Syntactic information useful for software maintenance

by JAMES S. COLLOFELLO and JOHELEN W. BLAYLOCK
Arizona State University
Tempe, Arizona

ABSTRACT

Software is an expensive asset, one which costs more to maintain than to acquire. The high cost of maintenance has been attributed to low programmer productivity in maintenance. This is due in part to the lack of tools existing within an integrated environment which address the full spectrum of maintenance activities. This paper addresses these concerns through the discussion of an ambitious research project to develop a Maintenance Engineering Environment (ME2). ME2 consists of three components: (1) a knowledge base, (2) an integrated toolset, and (3) maintenance personnel. Following a general discussion of ME2, this paper presents research results that identify the set of syntactic information needed to perform software maintenance activities. This information is critical to establishing the foundation of the maintenance-knowledge base in the ME2 project. The results also demonstrate the feasibility of evaluating source code for maintenance-specific information.
INTRODUCTION

There are a number of problems associated with maintenance, the greatest of these being that a programmer must understand a system before modifying it. A simple solution to this problem would be to retain the development team to function as the maintenance staff, but this is not always possible. Due to high programmer turnover in the industry coupled with the stigma attached to the maintenance task, it is difficult to retain experienced personnel in maintenance. Thus, maintenance is performed by unskilled programmers who must put considerable effort into learning the workings of the system before they can competently modify it. This learning task is often obstructed by a number of problems: documentation may be non-existent, insufficient, or outdated and useless, leaving the programmer with only the code to go by; the code may be unstructured and highly complex, making it difficult to understand; and the code may have been subjected to numerous modifications by other programmers who also did not understand the program very well, thus degrading system structure and system robustness.

It has been suggested that one way to improve maintenance productivity is to produce development tools that can be carried over into the maintenance phase, but development tools do not always translate well to the maintenance task because they may not address a number of maintenance problems. One maintenance problem that does not exist during development is the task of coming to understand an unfamiliar program. This stems from the fact that personnel with intimate system knowledge acquired during design are largely unavailable for making changes during development. Another maintenance-specific problem is the necessity of working within established system constraints such as available variable storage space and performance requirements. In addition to the problems peculiar to maintenance, there are many programs in existence which were not developed with the aid of advanced tools that must be maintained. These factors coupled with the high cost of software maintenance point to the need for a maintenance environment directed at improving maintenance productivity by reducing the time spent by maintenance programmers to acquire knowledge about unfamiliar programs, design and install modifications, trace ripple effect, and retest the system.

A MAINTENANCE ENGINEERING ENVIRONMENT

A project is currently underway at Arizona State University to produce a prototype Maintenance Engineering Environment (ME2). The environment will be applicable for both programs with no support documents other than the source code, and programs with a full complement of support information. ME2 has three components: (1) a maintenance knowledge base, the foundation of the maintenance environment, (2) an integrated toolset, and (3) maintenance personnel.

The Maintenance-Knowledge Base

The maintenance-knowledge base is the core of ME2, providing the power of a database through storage, modification, retrieval and deletion of data. As the maintenance personnel interact with the information, the knowledge base "learns" by deriving new information from the interaction and adding it to the environment. With the new information, the knowledge base is able to create new relationships not explicitly given to it by the maintenance personnel.

The Integrated Toolset

The ME2 toolset focuses on technical support tools covering each of the maintenance activities of understanding, modification, and ripple effect analysis.

The tools directed at the first maintenance task, that of aiding a programmer to understand a program, provide the user with a high-level, problem-knowledge domain view of the program by providing program requirements information. At the lower-level program-knowledge domain, the maintenance programmer is able to view a graphic display of the module-calling hierarchy, along with module imports and exports for each module. Syntactic information on module import and export variables and semantic information on the module and its purpose is also available.

During the second maintenance task, that of modifying the software, ME2 provides high traceability from requirements through design of the code. ME2 also supports software change control management.

A ripple effect analyzer tool is incorporated into the maintenance environment to support the third maintenance activity. For each module in the program, assumptions are identified which must be satisfied for correct operation of the module. The module, in turn, makes "decisions" based upon these assumptions. These decisions, in turn, may affect other assumptions. When a decision in a module is changed during maintenance, the change is statically traced to determine if the assumptions dependent upon the decision have been affected.

Cost-effective regression testing, although very difficult, is also under investigation for the maintenance environment.
The Maintenance Personnel

As noted, experienced maintenance personnel are difficult to acquire and keep. They are also the most valuable element in the maintenance environment. ME2 proposes to capture experience gained in the maintenance-knowledge base for the purpose of simplifying the learning task for replacement personnel. ME2 accomplishes this by querying the programmer for semantic problem and program knowledge acquired during the maintenance process. Thus, as more maintenance experience is acquired and captured, the power of the ME2 toolset is increased.

SYNTACTIC INFORMATION USEFUL TO MAINTENANCE

A key aspect of the ME2 project is defining information which will be useful in the maintenance task. This information is the substance of the maintenance-knowledge base and is based upon a foundation of syntactic information derived from the program. In order to describe the set of syntactic information which facilitates maintenance, a set of templates was developed. These templates were created to be language independent. This means that some fields in the template may have information which is syntactically available from some languages but only semantically available from others. The set of templates for capturing syntactic information are provided at the end of this paper, and are described below.

Control Flow Information

Knowledge of a program’s control flow is a key component in gaining an overall understanding of how a program works, and in ripple effect analysis and regression testing. Control flow information is also basic to an understanding of program data flow.

Preliminary to the task of understanding a program’s control flow is the task of identifying program basic blocks and modules. Basic blocks are defined as a “maximal group of statements such that no transfer occurs into a group except to the first statement in that group, and once the first statement is executed, all statements in the group are executed sequentially.” Modules are defined as the subprogram facility for the particular language. In Pascal, modules are the defined procedures and functions; in Fortran, they are subroutines and functions. Assembly languages may provide module facilities through a Jump-to-subroutine (JSR) or some other type of command which may use a stack to invoke and return from a subprogram.

Once the set of program blocks and modules is defined, inter-block control flow can be modeled using a directed graph where each node of the graph corresponds to a basic block. There is an arc (x, y) from block x to block y if control can potentially transfer from block x to block y at run-time. Intermodule control flow may be captured in a representation of the module-calling network of the program using a similar graphing technique. Both inter-block and inter-module control flow information may be derived from a syntactic analysis of the program.

Control flow information is useful to the maintenance task in a number of ways; identification of program blocks, modules, and processes is basic to identification of the control flow between these entities; block and module flow networks provide the programmer with information on how the program modules interact; and information used to create a reachability matrix for use during regression testing is available from inter-block control flow information gleaned from the program. Control flow is also a foundation upon which to build an understanding of program data flow.

Data Flow Information

Basic to a discussion of data flow analysis are the terms definition and use. A variable definition can modify the value of the variable, as by an assignment or a READ statement. A variable is used if it is referenced without modification, as by a WRITE statement or when it is the operand of a computation.

Three concepts of data flow analysis are built upon knowledge of program control flow and variable definitions and uses: (1) reaching definitions, (2) live variables, and (3) use-definition chains. The first concept, reaching definitions, a term used by Hecht, describes the problem of determining the sets of variable definitions that can "reach" the top of each node in an inter-block control flow graph. The second concept, live variables, deals with determining the set of variables that may be used after control passes to a given point in the control flow graph. The third concept, use-definition chains, involves linking the definitions of variables to their uses in the definitions of other variables. Variable uses are also linked backwards to the definition of that variable. This double linking from definition to uses and use to definitions forms a chain of data flow information which may be traversed for flow analysis.

Information which is basic to each of these data flow concepts is derived from knowledge of variable definitions and uses. The use and definition information may be derived from a syntactic analysis of program source code, allowing the sets of reaching definitions, live variables, and use-definition chains which describe inter-block data flow to be generated automatically.

In languages which provide a facility for modules, global data flow describes the inter-module flow of information through parameters and global data structures. When analyzing a program to determine inter-module data flow, it is necessary to capture the set of global data structures and parameters as well as their mode of passing.

Both inter-block and inter-module data flow analysis provide the maintenance programmer with knowledge of the flow of information through a program. Such knowledge is indispensable when attempting to understand how a program works. Data flow information also facilitates the tracking of program errors by providing the information flow which can be followed to find the location where program data goes bad.

Logical ripple effect tracing may be supported using use-definition chains to identify assumptions on variable values. This information along with inter-block control flow may be used to identify the set of program blocks which must be
examined for ripple effect problems for each potential modification.

Declaration Information

Declaration information is the lowest available level of syntactic information above that available from the source code itself. It includes the variable’s name and type. If the language provides a facility for declaring the variable’s valid range of values, initial value, or specific usage (i.e., in COBOL, the programmer is allowed to declare the variable as either computational or display), this information should also be recorded to provide the maintainer with information on the structure and initial status of the variable. Variable aliases should be listed in the declaration information in order to determine correct data flow information.

The variable’s defining module should also be noted. This, along with scoping rules for the language (possibly recorded as the list of modules that the variable is visible within) will provide the maintenance programmer with information on where the variable may be modified throughout the program.

Module declaration information includes the module name, type (if defined), and parameters. For parameters, information on name, type, and method of passing is also needed. A full set of declaration information for all objects defined within the module should include variables, other modules, constants user-defined types, labels, and external files. This information provides the maintenance programmer with an understanding of the environment created by the module.

Information on module visibility is also included in the declaration information. This entails not only what modules are visible to the given module, but what other modules can "see" and therefore invoke the given module. This information is the basis for defining the module-declaring hierarchy.

Constant names and their corresponding values are basic to the declaration information to be recorded for constants. Identification of the module that the constant is declared in provides the programmer with information on the visibility of the constant and which modules may therefore reference the constant. It is also necessary to capture the locations of references to the constant to provide the maintainer with information on what areas of the program will be affected if the constant is modified.

Information on types provides the maintainer with a set of templates for identifying variable structures. Information on user-defined types should include the name of the type and the definition of the type, including complete definitions of all subfields and indexes.

Label name and location form the basis of label declaration information. The visibility of the label should be included to provide the programmer with a reference on what program areas may jump to the label. Information on jumps to the label is available from the control flow information.

Declaration information is useful to the programmer when trying to establish a knowledge of a program’s naming conventions and data structures. Declaration information tells the programmer what the data structures used by the program look like and where they are used, but does not tell the programmer how the structures are used or what information they may contain at any time.

Every program contains a wealth of syntactic information. The template for cross reference information at the end of this paper does not attempt to identify all existing cross reference information, but focuses on that information which is particularly useful to the maintenance programmer for program support. It should be noted that the set of information available from any given program is language dependent.

THE ME2 SYNTACTIC ANALYZER

To demonstrate the utility of a maintenance environment, a prototype is currently under development at Arizona State University. As part of this environment, a syntactic analyzer has been developed to analyze Pascal source code for information useful to the maintenance task. The analyzer accepts Pascal source code that has already been verified by a compiler for syntactic correctness, and outputs information to the knowledge base. The information produced is similar to that represented in the templates. The user may then interact with the knowledge base to access information for the purpose of understanding or modifying a program, or analyzing a program for potential ripple effect problems.

CONCLUSIONS AND FUTURE RESEARCH

The prototype syntactic analyzer implemented as part of the maintenance environment currently under development at Arizona State University demonstrates the feasibility of evaluating source code for maintenance-specific information. The utility of this information is clear in view of the need for consistent, correct, and current program information for support of the maintenance tasks of program understanding, program modification, and analyzing a given modification for potential ripple effect. The implementation of this analyzer has shown that performance of the maintenance task may be greatly enhanced at relatively low cost by providing the programmer with an automated means for acquiring information necessary to perform the maintenance task.

Our future research efforts are centered upon developing other aspects of the maintenance environment including expansion of the knowledge base to contain semantic information and the building of a powerful graphic user interface.

MAINTENANCE INFORMATION TEMPLATES

Control Flow Template

1. Identification of blocks and modules
   a. Basic block identification. Basic blocks are sequences of consecutive instructions that are always executed from start to finish.
   b. Module identification. Modules are characterized as the subroutine facility provided by the language. If no subroutine facility is available, only program blocks may be identified.
   c. Process identification. Processes are units of independently executing code. They may be identified by defining requestor and server modules within the program.
2. Identification of inter-block control flow. This information may be shown using the graphing technique described by M. S. Hecht.

3. Identification of inter-module control flow. This information may be captured in a module-calling network. Recursive modules and the existence of abstractions, macros or library routines in the network should be noted, along with their interfaces.

4. Identification of inter-process control flow. This includes identifying corresponding requestor and server tasks. Exact rendezvous sequencing information may not be derived from program syntax since it is determined at run time.

Data Flow Template

Note: completion of the control flow template for a program is a prerequisite to identification of data flow.

1. Basic data flow identification
   a. Variable definitions. These are locations where the variable may be modified.
   b. Variable uses. These are variable references without modification.

2. Inter-block data flow
   a. Identification of the set of reaching definitions. These are variable definitions that can reach a given block due to the existence of definition clear-paths from the definition to the beginning of the block. This information may be collected by identifying all variable definitions within a block and then establishing whether the definitions reach the subsequent block by determining if there is a definition clear-path from the definition to the end of the block.
   b. Identification of live variables. These are variables which reach a given block and are also used within that block. This information may be collected by intersecting the set of reaching definitions with the set of variables used before being redefined by the given block.
   c. Identification of use-definition chains. These connect variable uses to definitions and vice-versa. This information is collected by associating every use of a variable with the most recent definition of that variable, and associating every definition of a variable with the uses of the variable that depend on that definition.
   d. Identification of ASSERTs. This includes identifying the assertion to be satisfied and its location in the program. This information, along with inter-block control flow, is used for ripple effect analysis.

3. Inter-module data flow
   a. Identification of parameters and method of parameter passing. This includes position of parameter in an argument list and is language dependent.
   b. Identification of non-local, non-parameter variables (global variables) which are used by a module.

Declarations Template

1. Variable Information
   a. Variable name.

b. Variable's defining module. Include scoping information as it pertains to the variable.

c. Variable type. If record-type, include subfield definitions; if array, include index types; for pointers, include access types.

d. Variable range of valid values (for languages which provide this facility).

e. Variable initial value (if available from definition).

f. Variable aliases. For each alias indicate where in the program the alias is used.

g. Variable usage. (i.e., COBOL classifies variables as either computational or display).

2. Module information
   a. Module name.
   b. Module parent or defining module. Include information on nesting and visibility.
   c. Module type (if defined).
   d. Module parameters. Name and position plus a full set of variable cross reference information for each parameter.
   e. Module locally defined entities. Full set of cross reference information for each variable, module, constant, type, file or label.
   f. Generic instantiation. If module is a generic instantiation, as provided by Ada, what is the generic instantiated and what types are given to the parameters?
   g. Overloading. Does module represent overloading of a system-defined function or operator?
   h. Abstractions. If module represents an abstraction, what are the defined operations on the abstraction?

3. Constant information
   a. Constant name.
   b. Constant defining module. Include scoping information as it pertains to the constant.
   c. Constant value. If constant is a deferred evaluation expression, note expression.
   d. Reference locations. Include all locations where a constant is used.

4. Overloaded operators
   a. New name assigned to operator. If a new function is assigned to the operator, detail it under module information.
   b. Type defining module. Include scoping information.
   c. Type definition. Define types for all subfields and indexes. Note if type is a derived type. In Ada, a derived type represents a variable type with the same structure and name as an existing type, however, for the purposes of type checking, a derived type is considered unique.
   d. Subtypes. If type is a subtype indicate parent type.

5. User-defined types information
   a. Type name.
   b. Type defining module. Include scoping information.
   c. Type definition. Define types for all subfields and indexes. Note if type is a derived type.
   d. Subtypes. If type is a subtype indicate parent type.

6. Labels information
   a. Label name.
   b. Label location.
   c. Label visibility information.

7. External files information
   a. File name.
   b. File type. Full description of type needed.
c. **File used for input, output or both.** Include locations of opens and closes for each use.

d. **File access information (random or sequential).** If random, detail locations of file pointer manipulations.

e. **Device file is assigned to, if available.**

f. **File sentinel, if defined** (as in COBOL).

**REFERENCES**


