is evidenced by its lack of support for sophisticated theorem proving technology such as decision procedures. However, the TPROVER tool provides a number of services via an API that facilitates its integration into several analysis tools, including AUTOSPEC. In addition, since the theorem proving requirements of the tool suite described in this paper are minimal, more powerful theorem provers are considered overkill.

6 Conclusions and Future Investigations

Given the size of existing programs, tool support for program analysis techniques is necessary to study problems of scale. From a maintenance perspective, the requirement is a pragmatic one since few programmers have the stamina nor the desire to manually analyze what can potentially be millions of lines of code. One of the benefits of formal methods is that the formal notations facilitate the construction and use of automated support tools.

In this paper we described a toolset for a reverse engineering approach that is based on the use of formal methods. These prototype tools further demonstrate the feasibility of constructing formal methods-based reverse engineering tools and have been used to analyze industrial software systems [24]. To further evaluate these tools, we are in

the process of developing a series of experiments that will facilitate the comparison of this toolset with other reverse engineering and design recovery tools [24]. Currently we are refining the tools described in this paper to increase the amount of support for user interaction. In addition, we are analyzing our theorem proving needs in order to determine if a more sophisticated theorem prover is needed as a replacement for the basic prover described in this paper. We are also developing Java implementations of the toolset in order to facilitate the use of the tools on a variety of platforms. Finally, we are developing an environment for integrating several reverse engineering techniques, both formal and informal, into a single framework in order to take advantage of the relative benefits of several approaches.

References

- [1] Roger S. Pressman. *Software Engineering A Practitioner's Approach*. McGraw-Hill, fourth edition, 1997.
- [2] Elliot J. Chikofsky and James H. Cross. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, January 1990.
- [3] K. Wong, S. R. Tilley, H. A. Müller, and M.-A. D. Storey. Structural redocumentation: A case study. *IEEE Software*, pages 46–54, January 1995.
- [4] Alex Quilici. A Memory-Based Approach to Recognizing Program Plans. *Communications of the ACM*, 37(5):84–93, May 1994.
- [5] Gerald C. Gannod and Betty H. C. Cheng. Strongest Postcondition 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.
- [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] 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.
- [9] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.
- [10] 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.
- [11] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the suif compiler. *IEEE Computer*, December 1996.
- [12] G. Sander. Graph layout through the vcg tool. In *Proceedings of Graph Drawing, DIMACS International Workshop GD'94, Lecture Notes in Computer Science*, volume 894, pages 194–205. Springer-Verlag, 1995.
- [13] Betty H. C. Cheng and Gerald C. Gannod. Abstraction of Formal Specifications from Program Code. In *Proceedings for the IEEE 3rd International Conference on Tools for Artificial Intelligence*, pages 125–128. IEEE, 1991.
- [14] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.
- [15] John Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.
- [16] J.R. Levine, T. Mason, and D. Brown. *lex & yacc*. O'Reilly & Associates, 1992.
- [17] Jan Wielemaker. *SWI-Prolog 3.0 Reference Manual*. University of Amsterdam, July 1998.
- [18] Robert H. Bourdeau and Betty H. C. Cheng. An Object-Oriented Toolkit for Constructing Specification Editors. In *Proc. of the Computer Software and Applications Conference (COMPSAC)*, September 1992.
- [19] Gerald C. Gannod and Betty H. C. Cheng. A Formal Approach for Reverse Engineering: A Case Study. In *Proceedings of the 6th Working Conference on Reverse Engineering*. IEEE, October 1999.
- [20] J.P. Bowen, P.T. Breuer, and K. Lano. The REDO Project: Final Report. Technical Report PRG-TR-23-91, Oxford University, 1991.
- [21] Martin Ward. Abstracting a Specification from Code. *Journal of Software Maintenance: Research and Practice*, 5:101–122, 1993.
- [22] 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.
- [23] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.
- [24] Gerald C. Gannod and Betty H. C. Cheng. A Framework for Classifying and Comparing Software Reverse Engineering and Design Recovery Techniques. In *Proceedings of the 6th Working Conference on Reverse Engineering*. IEEE, October 1999.