An interactive software maintenance environment*

by STEPHEN S. YAU, SYING-SYANG LIU, and SHEAUSONG YANG
Northwestern University
Evanston, Illinois

ABSTRACT

In this paper, an interactive software maintenance environment is presented in which software maintenance tools, such as a syntax-directed editor, a pretty-printer, control and data flow analyzers, a data flow anomaly detector, a program slicer, and logical and performance ripple effect analyzers, are integrated together for effective software maintenance. The environment is based on a unified program representation model which is constructed by a pair syntactic-semantic tree. The advantage of this environment is that it allows software maintenance tools to use the common information which is supported by the database manager for the environment. The communications among different software maintenance tools are high because each maintenance tool can retrieve data from the common database. An experimental system has been implemented to demonstrate this interactive software maintenance environment.

*This work was supported by the Office of Naval Research under Contract N00014-80-C-0167.
INTRODUCTION

It is well known that software maintenance has become the dominant factor of the high cost of software, and software maintenance costs are still increasing. Maintenance is frequently performed on a large-scale software system because of the existence of system errors, changes of operating environment, code optimization, functional enhancement, deletion of obsolete features, and improvement of efficiency. Because of lack of effective maintenance techniques, the reliability of software systems likely deteriorates as more maintenance activities are performed on the systems. Consequently, the systems soon become unmaintainable and hence unusable. An effective approach to reducing the high cost of software and increasing the useful life of many software systems is to establish a software maintenance environment that would facilitate the proper use of various techniques and tools for effective maintenance.

Traditionally, a maintenance tool included in a software maintenance environment operates on the basis that the necessary program information needed by the tool is extracted from the program. The disadvantage of this approach is that a large number of special software packages must be written to extract the necessary information from the program and organize it in various forms required by the maintenance tools. The information generated by some software maintenance tools is frequently used by other software maintenance tools. For example, the graphical program flow generator, data flow anomaly detector, and program slicer need control and data flow information which has already been generated by the control and data flow analyzers. The communications among software maintenance tools in this type of environment are usually very low because one translator is required between each ordered pair of related maintenance tools. In this case, it is better to directly extract the information from the program under maintenance than to construct a translator between each ordered pair of related maintenance tools. Since maintenance tools are usually developed continuously and independently of the software maintenance environment, the total system for software maintenance would be huge, expensive, and difficult to control and maintain.

In this paper, we present a software maintenance environment based on a unified program representation model that facilitates the integration and interface of various software maintenance tools. The advantage of this environment is that it allows the software maintenance tools to use the common information that is supported by the database manager for the environment. The communications among different software maintenance tools are high because each maintenance tool can retrieve data from the common database. An experimental system has been implemented to demonstrate this interactive software maintenance environment, and has shown that the productivity for software maintenance using this environment is improved by a significant order of magnitude.

MAINTENANCE OF LARGE-SCALE SOFTWARE SYSTEMS

Before we present our software maintenance environment, let us review the maintenance process for large-scale software, which involves several phases and can be illustrated as shown in Figure 1.

The first phase determines the overall maintenance objectives. The second phase consists of analyzing a program to understand it. This phase is affected by the complexity, documentation, and self-descriptiveness of the program. The third phase consists of generating a particular modification proposal to accomplish the implementation of the maintenance objective. This phase is affected by the extensibility of the program. The fourth phase consists of accounting for the ripple effect. The primary attribute affecting ripple effect is
the stability of the program; the stability of a program is defined as the resistance to the amplification of changes in the program. The fifth phase consists of revalidating the modified program to ensure it has at least the same reliability as before. This phase is affected by the testability of the program. The whole or part of phases one through five is repeated until the modified software system passes the test.

The tools in our maintenance environment are used for software maintenance as follows: At the beginning, a graphical program flow generator is invoked to generate a graphical view of the software system under maintenance so that the system hierarchical structure can easily be understood. At the same time, a pretty-printer displays the structured program source code on the terminal. After locating the program information and deciding how to achieve the maintenance objectives, the user will use a syntax-directed editor to modify the program. During modification, other software maintenance tools may be invoked by pressing a function key to analyze program flow, detect data flow anomalies, slice the program, and accommodate ripple effect. Revalidate modified program, and compute important metrics whenever it is necessary.

MAJOR FEATURES OF THE ENVIRONMENT

Our interactive software maintenance environment is shown in Figure 2. The major features of the environment include a modified compiler, a syntax-directed editor, control and data flow generators, a graphical program flow generator, a data flow anomaly detector and a program slicer. In addition, a database manager is used to provide the interface between software maintenance tools and the database. All these software maintenance tools can be used interactively.

Database Manager

To support a large-scale software maintenance system, we need a database to store the whole syntactic and semantic information of the program to be maintained. The database manager provides an effective way to access the database and supports the interface for various software maintenance tools.

Modified Compiler

The purpose of the modified compiler is to translate the program source code to the internal syntactic and semantic structures and store them in the database. Various software maintenance tools can access the database through the database manager.

Modified Syntax-Directed Editor and Pretty-Printer

The pretty-printer (PP) and syntax-directed editor (SDE) offer an environment for creating and manipulating VAX-11 PASCAL programs. (VAX-11 PASCAL is an extension of standard ANSI PASCAL with the separate compilation capability.) The PP and SDE environment provides an easy-to-use editor that promotes step-wise refinement of PASCAL programs by simulating the program conception at a high level of abstraction. All input is type-checked for syntactic and semantic correctness and the program is automatically indented to emphasize the program structure.

The major features of SDE and PP include:

- A menu-driven facility to make SDE and PP user-friendly.
- A structured program representation facility for online modification and understanding.
- Using program templates to enforce the user to type in a well-structured and syntax-correct program.

Control and Data Flow Generators

When a program is input, control and data flow generators generate control flow and data flow information which is used by the graphical program flow generator, data flow anomaly detector, and program slicer.

Graphical Program Flow Generator

The graphical program flow generator draws the system hierarchical structures including the relation between modules, subprograms, and statements that are shown in Figure 3. With the aid of the graphical hierarchical structures, a user
can take a graphical view at different levels of the program and understand its control and data flow information. Such information is helpful for understanding and modifying the program. Figure 4 shows the hierarchical view of the system-module level on a graphics terminal.

Data Flow Anomaly Detector

Data flow anomaly includes defined-undefined, undefined-referenced, and defined-defined anomalies which indicate possible program errors. The data flow anomaly detector displays data flow anomaly on the terminal whenever it is invoked.

Program Slicer

The program slicer decomposes a program into a reduced program subset with respect to some statement and variables. Since a reduced program subset is greatly smaller than an entire program, modification and debugging are easier and more efficient to perform on reduced program subsets than on a whole program.

SOFTWARE MAINTENANCE USING THE ENVIRONMENT

A user may use the commands provided by the syntax-directed editor to create a new program or modify an existing program. Cursor movement can be node-oriented, line-oriented, or page-oriented depending on the user's request. Movement is controlled by moving terminal keys, such as, [], [], [], or by specifying a line number to jump to a destined line. To make any modification, a user first moves the cursor to a destined program node and then types in the correct command and code.

As shown in Table I, the whole program template is displayed on the screen when a user creates a new program. To find the features of all keys, the user may press PF2 to invoke the help menu screen as shown in Table II. Starting from

### TABLE I—A program template

```plaintext
program <Identifier> (<External-File-List>); 
{ Declaration Part } 
{ LABEL Declaration } 
{ CONST Declaration } 
{ TYPE Declaration } 
{ VAR Declaration } 
{ MODULE Declaration } 
begin 
{Statement-List} 
extend. { Of program }
```

### TABLE II—A key feature menu

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>insert template</td>
</tr>
<tr>
<td>D</td>
<td>delete a node</td>
</tr>
<tr>
<td>R</td>
<td>replace a node</td>
</tr>
<tr>
<td>F</td>
<td>go to parent node</td>
</tr>
<tr>
<td>S</td>
<td>go to first child</td>
</tr>
<tr>
<td>P</td>
<td>go to previous sibling</td>
</tr>
<tr>
<td>N</td>
<td>go to next sibling</td>
</tr>
<tr>
<td>DELETE</td>
<td>delete line feed</td>
</tr>
<tr>
<td>RETURN</td>
<td>insert line feed</td>
</tr>
<tr>
<td>↑</td>
<td>to 'up' node</td>
</tr>
<tr>
<td>↓</td>
<td>to 'down' node</td>
</tr>
<tr>
<td>←</td>
<td>to 'left' node</td>
</tr>
<tr>
<td>→</td>
<td>to 'right' node</td>
</tr>
<tr>
<td>move section down</td>
<td>move section up</td>
</tr>
<tr>
<td>move command on/off</td>
<td>move command before</td>
</tr>
<tr>
<td>move command on/off</td>
<td>move command after/</td>
</tr>
<tr>
<td>move command on/off</td>
<td>move command enter/</td>
</tr>
<tr>
<td>move command on/off</td>
<td>move command leave/</td>
</tr>
<tr>
<td>move command on/off</td>
<td>move command edit/</td>
</tr>
<tr>
<td>move command on/off</td>
<td>move command show/</td>
</tr>
<tr>
<td>move command on/off</td>
<td>move command edit/</td>
</tr>
<tr>
<td>move command on/off</td>
<td>move command mode/</td>
</tr>
</tbody>
</table>

*From the collection of the Computer History Museum (www.computerhistory.org)*
Table I, the following steps would allow the user to key in a program similar to the program shown in Table III.

1. Move the cursor to the (identifier) by pressing \[\text{ENTER}\]. Press \[\text{ENTER}\] to enter edit-mode and edit (identifier) to program name “Max” and then press \[\text{ENTER}\] to leave edit-mode. Repeat this step to edit (File-Identifier) to “(input, output).”

2. Press \[\text{F1}\] to move the cursor to “[LABEL Declaration]”-node and press \[\text{D}\] to delete this node. Repeat this step to delete “[CONST Declaration]”-node and “[TYPE Declaration]”-node.

3. Now, the cursor is positioned on “[VAR Declaration]”-node. This node will be replaced by a new node “var (Identifier-list):(Type-Specification);” by pressing \[\text{R}\]. Entering edit-mode by pressing \[\text{ENTER}\], the text “a,b,c: integer” is input.

4. Repeating steps 1 to 3, the whole program will be input through the aid of screen online menu-driven features which provide all the possible selections. For example, when the cursor is positioned on “[Statement-List]”-node and \[\text{R}\] is pressed, then a menu, as shown in Table IV, is displayed. After selection, for example “D. If-Then-Else,” this menu will disappear, and “[Statement-List]” will be replaced by an “if”-statement template.

To perform program slicing, we need to specify the software module name, the starting statement, and variables. The graphical program flow generator is used to improve program understanding and facilitate program analysis by displaying program flow information on a color graphics terminal.

THE UNIFIED MODEL OF THE ENVIRONMENT

In this section, the unified model for the environment is described. Hierarchical graph model (HGM)\(^9\) and Typed tree representation (TTR)\(^10\) are two unified program representation models for incremental modification. HGM is based on the concepts of recursive graphs and Codd relations to represent a program. The major disadvantage of HGM is that the internal program representation structure, which is in a relational form, is inconsistent with the program hierarchical structure, which is the interface between the user and the syntax-directed editor. Therefore, each movement on the program hierarchical tree needs several relational tables to reconstruct the program tree and display it on the screen. TTR is a program tree representation, but the tree also contains semantic information. The disadvantage of TTR is that the syntactic and semantic information of a program is combined together and associated with the nodes of the tree. In this case, the internal tree structure is still inconsistent with the program structure displayed on the screen. The syntax-directed editor and pretty-printer have to reformat the positions of nodes on the screen. Obviously, the implementation of software maintenance tools based on HGM or TTR will be more difficult and complicated than on a model which is consistent between the internal program representation and the external representation from the user’s view.

The unified model used in our environment is a pair of syntactic-semantic tree which consists of two parts: (1) a syntactic tree for incremental program modification and (2) a semantic tree for the storage and retrieval of semantic and flow information for various software maintenance tools. The syntactic information and semantic information are stored independently, but connected by several pointers for syntactic and semantic checking whenever it is necessary. To define a structure that is suitable for interactive modification, the program hierarchical structure should be maintained and should be consistent between system internal view for retrieval and external view for display. The syntactic and semantic structures of our system can be described as follows: A program is composed of two parts: (1) declaration parts, including “label,” “constant,” “type,” “variable,” and “procedure and function” declaration and (2) statement parts, including assignment statement, procedure statement, go-to statement, compound statement, conditional statement, repetitive statement, and with statement. Syntactic nodes are defined through the whole program, such as program header, type declaration, variable declaration, and statement in statement parts. Because the control and data flows exist in statement parts only, flow information is only defined in semantic nodes for statement parts. For each statement, variables in the “assignment” and “for” statements and expressions in various statements, such as “conditional” statements and “for” statements, are considered as semantic nodes. A syntactic or semantic node can be a simple node, such as “expression” node and “go-to” node, or a structured node, such as “com-

| A. Module-Call | G. For-To |
| B. Label | H. For-Downto |
| C. Assignment | I. With-Do |
| D. If-Then-Else | J. Case |
| E. Repeat-Until | K. Goto |
| F. While-Do | L. Compound |
| X. Quit |
pound” node, “conditional” node, “repetitive” node and “with” node.

In order to have good system performance, semantic nodes and syntactic nodes are stored independently except that some pointers indicate their relations. The relation of each pair of a semantic node and a syntactic node is illustrated in Figure 5 and can be described as follows: (1) There are two pointers from the head and from the tail of a syntactic node referring back to its semantic node. (2) There is one pointer from a semantic node point to its syntactic node. Figure 6 shows the semantic node structures, syntactic node structures, and their relationship for six common statements—“assignment,” “compound,” “if,” “while,” “repeat,” and “for” statements.

The cursor is positioned on a syntactic node, which is exactly the same as the cursor on the program source code displayed on a terminal. If a user modifies the text; for example, the superscript in “while” statement, then it can be done in syntactic structure by moving and editing key features. After modification, the updated information will be inherited to related semantic nodes for semantic checking and flow computation. The flow information will be associated with semantic nodes. If a user tries to add a new node, update or replace an old node, then the system will locate the semantic meaning from the semantic node, determine the necessary action, and then tell the user what to do by displaying a menu on the screen. Therefore, the syntactic and semantic information can be handled independently and be connected together whenever it is necessary.

The following example explains the data set stored in semantic nodes, which is used in graphical program flow gen-

![Figure 5](image)

**Figure 5—The relation between a syntactic node and a semantic node**

![Figure 6](image)

**Figure 6—The structure of semantic nodes and syntactic nodes**

From the collection of the Computer History Museum (www.computerhistory.org)
eration, data flow anomaly detection, program slicing, and logical and ripple effect analyses. Consider a statement “if \( a > b \) then \( c := a \) else \( c := b \)” and assume that \( D_i(n) \) and \( D_o(n) \) denote the input and output data sets of node \( n \) respectively. Then, we have the following equations:

\[
\begin{align*}
D_i(\text{exp}) &= \{a, b\} \\
D_i(\text{stmt}_j) &= \{a\} \\
D_o(\text{stmt}_1) &= \{c\} \\
D_i(\text{stmt}_2) &= \{b\} \\
D_o(\text{stmt}_2) &= \{c\} \\
\end{align*}
\]

IMPLEMENTATION AND EXPERIMENTS

So far, we have integrated the following tools in our environment: a syntax-directed editor, a pretty-printer, control and data flow generators, a graphical program flow generator, a data flow anomaly detector, a program slicer, and part of a modified compiler. We have not yet integrated logical and performance ripple effect analyzers in the environment; however, even with only the tools already integrated, we have experienced major improvement in productivity of performing software maintenance. Comparing our experimental results, shown in Table V, for maintaining a program of about 2,000 lines with and without using the environment, we notice that the improvement is in a significant order of magnitude. Larger programs will also be experimented after we complete the modified compiler so that large programs can be automated, input, and formatted in the environment.

The current environment is being implemented in a DEC VAX/VMS 11/785 using PASCAL. The necessary space used by our model is about eight times the space required for the program source code. The number of nodes is proportional to the number of lines of the program source code. In the worst case, the space occupied by data set in semantic nodes is \( O(cd) \), where \( c \) is the number of lines of the program source code, and \( d \) is the number of identifiers in the program. Since a program usually contains many subprograms and each subprogram has few identifiers, the space used by the data set is usually much less than \( O(cd) \). In practice, the space used by the data set is usually close to \( O(c) \). However, the data set can be calculated dynamically instead of being stored in the semantic nodes if the size of the memory space causes any implementation problem. As usual, there is a tradeoff between the amount of memory space used and the execution time.

CONCLUSION

Based on the unified model, various software maintenance tools can be integrated to form an effective software maintenance environment. The major advantage of this environment is that a user works with a two-dimensional, graphical representation of a program, instead of a linear text string representation. Specifying a program as a two-dimensional structure results in better understanding and easier modification of the program, which reduce the time and effort for software maintenance. At present, we have integrated several software maintenance tools in our environment including a

<table>
<thead>
<tr>
<th>TABLE V—Some experimental results for using the software maintenance environment</th>
<th>without our tools (in minutes)</th>
<th>with our tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>Understanding system structure</td>
<td>60</td>
<td>15</td>
</tr>
<tr>
<td>Algorithm analysis</td>
<td>same</td>
<td>same</td>
</tr>
<tr>
<td>Adding new feature (each local area)</td>
<td>(assume the adding process is compiling error, linking error, correct code)</td>
<td>(assume the adding process is compiling error, linking error, correct code)</td>
</tr>
<tr>
<td>* Typing added text</td>
<td>same</td>
<td>same</td>
</tr>
<tr>
<td>* Compiling, linking, and correcting errors</td>
<td>(10 + 10 + 3) (assume correct at 3rd time)</td>
<td>(10 + 10 + 3) (assume correct at 3rd time)</td>
</tr>
<tr>
<td>Debugging run-time error</td>
<td>(assume the correction process is guessing error, modifying, correct code)</td>
<td>(assume the correction process is guessing error, modifying, correct code)</td>
</tr>
<tr>
<td>* locating error codes</td>
<td>same</td>
<td>same</td>
</tr>
<tr>
<td>* understanding statements</td>
<td>(60 + 60 + 30 = 150)</td>
<td>((10 + 6) \times 3 = 48)</td>
</tr>
<tr>
<td>* debugging error codes</td>
<td>same</td>
<td>same</td>
</tr>
<tr>
<td>* recompiling and linking</td>
<td>(0 + 13 + 3 = 16)</td>
<td>(0 + 0 + 3 = 3)</td>
</tr>
</tbody>
</table>
syntax-directed editor, a pretty-printer, a graphical program flow generator, control and data flow generators, a data flow anomaly detector, and a program slicer. We are integrating logical and performance ripple effect analyzers in the environment. We plan to use artificial intelligence techniques to develop a software maintenance tools synthesizer to construct software maintenance tools automatically. 13

REFERENCES
