Ripple effect analysis based on semantic information

by JAMES S. COLLOFELLO
Arizona State University
Tempe, Arizona

and

D.A. VENNERGRUND
TRW Federal Systems Group
Fairfax, Virginia

ABSTRACT

Maintenance of large-scale software systems is a complex and expensive process. This process is often unreliable due to the ripple effect of modifications in one component of the system adversely affecting other components. Although syntactic techniques exist for tracing ripple effect, their results are often crude and require considerable interpretation by maintenance personnel. In this paper, a prototype ripple effect analysis tool based on both syntactic and semantic information will be described. This tool enables maintenance expertise to be captured in the form of semantic conditions which can then be linked to syntactic components. The ripple effect tool can then guide the maintenance personnel in tracing ripple effect as a consequence of a program modification. The functional capabilities of this tool are presented in this paper as well as an overview of the tool’s architecture. Some experience with the tool as well as suggestions for future research are noted.
BACKGROUND

The development and maintenance of large software systems is a difficult and complex task, compounded by both technical and management factors including the size and complexity of the software system and the teams developing it. Today's larger systems contain thousands of software and hardware components developed by large teams with varying levels of experience. Development may take years with maintenance of many systems spanning decades. Ensuring a working system over long development and maintenance life spans is a difficult task.

A major software engineering concern is that software maintenance accounts for such a large percentage of the total cost of software systems. J. Martin, C. McClure, and B. Patkau estimate that from 40% to 80% of a system's cost is maintenance, costs incurred after the initial release. The goal of several software engineering techniques and research methods is to lower this cost.

Software maintenance, in this context, encompasses the correction of incorrect software, the adaptation of software to a new or changing environment, the perfection of software, and the addition of new software. In short, software maintenance can be defined to be any change made to an existing software system. With this definition, a model of software maintenance can be described.

The software maintenance model proposed in "Ripple Effect Analysis of Software Maintenance" provides a useful basis for describing the software maintenance process. The model presents four phases followed to make a change: understand the software, propose a solution, account for ripple effect, and test the solution in the system. Each of these tasks is difficult and requires a great deal of knowledge about the system under maintenance. Unfortunately most of this knowledge is experiential, unformalized, and thus unavailable to the novice maintainer.

The software maintenance tasks are difficult to begin with; compound this with out-of-date and incomplete information, high turnover rates of experienced designers, inexperienced designers assigned to maintenance, and a cumbersome methodology, and the tasks become very costly.

Improvements to the difficult task of software maintenance can be made by providing formalization in the form of automated assistance. Tools are needed for each of the software maintenance tasks including: creating, managing, and storing maintenance information, suggesting solutions, verifying changes, and enforcing methodologies. With such tools and information, a body of knowledge useful for software maintenance can be built and maintained for future generations of maintainers.

Few such tools exist to date. Most maintenance activity is accomplished with the aid of a minimal toolset containing a text editor and a compiler. More advanced maintenance environments may contain syntax-directed editors, diagnostic compilers, change tracking data bases, design languages, on-line documentation relating to requirements, design and testing information, and various metric and syntax analysis tools. Such tools might include data and control flow mappers, symbol cross-references, data dictionaries, ripple effect analyzers, code auditors, and complexity analyzers. Few environments provide a coherent integration of these tools and fewer still capture and provide experiential knowledge of the system under maintenance. Semantic information is desperately needed to perform effective software maintenance.

The remainder of this paper focuses on the providing some experiential information to the novice in one task, ripple effect analysis, the process of examining system software for impacts that may result from changes. A tool which ties semantic information to syntax information based on today's systems is described.

CURRENT RIPPLE EFFECT ANALYSIS TECHNIQUES AND TOOLS

Analyzing the effects of change on software is a difficult task, complicated by program size, complexity, and information hiding. Complicating factors external to the program include: multiple representations of information, out-of-date documentation, undocumented previous changes, and difficulty in tracing code to the design and requirements. Several ripple effect analysis tools and techniques are known, and two distinct categories of these tools can be identified.

The first, syntax-based ripple effect analysis techniques, work on source code representations alone. These techniques determine the effects of changes by examining syntax information like control and data flow. The second, semantic-based ripple effect analysis techniques, work on higher-level information not derivable from the source. This semantic information reflects the intent of a program and represents knowledge of the basic assumptions which must be true for the correct operation of the system.

There are many differences between these two strategies; syntactic information is easily derived from the source code, semantic information is difficult to derive, and difficult to verify. Analysis based solely on syntactic information is worst case, and may implicate many sections of code not truly affected, whereas semantic analysis can pinpoint effects more precisely at the cost of being incomplete. Semantic informa-
tion is informal and manually-derived, thus any analysis based on it is incomplete at best.

Syntax Methods

The most basic ripple effect analysis method, manual inspection, is performed with no tools except perhaps a text editor. This method is the most labor-intensive and incomplete yet it is widely practiced in industry. Automatically derived control and data flow information help the manual inspection method by providing details on the possible propagation of the change. Program data bases are useful change impact tools since they provide a consistent representation of program information which can be queried. Data dictionaries and symbol cross-references are basic program data bases. Tools like DAVE, FAST and ISUS detect certain ripple effect errors in a data base query fashion.

Typestate and Assertion statement methods are also useful in detecting ripple effect at compile and run-time. Both methods detect nonsensical errors, many caused by ripple effect. The highest level of syntax analysis is embodied in logical ripple effect analysis tools which are focused on determining the worst case extent of change effects. The weakness of this method is that in large systems it produces far too much information. The need to reason about change is apparent.

Semantic Methods

Semantic methods attempt to reason about the meaning of changes. Many approaches have been taken to meet this task. Each recognizes the need to represent and use knowledge of the application domain as well as the programming language. Formalizing this kind of knowledge is a difficult task, as represented by the many varied approaches. Each of these approaches require the following:

1. Formal representations for requirements, design and related documentation
2. Formal representations of programming constructs
3. Methods for manipulating all representations
4. Methods for relating all representations
5. Methods for acquiring the representations

The trend in ripple effect analysis, as well as software engineering, is toward representing and using higher level concepts. The conceptual leap from assembler to high-level languages opened a new world of programming opportunities. A similar jump to another level of languages has been predicted and anticipated for years. Studies in artificial intelligence, data base technology, conceptual modeling and cognitive sciences all attempt to answer the basic questions which will open the door to higher-level language programming: How to represent knowledge, and how to reason with knowledge.

Recent success and popularity in the expert system subfield of artificial intelligence has sparked a new euphoria. The application of expert system technology is appropriate for limited domains of knowledge, problems already solvable by a set of best guesses, or heuristics. Unfortunately, most of the problems associated with reasoning about computer programs remain unsolved, or contain far too much knowledge to be represented.

In the ripple effect problem, many interesting pieces may be solved with expert system technology. The derivation of interesting, subtle or confusing dependencies can be based on knowledge of programming constructs and application intent. Such dependencies could be partially derived from source comments and design documents. Smart documentation assistants, as described in the Intelligent Program Editor and the Programmer's Apprentice are essential to the organization and derivation of such dependency information.

Change propagation libraries may also be amenable to expert system technology. Such libraries would contain a pattern-matchable description of all manner of ripple effects, both logical and performance. Thus when a certain change is made, the change propagation knowledge base can pinpoint with best guesses all effected components.

Unlike expert system applications include the derivation of semantic impact from a syntax change. To understand the effects of a change, a semantic representation of the change must also be made. No method for inferring the meaning of a change, based on syntax alone is possible.

Other approaches to semantic representation are found in the integration of software engineering environments. Based on data base techniques, this approach can model any relationship and enforce any constraint. Thus semantic information is able to be represented and manipulated. Conceptually, then, the knowledge-based approach and integrated environment are equivalent. Clearly, in order to reason about meaning we need to represent meaning.

A PROPOSED SEMANTIC INFORMATION TOOL

Introduction

The existing software base is a significant asset, and little of it contains any semantic information. Reasoning about this software is a difficult and mostly manual task. Providing automated assistance to the maintenance of such programs is the goal of today's tools. Semantic annotations of syntactic relationships is a first step towards incorporating information useful in formalizing the meaning of syntax. Later steps will build on this information until a new development paradigm evolves. This incremental development of support tools, that transform today's meaningless programs into tomorrow's meaningful programs, is a viable approach.

A tool which can provide the maintainer with up-to-date semantic information tied directly to source code and express the meaning of the source is necessary for the efficient and effective analysis of ripple effect. The many approaches to this need were discussed in the previous section. Yet most of the approaches cannot be applied to today's software base, because they are designed for symbolic languages and specially created programs. Examples include the formally verified programs built by the Designer/Verifier's Assistant and the Programmer's Assistant.
A tool that captures and presents semantic information directly related to program dependencies is now described. The tool, Semantic Information Tool (SEMIT), is targeted for the majority of today's software systems written in imperative, high-level programming languages.

Imperative programming languages achieve their primary effect by changing the state of variables with assignment statements. Since ripple effect is propagated through the path of variable assignments, a documentation of critical assignment effects by changing the state of variables with assignment statements will aid ripple effect analysis.

SEMIT addresses programs with these characteristics by assuming an imperative-based syntax analysis, creating a syntax and semantic data base, and directly linking the semantic information to the program syntax. The combination of semantic information and program source presents a model of the program that can grow with software maintenance.

**SEMIT Overview**

SEMIT provides the ability to link semantic information to source code by representing the syntax in a relational format which captures data flow dependencies and any useful descriptions of the dependency in the form of semantic conditions. Semantic conditions are assumptions or assertions about data item properties or program states. Examples include, "the array is sorted," and "the input value is non-negative."

The conceptual model of SEMIT is based on the Semantic NET. A network records all syntax relations. Logical relations describe key information derived from procedures and data. Figure 1 shows a simple network for a procedure FIND_TOP_SALESPERSON. Procedures and data items are represented by boxes, relations between them with arcs. Note how the relations between the procedure and other program components are represented with relations. Modified data, used data, called procedures, and passed parameters are easily represented. These relations are strong enough to describe all possible logical ripple effect paths for inter-procedure cases.

The syntax in Figure 1 represents the fact that "the procedure PRETTY_PRINT is called by the procedure FIND_TOP_SALESPERSON with the parameter TOP_SALESPERSON which has been modified by the procedure FIND_TOP_SALESPERSON." Stated in a set of logical relations:

1. modifies(FIND_TOP_SALESPERSON, TOP_SALESPERSON)
2. calls(FIND_TOP_SALESPERSON, PRETTY_PRINT)
3. called-with-param(PRETTY_PRINT, TOP_SALESPERSON)

A simplification of this relationship is noted by the existence of a modifies-uses chain between the modification of TOP_SALESPERSON by the procedure FIND_TOP_SALESPERSON and the usage of TOP_SALESPERSON by the called procedure PRETTY_PRINT. There exists a dependency between these two procedures, since a change in the procedure FIND_TOP_SALESPERSON in the modification of TOP_SALESPERSON may affect the correct operation of the procedure PRETTY_PRINT. This dependency may be stated with the pair of logical relations:

1. modifies(FIND_TOP_SALESPERSON, TOP_SALESPERSON)
2. uses(PRETTY_PRINT, TOP_SALESPERSON)

Semantic conditions can be used to describe the dependency between the modifies and uses relations. A semantic condition, for example, may describe the condition established by the procedure FIND_TOP_SALESPERSON and used by the procedure PRETTY_PRINT. Examples include: TOP_SALESPERSON must not be the TOP_SALESPERSON from the previous month, or TOP_SALESPERSON is a record containing first and last names. Only the more interesting, subtle, or confusing relations need be described. Typical semantic conditions include the assumptions on program control, whether an action has been performed or not, and constraints on data item values, especially input and output parameters.

If component A establishes a condition and component B uses that condition to execute correctly, then component B is said to be semantically dependent on the condition established by component A. Figure 2 illustrates a simple semantic condition between a procedure A and procedure B. Procedure B depends on procedure A properly sorting the array Y. With semantic conditions linked to syntax, the effects of a change to syntax can be traced to semantically dependent components. In this manner, knowledge of semantic dependencies effectively reduces the very large set of possible ripple effects to a much smaller set of probable ripple effects.

The construction of such a base of semantic knowledge linked to syntax information will limit ripple effect analysis to

<table>
<thead>
<tr>
<th>Component</th>
<th>Relation</th>
<th>Component</th>
<th>Semantic Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>procedure A</td>
<td>modifies</td>
<td>data-item Y</td>
<td>&quot;Array Y is sorted&quot;</td>
</tr>
<tr>
<td>procedure B</td>
<td>uses</td>
<td>data-item Y</td>
<td>&quot;Assumes Array Y is sorted&quot;</td>
</tr>
<tr>
<td>procedure B</td>
<td>depends-on</td>
<td>procedure A</td>
<td>&quot;Array Y is sorted&quot;</td>
</tr>
</tbody>
</table>

Figure 2—Semantic dependency
simple network transversal. When a syntax change affects modifies-uses relations, the linked semantic condition is implicated. If the maintainer determines the semantic condition is still valid, no further analysis occurs. Otherwise a display of all the components dependent on the semantic condition is presented. The semantic data base will serve as "corporate memory," a structured repository for information typically embedded and forgotten in comments and supporting documentation.

Capabilities

SEMIT provides capabilities to the software maintainer in performing the following tasks: linking semantic descriptions to syntactic dependencies, using the semantic information in ripple effect analysis, and linking semantic conditions to external documents.

Syntax and semantics linked in a semantic data base

SEMIT creates a semantic data base by deriving default syntax relations from the source code, grouping the relations into possible dependencies based on modifies-uses paths, and then prompts the maintainer to describe the dependent components with semantic conditions. This keeps semantic and syntax information in-step and consistent.

Semantic ripple effect analysis

SEMIT provides semantic ripple effect analysis by presenting the semantic information in the semantic data base to the maintainer. This allows the maintainer to determine the effects of change based on both syntax and semantic information. If a procedure which establishes a semantic condition is changed in such a manner as to affect that condition, then all other components which depend on that condition are possibly impacted. In doing so, the analysis ignores the numerous syntactic ripple effects as derived by data and control flow analysis and thus focuses the analysis to more probable paths as defined by the expert maintainer.

External documentation mapped to semantic conditions

Semantic conditions are also useful for describing dependencies to system documentation external to the source code. These dependencies provide the traceability necessary for up-to-date and consistent documentation. The addition of one-directional dependencies is an ad-hoc procedure, based on the existence and format of existing documentation. The maintainer must explicitly describe the dependency, no syntactic defaults are derivable.

Architecture

The architecture of SEMIT is composed of two basic components, the Semantic Data Base and the SEMIT system. External components include the Source Code and the Syntax Analysis module which builds the default Semantic Data Base.

Figure 3 represents the architecture of SEMIT. Note the bidirectional flow of information from the user to the Semantic Data Base via the SEMIT Control. This represents the two modes of SEMIT usage, adding information and using information.

Syntax Analysis

The Syntax Analysis program builds the initial Semantic Data Base based on program control flow and data flow. For each procedure in a program, all external data used and modified by the procedure are represented in the network. All data item and control flow relations are represented with the following relations:

1. uses (procedure_name, data_name)
2. modifies (procedure_name, data_name)
3. calls (procedure_name, procedure_name)

SEMIT Control

SEMIT Control performs three different functions. It assists the annotation of default dependencies with semantic conditions, maps syntax changes to semantic conditions, and lists the dependent components of changed semantic conditions.

Annotating default dependencies to create semantic conditions is the primary knowledge acquisition function of SEMIT. The maintainer documents the more interesting syntactic or performance dependencies internal to the source, and dependencies to external documentation. As the description is made, a consistency check is made to insure at most one establisher and at least one user of the semantic condition exists.

The Change Analysis function is an interactive tool to aid the user in the process of making a change. It maps a syntax change to existing semantic conditions if directly linked. The impacted conditions are a list of all the semantic conditions established by the procedure which may now be inconsistent. The user examines the list and filters out those that are still valid, then performs dependency analysis on the changed conditions.

The Dependency Analysis function examines the Semantic Data Base for all components reliant on the impacted semantic condition. It lists those components while prompting the maintainer for verification of correctness. SEMIT cannot reason about the consistency, and must rely on the maintainer's
skill and judgment. Dependency chains may then be followed by the maintainer if a change ripples through more than one set of dependencies.

Prototyping Experiences

SEMIT was prototyped to show the feasibility of the concept and model the user interface. A series of adjustments to the tool were identified based on the prototype. The first observation involved the nature of ripple effect error flow and the types of semantic conditions typically entered into SEMIT. Examples tended to emphasize modifies and uses pairings. Thus a simplification of all syntax relations summarized by a modifies-uses pairing seems possible. Annotation of these syntax pairs is a simple and useful solution.

Further understanding of the difficulty of performing semantic ripple effect was also discovered in the prototype by the attempt to map syntax changes to semantic conditions. The prototype only considered the simplest class of syntax changes; a modification of an existing source line. SEMIT then checked for all the semantic conditions which referred to that source line, and considered them impacted. Deleted lines, too, were merely mapped to any semantic condition which used it. All semantic conditions that were linked to source lines after the deletion were implicated.

A more interesting situation becomes apparent when the addition of source lines is considered. In the event of a maintenance operation which added syntax lines, the prototype initially assumed no impact on existing semantic conditions. In practice, the adjacent semantic conditions, the conditions linked to source before and after the added lines are often affected.

User interface issues were also identified by the prototype experience. Menus were useful in coordinating the user activities. A fully interactive interface, however, would provide more benefits. Stereotyped action patterns (schemes) which lead the user through maintenance actions should be developed. For example, a common scheme involves making a syntax change, mapping its impacts to semantic conditions, pursuing a breadth-first examination of all primary and secondary dependent components, followed by the iterative examination of all dependent components. Primary dependent components are those components directly dependent on a particular semantic condition, secondary dependent components are those components indirectly dependent on a particular semantic condition via dependency on a primary dependent component. Other schemes might enforce a particular interaction, requesting reports, creation of documentation, and communication of impacts to other maintainers.

In following semantic ripple effects, it quickly became apparent that many screens (or windows) are necessary to fully understand the current situation. A single screen was used with a small work list tracking actions taken and actions to take. This list was useful. A more appealing solution would utilize multiple windows: displaying source code, the change, semantic conditions, users of conditions, and a network of related components. Especially important is the need to use such a tool from within an editor, thus allowing maintenance and analysis in parallel. With such a wide-band of information, maintenance would be easier to relate back to the original change, rather than through a long string of dependent impacts.

Finally, feasibility was shown in the sense that semantic information can be stored and used for an approximation to semantic ripple effect analysis. The power of the information used to reason is only as useful as the information originally entered.

FUTURE RESEARCH

In the course of prototyping, many ideas toward integrating and extending SEMIT surfaced. User interface concerns and functional extensions were noted. Considering SEMIT primarily as a semantic data base representation of a program's dependency information, the following capabilities would extend SEMIT into a more useful maintenance tool.

Incorporate Schema-Driven Assistance. A Software Maintenance assistant, similar to the Designer/Verifiers Assistant or the Programmer's Apprentice, could lead maintainers through the basic tasks of creating and using semantic conditions in a mixed-initiative interaction. Schema-driven assistance is based on the knowledge of a stereotypical interaction. The assistant would lead and follow the maintainer based on the type of actions being performed. Such an assistant would eventually be knowledgeable of all system documentation and helpful in its presentation.

Improve User Interface. A multi-window based approach is required to present the wealth of information necessary. Work on multiple tasks in the same ripple analysis needs to be supported.

Incorporate Semantic Data Base in A Relational DBMS. The prototype of SEMIT does not represent the semantic information in a relational manner. However, the semantic dependencies are relations between objects, perfectly suited for a relational data base representation.

Develop a Relationship Library. The prototype SEMIT creates default dependencies based on syntax analysis alone. The relations "modifies," "uses" and "calls" are system defaults. Any relation can be represented. A library of the most common and useful relations could be developed. Examples include the performance relationships described in "Ripple Effect Analysis of Software Maintenance," and relationships to related documentation external to the source. Automatic derivation of some of these relations is possible during the syntax analysis stage.

Integrate Syntax-Directed Editor. A link between an existing semantic data base and a syntax-based editor would provide intelligent editing. Any syntax change made that affected a semantic condition could be examined and processed in the background unbeknownst to the maintainer. Special considerations would be necessary when reasoning about the effects of partial changes, and the addition and deletion of source lines.

Provide Analysis Control. Finally, the analysis performed by SEMIT should be controllable by the maintainer. Extent options might include program, sub-program, or module level analysis. Summary information options would limit deep analysis, producing various summary lists of possible impacts.
REFERENCES

15. Pattacini, A. Foundation for Software Maintenance, Department of Computer Science, University of Toronto, 1983.