Reverse Engineering: Resolving Conflicts between Expected and Actual Software Designs

Stephen B. Ornburn and Spencer Rugaber
College of Computing and Software Research Center
Georgia Institute of Technology
Atlanta, GA 30332-0280

Abstract

A real-time embedded system was the subject of a series of experiments in reverse engineering. These experiments employed a method of reverse engineering, called Synchronized Refinement, *that analyzes a program, describing its behavior in the vocabulary of the application domain and its structure in terms of design decisions. The results provide insight into the role of domain knowledge in this type of analysis together with the tools used in the detailed analysis of code. The experiments, which included the re-design of a component and the diagnosis of a critical software failure, showed how the real work of software maintenance is in resolving apparent inconsistencies between the expectations that have been derived from domain knowledge and the facts that have been uncovered by applying reverse engineering tools to the software.*

1 Introduction

Synchronized Refinement [4, 7], a method for reverse engineering, is a process by which a re-engineered design is constructed from an existing software system. Synchronized Refinement consists of two parallel activities: the synthesis of functional and nonfunctional behavioral descriptions and the code-level analysis of the program text. In the synthesis process, the software engineer "executes" design decisions that expand, combine and refine descriptions of the program's various functional and nonfunctional descriptions, producing a design which is expected to reflect the actual structure of the program text. These functional and nonfunctional descriptions constitute a conceptual model of the software. As part of comparing the program to the conceptual model, considerable code-level analysis is necessary. Code-level analysis includes the restructuring of the program text by detecting and then "undoing" design decisions. When a design decision is undone, the original code is rewritten in an abbreviated form, possibly in terms of pseudocode such as might be used for detailed program designs. As more and more design decisions are undone, the program description becomes smaller and more abstract.

Synchronized Refinement attempts to confirm an expected design by comparing it to the actual design recovered through code-level analysis. If the expectations are not confirmed, then they must be revised and the code-level analysis must be extended. Code-level analysis, including both the testing of expectations and code restructuring, is supported by simple queries against a database of information extracted from the program text. The design that emerges by resolving conflicts between the expected and recovered designs will still differ in structure from the recovered design and is a re-engineered design for the software system.

The processes for synthesizing an expected design and for code-level analysis feed into each other: the synthesis process establishes expectations about what should be found in the code, and the code-level analysis tests those expectations against the actual code. Generally, code-level analysis confirms the expectations and then fills in additional details. The payoff from a disciplined approach to reverse engineering comes, however, when code-level analysis fails to confirm the expectations; it is in coping with these inconsistencies that the software engineer comes face-to-face with the hard-to-understand portions of the software. If software engineers are to use reverse engineering techniques to redesign a software system or, more simply, to explain what a software system does and how it works, they must resolve inconsistences between the design they expect to find and the design

actually used. It is in this resolution process that the hard work of software maintenance gets done.

2 Design decisions

When an experienced software engineer looks at a program, he expects to see it built around an appropriate architecture. If on reading the code he does not see the expected landmarks, he must begin a more careful study to determine how the developers solved various design problems. This determination becomes more complex if the original developers did not address important design problems before they begin coding, keeping their options open so that they could more easily incorporate *ad hoc* solutions to unexpected design problems during implementation and later during maintenance.

We have observed that because of the flexibility multiprocessing affords, there is an especially strong temptation to use *ad hoc* solutions to design problems when developing real-time systems. For example, synchronization mechanisms can be used to carefully control the order in which tasks execute, thereby compensating for a designer's careless functional decomposition of the original problem.

2.1 Expected vs. actual designs

Chikofsky and Cross[2] define reverse engineering as "the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction." Typically, this means constructing a re-engineered architectural design of a system from its source code. To be comprehensible, the re-engineered design must improve on the actual design of the software. Of course, the re-engineered design will reflect the preferences and expectations of the software engineer doing the analysis.

Since the re-engineered and original designs often differ, the re-engineered design must be accompanied by a set of design decisions, some times complex and difficult to recognize, that transform it into the original design. Synchronized Refinement, one strategy for recognizing these decisions, requires that the software engineer hypothesize a conceptual model, an expected software design, and a set of design decisions linking them. This initial set of design decisions is then modified to account for the extra complexity of the actual code. Because of this extra complexity, much of the effort in understanding a software component is expended in resolving conflicts between it and the expected design. Once the conflicts have been resolved, the software engineer is in possession of two sets of design decisions, one linking the conceptual model to the actual code and the other to a re-engineered design. Furthermore, the re-engineered design can be use to reimplement the software.

In our experiments with real-time embedded software, there were several ways of resolving conflicts between the expected and actual design:

- modify the conceptual model because the program supports unexpected features or a variant of the expected behavior;
- modify the program text because a coding or design error has been uncovered and must be fixed before the analysis can be completed;
- reject a hypothesized design decision because the program supports the expected behavior but in an unanticipated way; and
- accept a hypothesized design decision while noting that evidence of the decision was obscured by other, irrelevant decisions.

When evaluating a hypothesized design or resolving design conflicts, the software engineer must often trace data and control flows in search of missing operations, data, and data structures. Because the expected landmarks usually associated with a particular design decision may not really be missing but merely hidden by other decisions, tracing data and control flows requires close, tool-supported analysis of the code.

Furthermore, in resolving design conflicts, the software engineer can backtrack and modify some of the previously hypothesized design decisions. This backtracking can relate both to the design decisions "undone" as part of the code analysis or to the ones "executed" to derive the expected design.

The top-down aspects of this process are similar to the psychological model of code comprehension described by Ruven Brooks[1]. While it is important that the reverse engineering process and tools be compatible with the underlying psychological process, considerable elaboration and refinement of the model is required if it is to be tool supported. To this end, Synchronized Refinement highlights the interplay of conceptual knowledge of the application and computer science domains, on the one hand, and data about the program being analyzed, on the other. Because of the large volume of detailed information that must be manipulated, tool support is required, and even relatively crude, "throw-away," tools suffice for collecting data

from the source code and subsequently reducing it to answer specific questions about the code and structural relationships among its parts.

The analysis of program text, as driven by design decisions, is new. Previous experiments contributing towards the development of this approach have been reported in [4] and [7]. Procedures based on control flow analysis and program slicing are described in a paper by Hausler[3].

2.2 An embedded software system

Synchronized Refinement was applied in experiments performed on several software components within a large, real-time software system built and maintained by a major telecommunications company. This product, a digital subscriber carrier that has undergone significant modification and enhancement in the five years it has been on the market, extends both regular and special telephone services from a switching center to residential and business communities. Its main application is to increase the number of subscribers that can be economically served by a feeder cable. Functionally, it converts between the digital signals carried on the telephone network and the analog signals required by telephone subscribers. It also does the time-slot management and multiplexing required to complete the interface between the telephone network and telephone subscriber.

Implementation: The application software is coded in a system programming language and consists of approximately 200K lines of code. The operating system on which the application runs is custom-built and consists of an additional 10K lines of assembly language code. Software engineers assigned to the product are responsible for maintaining both the application and the operating system.

The system is based on a multitasking operating system implemented on a single processor. The operating system schedules tasks in response to either internal or external events. The events may be signaled by messages, timers, or hardware interrupts. Events requesting action within real-time constraints are monitored by hardware detectors, and overall system sanity is maintained by a hardware-based watchdog timer—if the watchdog timer is not periodically reset, it will, on expiring, force the system to reinitialize.

The software running in this environment is responsible for system initialization; call processing and associated messaging; periodic audits and on-demand tests of selected components; failure analysis and recovery, which includes notifying the network that faults have occurred; and periodic switch-overs in which primary and backup components exchange roles.

Some of the experiments were run on a task responsible for scheduling and carrying out a variety of system tests. This automated system testing (AST) is responsible for verifying transmission paths and for auditing the operational status of line cards, the hardware components in the digital subscriber carrier to which the subscriber's telephone line connects. Other experiments included a study of the interactions among the watch-dog timer, interrupt service routines, and system tasks.

Design and design documentation: Design documentation was not systematically archived during the early years of the product's life. Consequently, design descriptions for many components are incomplete or unavailable. A considerable amount of information is available from personnel assigned to the product, but, because of the poor documentation, the learning curve's slope is relatively shallow and, once having mastered the design, engineers are not quickly rotated to other products.

There is a one-to-one correspondence between tasks and modules. Activities carried out in a module generally exhibit logical cohesion,[1] each task providing a designated class of services, e.g., testing, fault diagnosis, and network interface. This is not rigorously observed, however, and for the sake of optimization or as the consequence of previous maintenance, a task can bypass the designated service provider and implement certain operations itself, as is frequently the case with testing and fault diagnosis operations. If one task requires a service provided by another, it can also bypass the second task's interface and directly invoke the lower-level procedures. While the operating system supports interprocess messaging, many of the processes interact through a combination of common and control coupling, one procedure communicating with another by manipulating global data structures and changing values in global logical flags. Frequently, one task will set global flags to influence the path another task takes through a conditional, and this is often the means by which exceptions are raised and handled. Procedures use this same mechanism to inform their invokers of exceptions, and, to conserve stack space, procedures frequently communicate with their invokers by referencing global variables.

Context switches occur when tasks explicitly give

[1] This term and related phrases used below are part of Structured Design[8].

up control; there is no round-robin scheduling. Synchronization, when necessary, is by means of globally-accessible flags.

Finally, while the code is structured in the sense that there are no explicit "go-tos," procedures are generally not well organized: they have been decomposed into monolithic, deeply-nested conditionals rather than sequences of simply described functions; algorithms are often implemented using convoluted idioms; and there is no distinction between the normal execution paths through a procedure and those associated with exception handling.

3 Design analysis

Design decisions are used to derive an expected design from a conjectured conceptual model. In addition, code-level analysis simplifies and abstracts source code by recognizing and undoing design decisions. In order for this process to reveal the underlying structure of the code, it is often necessary to extract and analyze a large amount of data from the program text. For example, if the expected design decomposes the system into a sequence of simply specified functions and includes explicit exception handling, the code-level analysis is needed to confirm this design and fill in additional details. This code-level analysis separates the normal execution paths from those followed in exceptional circumstances. In addition, code-level analysis modifies the structure of the actual design by reordering program statements to match the expected design.

Whether deriving an expected design or performing code-level analysis, the software engineer must be familiar with a wide range of techniques for recognizing design decisions. This section describes several techniques for recognizing a decision's imprint on the code and distinguishes between high-level design decisions, which determine the overall software architecture, and low-level design decisions, which determine the data representations and control structures used within a functional component.

3.1 Recognizing high-level decisions

Many design decisions contribute to the design of the mechanisms responsible for program control, and the most prominent of these mechanisms constitute the software architecture. The architecture can include mechanisms for sequencing operations, managing memory, and controlling access to and representation of data. These architectural mechanisms supplement the program control provided by the programming language primitives and the run-time environment including the programming language's run-time system and the machine's operating system. In complex systems, this additional program control can be provided by layers of custom-built code running between the application and the predefined run-time environment.

Generally, we use an intermediate-level design language, i.e., a pseudocode, to express a simplified model of program control. In this model, control decisions, which are actually made by the application itself, appear as separate lower-level services. We find that the extra level of abstraction in the description of program control adds considerable clarity to our software models, hiding the application-level program control logic that makes code difficult to comprehend. Several examples of these simplification techniques are summarized here.

When recognizing the primary program control mechanisms in the software architecture, we begin by looking for mechanisms controlling the component as a whole and for idioms, or programming clichès, that are repeated throughout the program text. For example, in many programs a particular combination of control constructs and data structures are used throughout the text to support behaviors such as backtracking and lookahead. Similarly, a program can provide its own run-time support for services not normally provided by the run-time environment. Services that can be provided in this way include exception handling, inheritance, and dynamic memory management based on reference counts, automatic garbage collection, and structure sharing.

In the AST software, variables are used to support operation sequencing. For example, when faults are detected, the alarm is not sent immediately; instead the alarm operation is scheduled for later execution. In a related example, logical flags are set by one task or procedure to direct the flow of control in another.

Another technique, generalization, also plays an important role in recognizing a program's primary control mechanisms. For example, in these experiments we introduced explicit exception handling and showed many procedure and task interactions in terms of data coupling, deferring to lower levels decisions about how data coupling might be simulated by common or control coupling. Such generalization is important because the details of the simulation may differ from context to context.

In recognizing the program's primary control mechanisms, we also used a third technique, factoring. In

factoring, an existing design is divided into functionally cohesive units. For example, we isolated various non-critical behaviors as independent activities, localizing information that is distributed throughout the actual implementation, thereby hiding many important but conceptually intrusive design optimizations. Factoring is accomplished by dividing a block of sequential code into a number of small, short-lived processes, each describing a conceptually distinct activity.

Because of the low cohesion within modules and the heavy use of control coupling, factoring was difficult and time consuming. Considerable effort was expended in understanding the application-level logic for sequencing operations. A list of program statements, in effect, hard wires a sequence of operations while parallel processes defer many of those sequencing decisions until run time. In much of the design we studied, conceptually parallel activities were implemented sequentially but included conditionals, logical flags, and loops to reintroduce some of the flexibility that had been lost by eliminating the explicit parallelism.

3.2 Recognizing low-level decisions

There is no strict order for detecting low-level design decisions, though in some cases considerable data manipulation is required to reveal the underlying structure of the code. Complex programs are typically designed as layers of abstraction. During the reverse engineering process, as one layer is detected it opens the door for the detection of other decisions in higher layers. The following is a short summary of how we attempted to recognize and undo low-level design decisions. A more detailed description of this aspect of Synchronized Refinement is given in [7].

Even if the program does not enhance its own runtime environment, it may simulate advanced programming language features in selected contexts. Thus, we specificly look for advanced language features which have been simulated by combinations of more primitive constructs, i.e., application-level simulations of program control mechanisms. A common example of this is the recognition of code that uses nested IF statements or GOTOs to implement a CASE statement.

Similarly, we look for the use of primitive data structures to represent unavailable ones, such as the use of specific integer or character constants to encode the elements of a set of values. More difficult examples arise when the program, intentionally or not, contains numerous violations of the principles of data encapsulation and information hiding.

The control flow of the program must often be modified during code analysis. For example, in these experiments we reduce conditionals to decision trees and decision tables, and then resequence the operations around a different functional decomposition.

It is also important to look for special cases that can be related by a common generalization. We do this by looking for similar sections of code that differ only in a small number of ways. These sections are replaced by the parameterized use of a more abstract construct. In these experiments, we generalized several scenarios for testing line cards to obtain a single test procedure.

When the implementation language does not support modularization or when the program has been improperly modularized, we look for code that should be grouped together and separated from the rest of the program by an abstract interface. In these experiments, the language did support some measure of modularization, but many modules were not cohesive and there was strong intermodule coupling. In contrast, the re-engineered design exhibited considerable functional cohesion.

4 Experiments

Re-design is an inherent part of reverse engineering: a clear design summarizes a software engineer's understanding of how the code works, and a convoluted original design may require a new design with a structure substantially different from the original. Of course, the penalty for a comprehensible design is an offsetting complexity in the set of design decisions linking the new design to the original code.

The two experiments described in this section illustrate the range of ways in which domain knowledge can be combined with data about specific software artifacts to recover, validate, and improve a design. In both experiments insight into the design is obtained by manipulating a simple database tabulating the locations in which procedures, functions, and variables are defined or used. The database holds information on over two thousand procedures definitions and more than twenty-five thousand procedure calls. Many of the queries against this database involved constructing calling chains or identifying the global data structures and variables through which particular procedures or tasks communicate. More generally stated, the database is referenced as part of code analysis, both in restructuring the program text and in comparing it to the expected design.

4.1 Diagnosing and fixing a design error

The software engineers responsible for the digital subscriber carrier, at the time they approached us, had a plausible hypothesis that a rare system failure was the coincidence of an error in the software controlling the watchdog timer and stack overflow.

Reverse engineering techniques were used to analyze low-level data to identify circumstances under which stack overflow was possible. Reverse engineering techniques were also used to abstract the actual code, thereby constructing a model of the interaction between stack overflow and the design error in the watchdog timer. The model was a simple software system, capturing the salient aspects of the problem. In the actual code, the problem began when a task's stack overflowed. The overflow corrupted the lower portion of a second task's stack. Following a procedure return, this second task loaded a corrupted value into the program counter. Because of the corrupted program counter, the software began executing data and randomly modifying memory. Under these conditions, the watchdog timer was supposed to expire, causing the software to reinitialize. On rare occasions, however, the software, after it had "gone random," instead corrupted a flag, disabling the timer and preventing the system from resetting.

The model describing this interaction was used to test the efficacy of the repair proposed by the software engineers. The tests had to be run against the model because the combination of events leading to the system failure involved interactions among internal components and could not be recreated by simply manipulating the system's environment. The model was derived primarily from an expected design provided by the software engineers; however, code-level analysis was necessary to confirm the model's applicability.

Queries run against the database proved to be of particular importance in this problem because the transitive closure of the "calls" relation, in which tuples denote "Procedure X calls Procedure Y," generates a set of calling chains. Analysis of these chains is an important part of estimating stack requirements. To complete the analysis, however, additional data describing the type and number of parameters to each procedure was required. A tool was built that extracted this data from the code, and from the extracted data we estimated the amount of stack space required for each procedure's activation record. By constructing calling chains from cross reference data and then joining them with data on the size of the activation record for each call, estimates of stack usage were constructed. A similar procedure was used to calculate the stack requirements for interrupt service routines, which run in the context of the current process.

Software maintenance had over the years increased stack requirements, but the stacks had not been routinely resized because of the tedious nature of the calculation. After several years of maintenance, worst case combinations of interrupts on nearly full stacks, while a rare event, could cause stack overflow and system failure. With the new tools derived from our reverse engineering work, it is now possible to periodically re-estimate stack requirements and to confirm that maintenance has not reintroduced the possibility of stack overflow.

4.2 Redesigning a component

One of the tasks in the software system, the automatic system test task (AST), is responsible for periodically testing transmission paths and line cards. Since the code for this task is known to be particularly difficult to understand, reverse engineering techniques were applied to one of its most difficult components, line card testing.

Process summary: The process we used in this experiment was a direct application of Synchronized Refinement. Since Synchronized Refinement is an iterative process, its steps can be carried out in various sequences. The process as it was applied in this experiment began with few initial expectations regarding design or function, but in the end, had developed a conceptual model that factored the computation into several parallel activities. In addition, a new program-level design was obtained in which the parallel activities had been interleaved and the code had been decomposed into a sequence of simple functions. This contrasted with the original design in which the parallel activities had been combined into a single, complex functional component consisting of a large, deeply-nested conditional.

While the goal of this experiment was to redesign the line card testing component, the work began with code analysis. This initial analysis was exploratory and was not guided by any expectations about how line cards were tested or how the code was designed. The initial analysis concentrated on extracting data from the source code and undoing a number of low-level design decisions. Only after completing this exploratory code-level analysis did we develop expectations about the overall design of the component, and the design we eventually constructed to explain line

card testing differed considerably from the actual design of the original code. We were able to use this alternate design to hide a large number of convoluted design decisions linking it to the original implementation. We were also able to reimplement line card testing around an alternate design, using a considerably simpler set of design decisions.

We represented the design using pseudocode and, in doing so, availed ourselves of a wide range of language constructs. The enriched vocabulary we used allowed us to clarify relationships between the structure and behavior of the software, and we used context-specific translation rules[5, 9, 10] to represent design decisions linking the pseudocode to the implementation.

For example, we used this technique to introduce exception handlers at the pseudo-code level. We also used this technique to abstract away the many resource-specific details associated with requesting and releasing resources. The deeply-nested conditionals proved particularly difficult to understand, and only after considerable code-level analysis were we able to factor them into the functions of line card testing, selecting resources, acquiring and releasing the resources, and deadlock prevention.

Code analysis: In the original component, a typical procedure consisted of a single, deeply-nested conditional. In the AST code, conditionals were nested as many as fourteen levels deep and could span a dozen or more pages. Primitive operations in one procedure often were calls to other procedures having similar structure. Adding to the complexity of the code were the individual conditions, often large Boolean expressions referencing many variables and functions.

After considerable code-level analysis we eventually discovered that the fragment shown in Figure 1 could be paraphrased as shown in Figure 2.

```
IF ((READ_AB_SUS_CND (ABORT_AUTO_CND) = FALSE_GL)
            AND
    (ABORT_FLAG_G = RESET_GL)
            AND
    AUTO_SYS_TST_ENABLE_G )
THEN DO;
    IF ((TST_RDY (LN_TST_TYPE_GL, GRP_LPT)=TRUE_GL)
            AND
       (RT_MTNC_CP = TRUE_GL))
    THEN DO;
        DO WHILE ((PROCEED = FALSE_GL)
                        AND
                  ATTEMPT < 3);
                    ⋮
```

Figure 1: Extract from sample code

"if the test has not been aborted and
the first two resources have been acquired then
try three times to get the next resource"

Figure 2: Paraphrased code extract

get_Resource_Set ($S := \{R_1, R_2, R_3, R_4, R_5\}$)
on Resource_Unavailable exception
Release_Resources(S)
Skip_Test
end

Figure 3: A higher-level paraphrase

With further restructuring, exception handlers were uncovered that directed execution when the test was aborted by other tasks or when resources were unavailable, e.g., Figure 3.

Our first step towards these higher-level models was to understand the various combinations of conditions that could cause primitive operations to be invoked, and we represented this model as a decision table. We also described the sequences of primitive operations on the various paths through the conditional, and attempted to identify the significance of the various tests controlling the path through the conditional.

Next, we recognized many of the conditions involved flags with names suggesting exceptional conditions. Following up on this recognition, we reorganized the decision table and path expressions to isolate the code related to exception handling from the main line of the line card test. In some cases, we examined references to a variable in several tasks before determining whether it was used to raise exceptions or for some other purpose. Further consideration revealed some similarities among the alternative normal paths and the various sequences of operations performed once an exception flag has been set. Our next step was to make sense of these apparent similarities.

Expected design: As has been mentioned, the expected design was formulated after some of the code-level analysis was completed. For example, landmarks in the program text suggested that the normal paths through the code were similar, all specializations of the same function, "line card testing." Our first conjecture for the structure of the canonical line card test was the following: set the test up; perform the test; collect, analyze and report the results; and, finally,

take the test down. After comparing this expected pattern to the actual paths through the code, we developed a refined set of expectations, shown in Figure 4, that fit the code quite well. Observe that this expected design composes the **Gather_Resources** component from Figure 3 with other operations.

```
Decide_On_Required_Resources
Gather_Resources
Perform_Test
if Test_Failed then Diagnose_Fault
Record_and_Report_Test_Results
Release_Resources
on Resource_Unavailable or Test_Abandoned exception
        Release_Resources
end
```

Figure 4: Expected design

The model in Figure 4 has several virtues, including the use of small functional components that divide the procedure into conceptually simple phases.

Similar techniques were also used to restructure the **Diagnose_Fault** component. Fault diagnosis tries various combinations of primary and backup equipment until it isolates the faulty component. Whereas the original design used the location in the conditional as an implicit record of events, the new model broke the process into phases and introduced additional variables to explicitly remember those events. Modifying the decomposition removed considerable redundancy in the code, allowed the reporting of test results to be separated from the process of diagnosis, and made explicit the circumstances under which various alarms are sent.

Testing expectations: The revised design set up several expectations that were difficult to verify, requiring the analysis of a large amount of data extracted from the code. For example, the order in which two operations were performed was not the same in all paths, i.e., one path showed **A** followed by **B** and other, **B** followed by **A**. We were concerned that the difference in order may have been significant. By tracing calling chains and collecting global variable references, including accesses to physical devices, we were able to determine that there were no read/write conflicts and the distinction between the two cases could be eliminated.

Once the existence of the exception handlers was recognized, we were still faced with the task of determining their role in the overall behavior of the component. In filling in these details we analyzed data recovered from the code. We made the empirical observation that many of the operations in the exception handlers were releasing resources. This led us to observe that many of the resources required for a test were shared among several tasks. This allowed us to recognize the potential for deadlock, and we then considered how deadlock was prevented. While there were no obvious landmarks in the code, we guessed that the software used a version of the deny-hold-and-wait strategy[6], i.e., if a resource could not be obtained, all resources acquired so far were released and the test was abandoned. From this we were able to produce an improved design showing how a deny-hold-and-wait strategy could be implemented using exception handlers.

Our analysis eventually confirmed this expectation, and by tracing calling chains and control and data flow, we were able to construct a list of resources, including some synchronization flags, that were not well marked. Code-level analysis also pointed out some variations on the basic pattern. One variation provided for preemption: if a high priority task set a global flag, for example, testing is abandoned and resources released.

Additional variations were discovered in the way some of the resources were handled. There were, for example, a number of variations in the protocols used for selecting, acquiring, and releasing various resources. In the case of two resources, we were initially unable to find evidence that they were being released when exceptions occurred. But, by referring to our database, we were able to construct and trace the calling chains associated with line card testing. For one resource we were able to find the "missing" release operations. It turned out that some of the low-level procedures were designed to be paranoid: as soon as a low-level procedure determined that the test could not be completed, this particular resource was immediately released, even before the exception flag was set or control returned to the main testing procedure. In the second case, we observed that along some paths a message releasing a resource was never being sent. While this appears to have been a bug in the original program, by examining various references to the resource, we found evidence that the bug was fixed, though in a convoluted way. Rather than figuring out who was failing to release the resource, the maintenance programmer modified other components so that they would forcibly reclaim that resource without ever identifying and notifying the task currently

holding it. The underlying assumption, correct but undocumented, was that the unidentified task was by that time no longer using the resource and had merely neglected to release it.

5 Conclusions

Our original purpose in these experiments was to evaluate the power of techniques that relied primarily on data extracted from the program text, e.g., constructing models of control flow, calling chains and variable references. While the database could be built and manipulated using simple tools, we quickly recognized that program text is inherently ambiguous: the purpose being served by a particular program structure depends on contextual information not found in the program text. Consequently, a software engineer engaged in reverse engineering must draw on a broader knowledge base, reconstruct that missing context, and derive expectations regarding the software design. In these experiments, we were able to improve both on the original developers' understanding of the program's context and on the program's basic design.

Acknowledgements

The authors gratefully acknowledge the cooperation and support of Richard LeBlanc, Jacques Bolduc, David Cullen, Yernie Rafol, Sharad Rao, and Jim White.

References

[1] Ruven Brooks, "Towards a Theory of the Comprehension of Computer Programs," *International Journal of Man-Machine Studies*, vol. 18, pp. 543-554, 1983.

[2] Elliot J. Chikofsky and James H. Cross, II, "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software*, vol. 7, no. 1, pp. 13-17, January 1990.

[3] Philip A. Hausler, Mark G. Pleszkoch, Richard C. Linger, and Alan R. Hevner, "Using Function Abstraction to Understand Program Behavior," *IEEE Software*, vol. 7, no. 1, pp. 55-63, January 1990.

[4] Kit Kamper and Spencer Rugaber, "A Reverse Engineering Methodology for Data Processing Applications," GIT-SERC-90/02, Software Engineering Research Center, Georgia Institute of Technology, March 1990.

[5] James M. Neighbors, "Draco: A Method for Engineering Reusable Software Components," *Software Reusability: Concepts and Models*, ed. Ted J. Biggerstaff and Alan J. Perlis, vol. 1, Addison Wesley, 1989.

[6] James L. Peterson and Abraham Silberschatz, *Operating Systems Concepts*, second edition, Addison Wesley, 1985.

[7] Spencer Rugaber, Stephen B. Ornburn, and Richard J. LeBlanc, Jr., "Recognizing Design Decisions in Programs," *IEEE Software*, vol. 7, no. 1, pp. 46-54, January 1990.

[8] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured Design," *IBM Systems Journal*, vol. 13, no. 2, pp. 115-139, 1974.

[9] David S. Wile, "Program Developments: Formal Explanations of Implementations," *Communications of the ACM*, vol. 26, no. 11, pp. 902-910, November 1983.

[10] D. S. Wile, "Local Formalisms: Widening the Spectrum of Wide-Spectrum Languages," *Program Specification and Transformation*, L. G. L. T. Meertens, Elsevier North Holland, pp. 165-195, 1987.