A Systematic Approach to Corrective Maintenance

KAMYAR JAMBOR-SADEGHI, MOHAMMAD A. KETABCHI, JUNJIE CHUE AND M. GHIASSI

Object Technology Laboratory, School of Engineering, Santa Clara University, Santa Clara, CA 95053, USA

A process-driven, model-based solution to corrective maintenance is described. The solution approach starts by identifying the set of ordered steps that should be performed to complete a corrective maintenance task. Once the steps in the process are clearly defined, the information needed at each step is organized into maintenance information models. A set of tools that operate on the models and provide the capabilities needed for the process of corrective maintenance are then identified. Realizing the models and providing the tools through a uniform interface lead to a software maintenance system that supports effective and reliable corrective maintenance. An overview of SAMS, a Software Analysis and Maintenance System developed based on this approach, is presented. SAMS integrates various tools that are needed to support maintenance processes including the corrective maintenance process. SAMS tools are developed on top of an object model of maintenance information realized using an object-oriented database management system.

Received February 1 1994, revised June 6 1994

1. INTRODUCTION

Corrective maintenance is initiated as a response to a software bug discovered during the operation of the software system and is completed when the bug is fixed and the software system is demonstrated to perform its intended functionalities correctly. Bugs, which are symptoms of system malfunction, are a common occurrence in large and complex systems. The key to effective corrective maintenance is identification of the code segment responsible for the bug.

Corrective maintenance is difficult because:

1. Bug reports often only describe the malfunction at a high level of functionality. It is difficult to establish the link between a bug and the code responsible for it in large and complex systems using a bug report.
2. Bug information is often ambiguous and lacks necessary details.
3. Due to hidden dependencies fixing a bug may introduce new bugs.

As shown in Figure 1, some of these difficulties are due to the large information gap that exists between the code and the bug.

The traditional corrective maintenance process consists of three phases: *locating* the bug, *fixing* it and then making sure no undesirable side-effects have been introduced. To locate the bug, software maintainers start by analyzing the bug information, then they try to understand the code, and finally establish association between the bug and the smallest portion of the code responsible for it. To fix the bug, software maintainers start by designing the changes to the buggy code, modify the buggy code, and finally testing to make sure that the bug is corrected and the functionality of the rest of the system is unaffected. The steps to support the corrective maintenance process is shown in Figure 2.

Software maintainers spend most of their time searching to locate the buggy code and then testing the fix and the entire software system to ensure that no regression is introduced (Sedlmeyer *et al.*, 1983). Step 4, fixing the bug is relatively easy. However, without a software analysis and maintenance system, the software maintainers must complete steps 1, 2, and 3 in Figure 2 to build a cognitive model of expected behavior based on code and existing documents. The cognitive model that is developed in this fashion is used to fill the gap between the code and bug, as shown in Figure 3.

The task of developing the cognitive model heavily relies on software understanding. Understanding software can be approached from two directions (Seviora, 1987). In the code-driven (or bottom up) approach, the programmer starts with the code and forms an abstract description of what the individual parts of the program do and then what the entire program does. In the functionality-driven (or top-down) approach, the programmer knows the specifications for the program. Using programming experience, the programmer forms hypotheses of the overall structure of the program and refines them to a point where they can be verified against the code. Top-down and bottom-up approaches are usually used together and iteratively.

Building this cognitive model for large and complex software systems is difficult. Furthermore, in the absence of correct and up-to-date documentation of software such cognitive models are often incomplete and inconsistent and may even be incorrect. Further complications

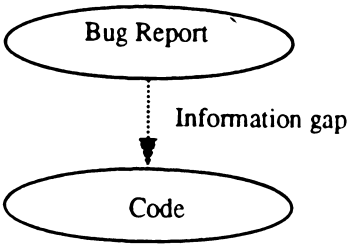

FIGURE 1. Bug report to code information gap.

are introduced when cognitive models developed by different maintainers working on a corrective maintenance task differ. Finally, there is very little chance that the knowledge and experiences gained in the process of fixing a bug by one maintainer is shared in subsequent maintenance activities by others. Several knowledge-based software debugging systems have been developed and they can be categorized into three groups based on the approach used for understanding and debugging software.

In the program-analysis approach the programmer would read the program, analyze it and try to understand it. In the process, the programmer would check whether the program is consistent with the specifications. If a discrepancy between the specifications and the code is discovered, the programmer would try to debug it. Proust (Johnson and Soloway, 1985) and PUDSY (Murray, 1980) are examples of knowledge-based systems that have adopted this approach.

In the I/O-based approach the program is executed with carefully selected inputs. If the outputs deviate from what is expected, the programmer examines the deviations to find clues to identify or at least approximate location of the bug. Falosy (Sedlmeyer *et al.*, 1983) is an example of a knowledge-based system that has adopted this approach.

In the internal-trace-based approach the programmer decides what to observe and examines the information collected to obtain further clues about the bug's location. The fault-localization process may proceed through several stages with increasingly finer resolution until the discrepancies between the expected and the observed behavior suggest plausible bug hypothesis. MTA (Gupta and Seviora, 1984) is an example of a knowledge-based system that has adopted this approach.

The idea of developing a comprehensive knowledge base to capture corrective maintenance knowledge to fill the bug to code gap is good. However due to the following reasons, building such a comprehensive knowledge base is difficult.

1. Analyze the bug report to understand the nature of malfunction.
2. Develop an understanding of the software.
3. Based on information gathered in steps 1 and 2 establish association between the bug and the code.
4. Design changes and modify the software to correct the bug.
5. Test to make sure the bug is fixed.
6. Test to make sure all other functionalities are working properly.

FIGURE 2. General process for corrective maintenance.

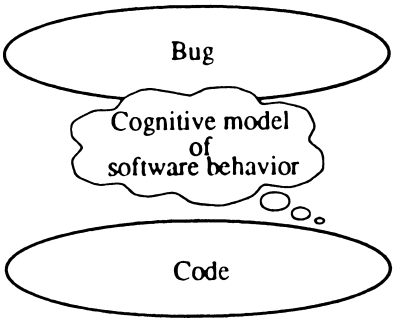

FIGURE 3. Software maintainers' cognitive model to fill the gap between bug and code.

1. Acquiring knowledge related to application domain, software domain, and debugging is difficult. This knowledge must be acquired through frustrating process of interviewing the different domain experts, as well as a comprehensive program analysis.
2. Representation of acquired knowledge in a coherent and usable format is difficult due to the richness and the variety of the knowledge needed in the corrective maintenance process.
3. Applying the knowledge acquired to identify and fix bugs in a large software system is a heuristic process. Automation of this heuristic process is difficult.

Our intention is neither to build a comprehensive knowledge base nor to automate the process of corrective maintenance. Instead, our goal is to develop an integrated system for software maintainers that allows sharing of information about the software system among maintainers by supporting various views of the software system and by providing a set of tools that simplify performing and verifying maintenance activities. The proposed system allows integration of the various aspects of the software system. The system facilitates analysis of the various aspects (source code, test suite, documentation and build procedures) and provides the necessary tools to establish logical associations among components of each aspect as well as related components across these aspects. For example, our approach supports analysis and presentation of the software application at both the structural and functional levels at various levels of abstraction. In the testing of software system, for example, our approach supports logical organization of the test suites (white box versus black box test suite) and provides the necessary tools to establish an association (logical links) between a test case and the segments of the code under test. All this information are maintained by an object-oriented repository which is the central component of the system. Availability of such a system will enable software maintainers to:

1. Form abstract models of the software system at various granularity ranging from the very high level

block representation of file, module and function levels all the way to the code level.
2. Establish association among the components of these abstract models and provide tools necessary for traversing among the various levels.
3. Localize a bug to its smallest possible segment of the code.
4. Identify all other portions of the application that may have been affected by fixing of a bug.
5. Form a regression test suite that will quickly verify and validate correctness of any bug fix.
6. Record the maintainers experience on line for future use.

The notion of developing a repository which maybe a database or a knowledge base has been used in other aspects of software maintenance such as software understanding, design recovery, reuse and reverse-engineering. Desire (Biggerstaff, 1989) is a system which attempts to build a knowledge base of conceptual abstractions and relationships among them. CIA (Chen *et al.*, 1990) is a tool for analyzing program structure. The system extracts information from C programs and stores the information in a relational database. The REFINE (Reasoning Systems, 1985) system allows users to define domain-specific languages by defining grammars that serve as a mapping between objects in the knowledge base and their textual forms.

This paper is organized as follows. Section 2 describes the corrective maintenance process, and the information and tools necessary to support it. Section 3 presents a system that provides the information and tools through an integrated, easy to use interface. Section 4 presents the concluding remarks.

2. SYSTEMATIC APPROACH TO CORRECTIVE MAINTENANCE

We have taken a systematic approach to the development of a system to support corrective maintenance. The approach starts by carefully analyzing the corrective maintenance activities and develops a model of corrective maintenance. Based on this model, we develop a process for corrective maintenance. This process consists of a set of ordered steps that should be performed to complete a corrective maintenance task. The information needed at each step is identified and organized into information models. A set of tools that operate on the information models are identified. Realizing the information models and providing the tools through a uniform interface lead to a software maintenance system that supports corrective maintenance.

2.1. Corrective maintenance model

Our model of the corrective maintenance process is shown in Figure 4. Software is developed as implementation of a set of requirements which constitutes its *intended functionality*. For a variety of reasons, in the process of *forward engineering* of a system some information present in the original requirements is lost, misinterpreted or miscoded resulting in functionality that does not exactly meet the intended requirements. Identifying the discrepancies between the *intended* and *actual functionality* of a system is a key factor in effective corrective maintenance. To identify such discrepancies a *reverse-engineering* process, to derive the actual functionality of the system, is needed. The actual functionality of the software system is organized into a hierarchy where each component in the functionality hierarchy achieves its task through its components at the lower levels. Information that is derived during the process of reverse engineering allows for building the mapping between the code and its actual functionality.

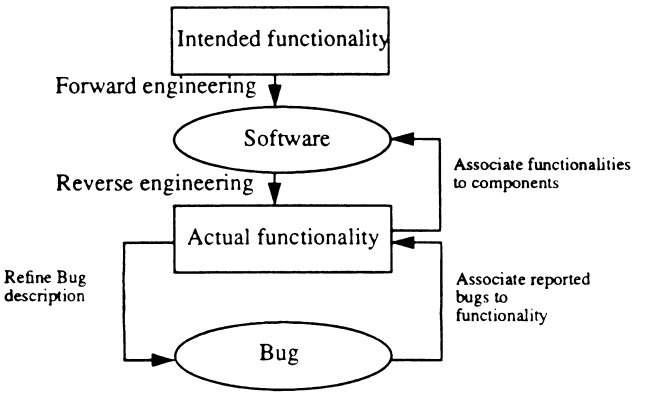

FIGURE 4. Model of corrective maintenance.

The multiple levels of functionality abstraction allow for *bug* to functionality association to occur in an iterative fashion with each step resulting in a more *refined* and *concise* definition of the bug. This association can also be extended to include the test suites as well. Once the bug description is refined to a point where it can be clearly associated with a functionality object, the software components associated with the bug could be easily identified.

2.2. Corrective maintenance process

Using the corrective maintenance model presented in Figure 4, a complete process which should be followed for a corrective maintenance task is shown in Figure 5. This process is the refinement of the process shown in Figure 2.

The process assumes that a hierarchical description of software has been developed. It further assumes that each node in the hierarchy is associated with an execution path in the software. Developing the functionality hierarchy and establishing the associations among functionality, execution path, and structural components of software are described in the following subsections. A careful analysis of the steps listed in Figure 5 leads to the identification and characterization of the required information and the supporting tools for corrective maintenance. An additional benefit of developing this process for corrective maintenance is that it serves as a

1. Query bug based on a given criteria (e.g. bug severity).
2. Identify and select the bug based on the result of the query of step 1.
3. Browse the description of the bug to better understand the faulty functionality.
4. Browse the actual functionality hierarchy for a match of the bug description.
5. If a functionality description matching the bug is found and no subfunctionality exists, then goto step 8.
6. Since the bug description does not fully and uniquely describe the faulty functionality, localize the bug to the smallest subfunctionality by ranking the subfunctionalities using test coverage and complexity metrics and testing each subfunctionality.
7. Modify the description of the original bug to reflect the refinements; goto step 4.
8. Identify the execution paths that realize the defective functionality.
9. Test the structural components (and the interface among them) in the execution paths identified in step 8 to further localize the bug to the smallest set of structural components that still displays the bug. Use test coverage and complexity metrics to identify components that were inadequately tested (most likely candidates).
10. Determine how the structural components identified in step 9 are to be modified.
11. Check out the structural components identified in step 9.
12. Identify all other functionalities that may have been affected. Keep this information for regression testing and add it to the history file of the product.
13. Identify the set of existing test cases associated with the affected functionalities.
14. Identify the set of existing test cases associated with the structural components related to the affected functionalities.
15. Collect test cases identified in steps 13 and 14 to form the 'local regression test suite'.
16. Modify the components checked out in step 11. Eliminate undesirable side-effects.
17. Test and validate that the bug is fixed and the buggy behavior is no longer present.
18. Test to make sure all other functionality is working properly by running the regression test suite.
19. Check in the modified components.
20. Modify the actual functionality to reflect the changes.
21. Change status of bug in the database to 'fixed'.

FIGURE 5. Refined process for corrective maintenance.

starting point for developing Corrective Maintenance Metrics. If we are able to compute the cost of operations performed in each step, the number of times each step is executed and the amount of information required by each of the steps within the process, then a cost function for corrective maintenance can be developed. The cost function takes the information such as the size of functionality and test suites used in the process as input and returns a total estimated cost of conducting corrective maintenance activities. Corrective maintenance metrics bases on this approach is currently being developed.

2.3. Required information for the corrective maintenance process

Figure 6 illustrates the information and tools that are needed to support the process described in Figure 5. The information is grouped into five information models: *Structure information, functionality information, Execution Path information, Bug information* and *Testcase information* which are shown as bold rectangles in the figure. Tools which are represented as rectangles in the figure operate on the information models. The solid labeled arrows represent the information and control flows, while the gray lines establish the relationship between information models and steps in the process. The numbers on upper left corner of the boxes refer to the steps in Figure 5.

The five types of information identified are not independent from each other. They are interrelated. Especially, the relationships between structure and functionality are very critical to corrective maintenance because such relationships form the mapping of software structure and functionality. Figure 7 illustrates the relationships between these information models.

Execution paths play an important role in linking software structure and software functionality. Execution paths are derived from structure model automatically. Functionality objects are associated with execution paths. Testcases are associated with functionality objects which can be used for blackbox testing.

In the following we describe the information content of structure, functionality, execution path, bug, and testcase models respectively. The information content of each model is described in form of a table. The table has three columns: a column for the field name, a column for describing the field and a column for relating the information associated with the field to the step(s) of the process of Figure 5. We use an example later in Section 2.4 to illustrate the need for the information that is contained within each model.

2.3.1. Structure model

Structure model of a software system consists of the components or building blocks that make up that system together with the part-of (composition) relationships. These building blocks are referred to as structural components. Table 1 lists the information in the structure model of a software system.

The structure model has two components. The language independent component which is fixed and defined once for all supported languages, and the language specific component which differs from

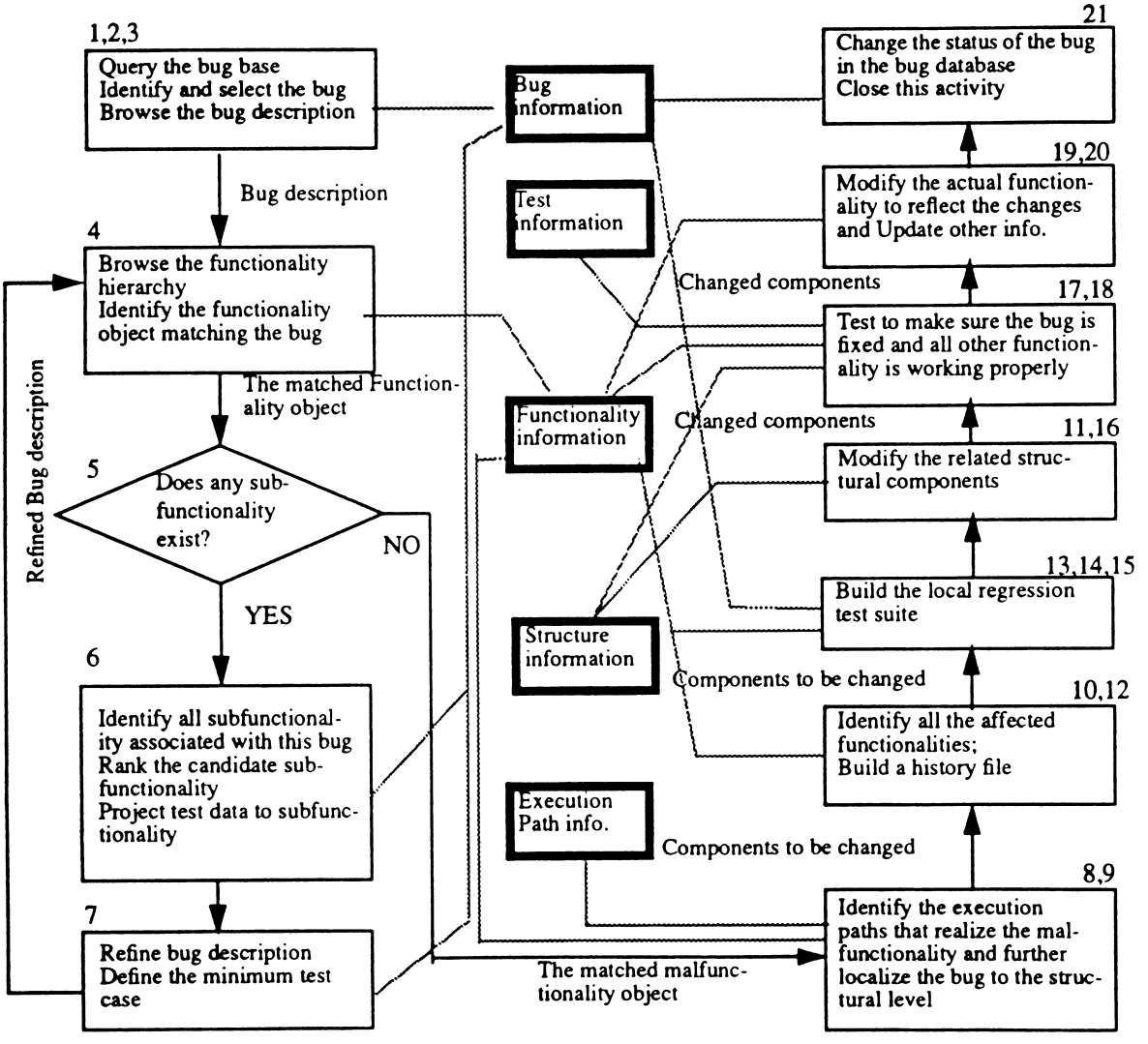

FIGURE 6. Information flow in the corrective maintenance process.

language to language. The specification of the programming languages can be used to derive the language specific component in the model. The specification can be in BNF extended with semantics or in attributed grammar (Aho *et al.*, 1986). (Ketabchi, 1993b) describes the generation of object models from formal language specification in detail. The structure model contains the

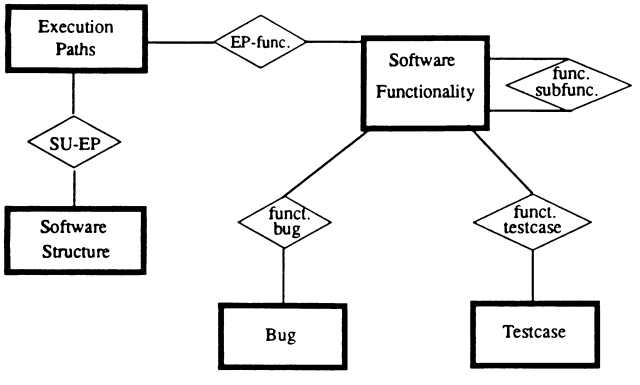

FIGURE 7. The relationships between information models in corrective maintenance.

part-of (component) relationship. The schema is object-oriented and consists of class hierarchy. All data declarations, definitions, and references are available as attributes of the associated components in the structure model. This information is used to generate control and data flow information.

The structure model is the largest of the models. To support operations such as reformatting a function, detailed information at the token level needs to be stored. For a large software system the amount of information generated could be up to 40 times the size of the original code. To manage the volume and complexity of the data a database management system must be used.

2.3.2. *Functionality model*

The functionality model consists of functionality objects organized into a functionality/subfunctionality hierarchy. A functionality has component subfunctionality that together realize the functionality. A functionality object in addition to subfunctionalities has a name, a description and is associated with an execution path model, an execution path may have sub-execution paths

TABLE 1. Structure information

Field name	*Description*	*Step*
Id	a unique identifier for the structural components	10,16
Type	type of the structural component. e.g. function definition	10,16
Container	parent object in the structure hierarchy	10,16
Components	children objects in the structure hierarchy	10,16
Functionality	set of related functionality objects	10,16
Annotation	additional comments associated with the structural component	10,16

which at the lowest level are composed of structure units. The subfunctionalities of a functionality are associated with the sub-execution paths of its associated execution path. The functionality hierarchy of a software system, therefore can be viewed as a higher-level abstraction of the execution paths of that software. This abstraction can easily be understood by the software users and engineers and can easily be mapped to the code.

In addition to execution path, bug reports and test case objects can also be associated with the functionality objects. These associations can easily be established by the users of the software and can be utilized by maintainers to localize bugs and to select the appropriate test cases when needed. Table 2 shows the attributes of the functionality objects.

The functionality hierarchy also plays an important role in data flow analysis. Since a functionality is associated with an execution path which in turn is composed of structure units, the variables referenced or modified in a functionality can easily be found by accessing the structure units from functionality object through execution paths.

2.2.3. *Execution path model*

The building blocks of execution paths are constructs that are supported by the implementation language of the software. Some constructs are simple building blocks because they do not change the flow of execution. An example of this type of construct is assignment of a value to a variable. Since these constructs do not have interesting properties with respect to execution path, the consecutive sequences of these constructs are logically treated as a single component in the execution path model. Non-branching constructs become significant when the data flow is analyzed along execution paths. Some constructs fork branches in the execution paths. An example of this type of construct is `if` statement. It is not difficult to identify this second type of constructs, which we will refer to as *branching*, from the first type of constructs, which we will refer to as *non-branching*. Branching constructs are of interest in computing the execution paths because they are sources of new paths.

The branching constructs can be identified by examining the implementation language of the software. For instance, the list of branching constructs in C language is shown in Figure 8.

Execution path model is derived from the structure model. The structure model is automatically derived from the specifications of the programming language. An execution path can be generated for a single statement, a function or the entire program. The model of the programming language is consulted to decide how the execution path should be generated for each structural component. A structural component for which execution paths are described, has a *begin* point which represents the source of all the paths and has one or more *end* points which represent the destinations of all the paths through the structural sub-components. The paths are then generated based on the type of the component object. For each branching construct (those shown in Figure 8), the model contains a 'template' which enumerates the paths that are generated by that construct. Figure 9 represents the template for the if-then-else construct. This template produces two major paths namely 'Begin-1-2-End', and 'Begin-1-3-End'.

Function calls are treated as special constructs and have representations at two levels. For these constructs the representation at the high level is non-branching (a single statement representing the function call), while the representation at the detailed (or expanded) level maybe

TABLE 2. Functionality information

Attribute	*Description*	*Step*
Name	name assigned to functionality	4,8,12
Description	textual description of the functionality object	4,8,12
Bug list	the associated bug(s) if the functionality is not properly implemented	6
Container	parent of the functionality object in the functionality hierarchy	4,8,12
Components	children of the functionality object in the functionality hierarchy	4,8,12
Execution paths	set of structural components that achieve the functionality. This set consists of statements which form execution paths.	4,8,12
Testcases	the set of testcases that exercise all execution paths related to the functionality	9,13,15

```
conditional statement:
      if (expression) statement
      if (expression) statement else statement
switch-statement:
       switch (expression) statement
iterative-statement:
      for (expression; expression; expression)
                                    statement
      while (expression) statement
      do statement while (expression)
goto-statement:
      goto identifier:
continue-statement:
      continue;
break-statement:
      break;
return-statement:
      return expression;
```

FIGURE 8. Branching constructs in C language.

branching. (System-defined functions are always non-branching and cannot be expanded.)

Figure 10 shows an example execution path. Paths are labeled as `f1`, `f2`,... and the structural components within paths are labeled as `SU1`, `SU2`,.... Paths that have common begin and end points are grouped together in a rectangle. The paths confined to a rectangle correspond to subfunctionalities that have the same parent functionality and are at the same level. Each rectangle is considered a node in the containing path at a higher level. To develop a functionality hierarchy software maintainer is presented by the execution path of the software similar to the one shown in Figure 10.

The maintainer can expand or collapse the paths and groups of parallel paths as he/she desires. The maintainer selects the paths in any desired order and assigns a name and a description to it. These names and descriptions are the properties of functionality objects that are created and organized by the system into a hierarchy that parallels the execution path hierarchy. Figure 11 shows an example of functionality hierarchy developed using the execution path shown in Figure 10.

Table 3 lists the attributes of the execution path objects.

2.3.4. *Bug model*

A bug describes a system malfunction. Information contained in a bug description includes the bug identification attributes which are used to form queries

if-else-statement: if (expression) statement else statement

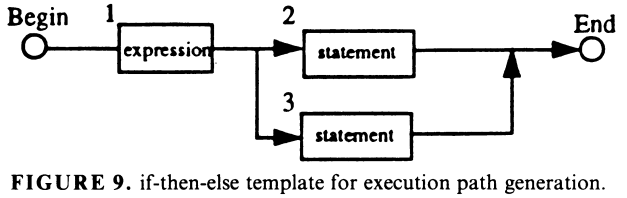

FIGURE 9. if-then-else template for execution path generation.

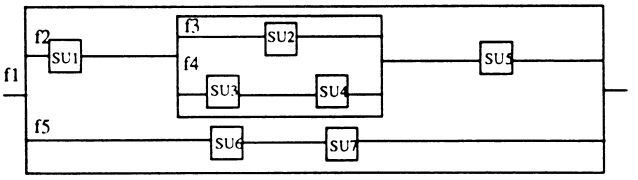

FIGURE 10. Abstract execution path representation.

to search the bug base and the malfunction description which is used to identify the functionality object that is related to the bug. Table 4 lists the attributes of the bug objects.

2.3.5. *Testcase model*

A bug that is fixed needs to be tested to verify correct behavior of the software system after the bug fix. Information contained in the testcase model includes the testcase identification attributes which are used to select testcases for execution. Table 5 lists the attributes of the testcase model.

2.4. Supporting tools for the corrective maintenance process

An integrated and complete set of tools that support the steps in the process of Figure 5 is necessary. The various tools required for the corrective maintenance process are derived from the operations that are performed in the various steps of the process. Table 6 summarizes the required tools that are identified following a careful analysis of the process of Figure 5. Column 1 in Table 6 identifies the main purpose of the tools, columns 2 and 3 together identify the elements of information that are used by tools, and column 4 identifies the output of applying the tools. Column 5 identifies the steps of the process that utilize the tools.

The bug browser allows for addition, deletion and query of bug information in the bug base. The testcase browser allows for addition, deletion and query of testcases associated with structural components and functionality objects. The functionality browser allows for addition, deletion and modification and browsing of the functionality objects. The process of creating functionality objects is semi automated. The system allows the software maintainer to examine candidate execution paths. Upon selection of a path, the user is prompted for the description and the labeling of the functionality that the path performs. A new functionality object is then created. Testcases could also be associated with functionality objects. A functionality object could

```
f1 <description of f1>
        f2 <description of f2>
                f3 <description of f3>
                f4 <description of f4>
        f5 <description of f5>
```

FIGURE 11. Functionality hierarchy derived from execution path of Figure 10.

TABLE 3. Execution path information.

Attribute	*Description*	*Step*
Id	a unique identifier which is used for selecting and locating an execution path	8,9,12
name	a unique name for the execution path	8,9,12
Container	the containing object which includes this execution path	8,9,12
Components	the structural units that makeup this execution path	8,9,12
description	a description of what the execution path achieves when executed	8,9,12

be removed if the execution paths associated with it are altered or removed. The addition, deletion and modification of structural components can be achieved automatically through the use of incremental compilers. The structure browser can be used to query various information on software structure such as complexity of a function or a statement, data references, data modifications, etc. As a browsing tool, the execution path browser can be used to show the execution paths that can be statically generated through structural analysis of the software system.

Additional tools (which are indirectly related to the steps of the process in Figure 5) are needed to support software understanding as well as help in fixing bugs. These tools are listed in Table 7. The call graph and data flow information is produced from the data model. The data model, which is derived from the structure model, captures information such as where data is used, modified or referenced. Since the data model is not directly related to the process of corrective maintenance presented in this paper, the details of its construction will not be presented here. The interested reader is referred to Sadeghi (1994).

2.5. An example

To illustrate the ideas described in this paper, a simple example program (about a 1000 lines of code, organized into four C files) is presented. The ideas could be applied with equal effectiveness to larger, more complex programs. The entire user interface code of SAMS, for example, which consists of about 80 000 lines of C code, has been loaded, and successfully maintained by SAMS.

A typical university registration application consists of *finance, course, student* and *professor* entities. A corrective maintenance activity is initiated as a request to fix a bug that has been discovered related to adding new students to the student list. The first step in the process is to construct a query to locate the bug that is assumed to have been submitted to the bug base. An effective way of constructing such a query is by filling a form similar to that shown in Figure 12. The key attributes for bug selection are presented as buttons. The users must provide a value for all the key attributes by selecting among one of the values presented as the button is activated. If more than a single bug matches the specified criteria, the users can page through the matched bugs using the retrieve button in succession. By reading the description of the bug (which is displayed at the bottom of the window of Figure 12) the maintainers acquire important clues necessary for successful identification of the offending functionality. In this example we assume that the bug has simply been put in the bug base and has not yet been associated to a functionality.

The next step is identification of the functionality that is related to the bug by browsing the functionality hierarchy. To identify the offending functionality the description of the bug is analyzed and keywords are identified. In our example, the related keywords are *student, list* and *add*. A search for functionality with names similar to these keywords, results in identification of the *Students* functionality. As shown in Figure 13, the *Students* functionality consists of three subfunctionalities, *Reports*, to generate a listing of the students in various forms, *Add*, to add new students, and *Delete* to delete existing students. These are four subfunctionalities associated with the *Add* functionality, *openDB*, to open the students database, *CheckVld*, to check validity of the students to be added (whether they are registered or not), *AddToList*, to add the new students to the list, and *CloseDB* to close the students database.

A further refinement of the bug description is

TABLE 4. Bug information.

Attribute	*Description*	*Step*
Bug Id	a unique identifier for the bug	1,3
Creation date	date of bug submission	1,3
Reported by	customer name who reported the bug	1,3
Assign Date	date bug is assigned to the person in charge of its fix	1,3
Assign to	user id of the person in charge of bug fix	1,3
Purpose	a short paragraph describing bug symptoms	1,3
Status	current status of the bug (investigating, open, close)	1,3
Severity	severity of the bug (high, medium, low)	1,3
Type	type of problem (code bug, document, feature, performance)	1,3

TABLE 5. Testcase model.

Attribute	*Description*	*Step*
Author	author of the testcase	17,18
CreateDate	date the testcase was created	17,18
ModifiedBy	person who modified the testcase	17,18
ModifiedDate	date testcase was modified	17,18
Purpose	a brief description of the testcase	17,18
HostOS	host operating system	17,18
HostProgram	host programming language	17,18
Product	software product name	17,18
SU	structure units object id	17,18
Functionality	functionality object associated with the testcase	17,18
Class	(correctness, regression, performance, conformance, error handling, quality)	17,18

necessary at this point to locate the exact structural components responsible for the incorrect functionality. The user is prompted for whether the bug is related to invalid entries in the list or missing entries. Note that if such verification is not possible, all four subfunctionalities need to be examined based on the code complexity of the functionality and its testcase coverage. Functionality objects of high complexity and low test coverage are usually selected first. In our example, further testing to reproduce the bug reveals that bug is actually related to existence of invalid students. Following the modification of the original bug to reflect the refinement of the bug description, the user is presented with the implementation details of the *checkVld* functionality.

Upon identification of the faulty paths, associated testcases can be analyzed and executed against the faulty paths to verify correct bug identification. Additional testcases may need to be developed if the original set of testcases fail to test the faulty paths. The next step is making necessary modifications to the software structure to correct the faulty paths, and executing the related testcases to ensure correct operation. If correcting the fault results in addition of new paths or removal of existing paths, new functionality objects may have to be created or old ones may have be deleted. It is generally assumed that the corrective maintenance does not alter the functionality hierarchy. If the software functionality is in any way affected during the corrective maintenance process, the process is considered to be an enhancement. A separate set of steps (which will not be discussed in this paper) need to be followed for software enhancement. The processes for such maintenance activities have been developed and are described in Sadeghi (1994).

3. SAMS — A SOFTWARE ANALYSIS AND MAINTENANCE PLATFORM TO SUPPORT CORRECTIVE MAINTENANCE PROCESS

SAMS (Ketabchi, 1989, 1990; Ghiassi *et al.*, 1992) is a multi-user, integrated software analysis and maintenance system developed based on the realization that the most

TABLE 6. Tools to support the corrective maintenance process

Functional description	*Information used*	*Operates on*	*Information produced*	*Step*
Query bug information database	bug selection criteria	bug base	list of matching bugs	1
Browse bug information	list of bugs	bug objects	bug description	3
Define/modify bug	bug information	bug object	bug object	21
Browse functionality information	functionality objects	functionality hierarchy	functionality object	4,5
Associate bug with functionality	bug object	functionality object	functionality object	6
Add/delete/modify bug	bug description	bug base	altered bug base	7
Browse execution path information	execution path information	execution path objects	analysis results	8
Complexity/test coverage analysis	test objects	structure object	analysis results	9
Version management	version object	structure object	version object	11,19
Add/delete/modify functionality	functionality description	functionality hierarchy	altered functionality hierarchy	20
Add/delete/modify structure	structure component	structure hierarch	altered structure hierarchy	16
Associate functionality with structure components that form an executive path	functionality object	structure object	altered functionality to structure mapping	12
Associate execution paths with functionality	structure object	functionality object	altered functionality to structure mapping	8
Associate test case with the structure	test object	structure object	mapping between test case and software	14,15
Associate test case with functionality	test object	functionality object	mapping between test case and software	13,15
Test structural components to verify correct behavior	expected results	structure object	actual results	17,18

TABLE 7. Additional tools to facilitate software understanding

Functional description	*Information used*	*Operates on*	*Information produced*
Generate call graph	structure components	data objects	call graph
Generate data flow	structure components	data objects	data flow
Generate control flow	structure components	data objects	control flow
Annotate structural component	textual description	structure object	annotated structure object

difficult part of creating an effective software analysis and maintenance system is not the creation of maintenance tools but rather the development of a platform that allows for effective *integration* of the necessary information, and the tools that operate on the information to support various software maintenance processes. One objective of SAMS project was implementing such a platform by developing an object-oriented model of software. Our model represents software as an interconnected assemblies of objects that form a hierarchy. Software analysis and maintenance tools are implemented as methods of the different classes of the objects in the hierarchy. Attributes associated with the different classes allow for derivation of all the different aspects of software at various levels of detail. SAMS' user interface allows for a uniform, and intuitive interaction between the user and the information. Upon selection of an object (through the various SAMS' browsers) the user is presented with a set of available operations on the selected object. The set of operations is presented in the form of a menu whose entries depend on the type of the selected object. A brief description of SAMS platform is provided in the next sub-sections. This platform provides the backbone for implementation of the software maintenance models described in Figure 7 of Section 2.3.

3.1. Architecture of SAMS

The SAMS prototype consists of three major subsystems: *Graphical user-interface, Software database generator* and *Software database*, as shown in Figure 14.

The primary goal in the design of SAMS' interface has been to provide a uniform object-oriented user interface. This uniformity of the user interface minimizes users' interactions with the system and thus reduces the amount of new information that a user needs to know in order to utilize the system effectively. Users interact with objects which have well defined interfaces; everything appearing on the screen is an object; any object can be selected by double-clicking on the mouse button; once an object is selected, the menu associated with the object shows all valid operations associated with that object. Multiple tools can be active at the same time and multiple objects can be browsed and analyzed simultaneously.

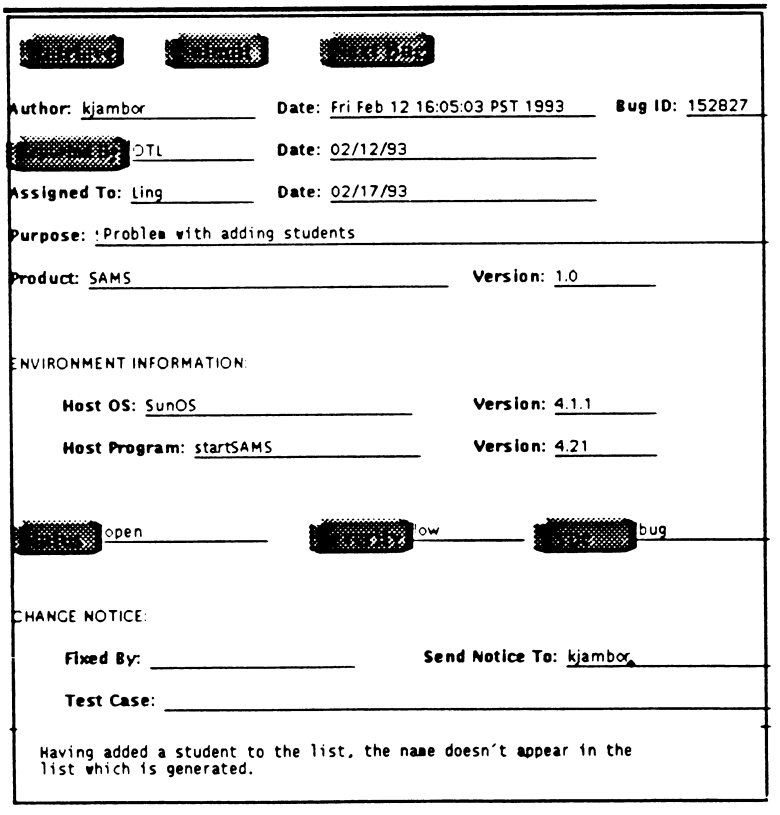

FIGURE 12. Bug query and selection process.

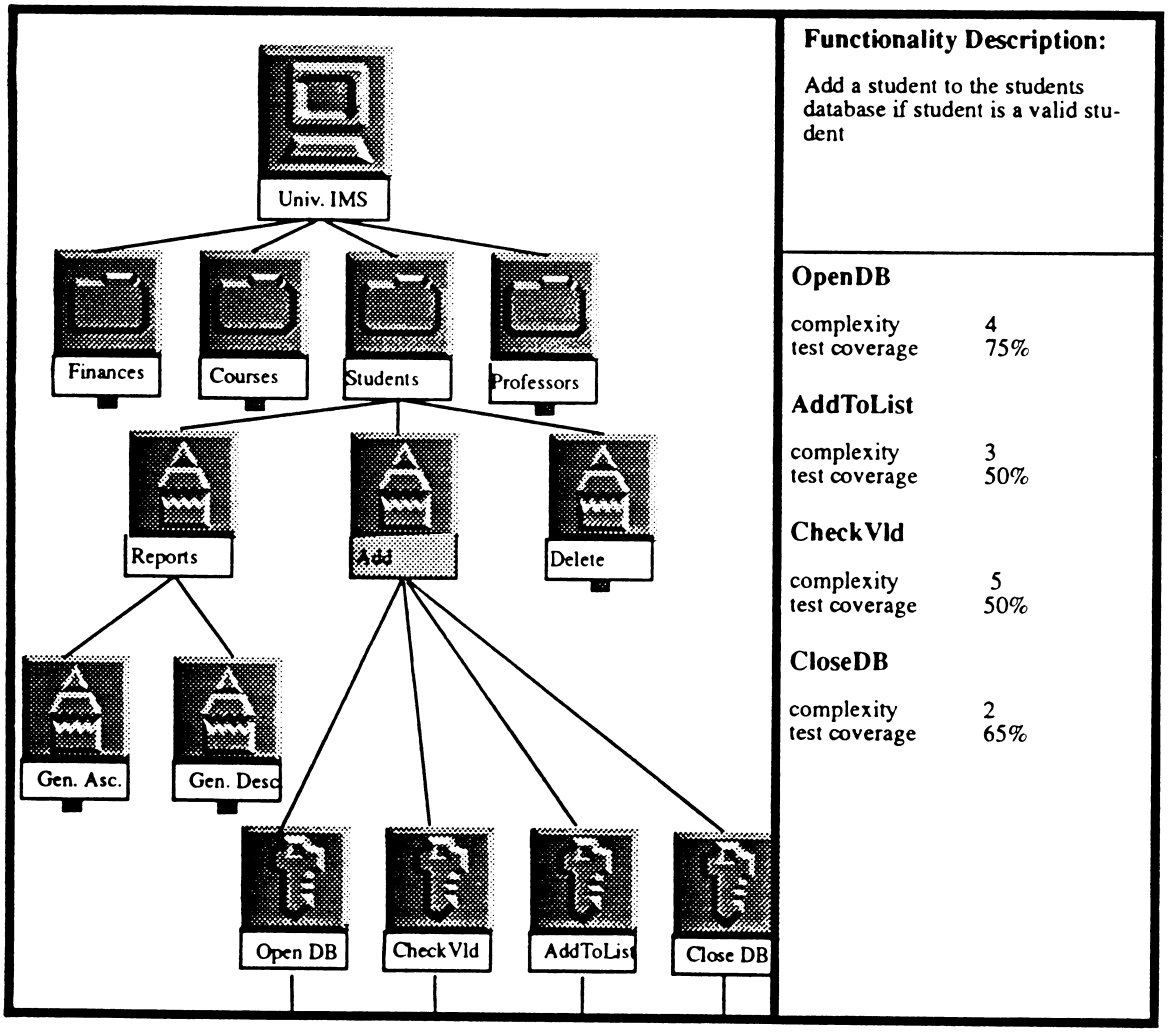

FIGURE 13. Functionality browser.

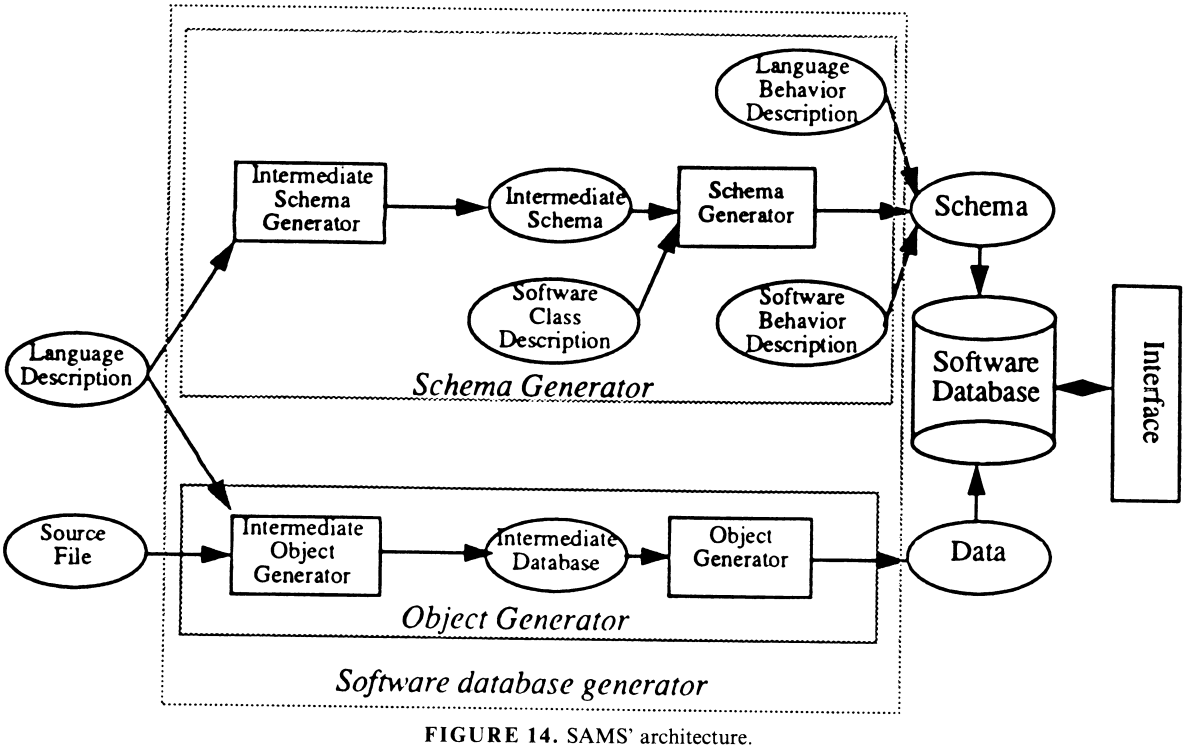

FIGURE 14. SAMS' architecture.

The SAMS' software database generator consists of a *schema generator* and an *object generator*. The schema generator is responsible for creating the SAMS database schema. The schema consists of a language-independent component and a language specific component. The language-independent component captures elements common to all software systems regardless of the implementation language. These elements include configuration, version control, functionality, bug and testcase. The language-specific component describes the elements of the implementation languages of software systems (such as functions, statements, variables, etc.) and the inter-relationships among these elements. The Schema of SAMS consists of approximately 400 classes. Of the 400 classes about 320 describe the C and Make language constructs. The remaining 80 classes support the language-independent features of software. The separation of language specific and language independent components of the schema simplifies support for multiple languages. The object generator is responsible for building object-oriented representation of the code. Understanding the syntax and semantics of the programming languages, SAMS is able to automatically populate the schema given the source of a program.

The layered architecture of SAMS reflects our attempt to achieve DBMS independence. The *intermediate schema generator* and the *intermediate object generator* are responsible for creating a DBMS independent schema and data referred to as *intermediate schema* and *intermediate database* respectively. The intermediate schema description language is an object-oriented language which supports class definition. It provides facilities to define class and instance attributes and operations. Because the part-of relationship is important, some attributes are marked as components. The intermediate database language provides facilities to represent the class instances and their relationships. A detailed description of the syntax of the intermediate schema and intermediate database languages is provided in Ketabchi and Sadeghi (1994). The intermediate schema and intermediate database are processed to generate Schema and data for a specific object-oriented DBMS [in the current prototype, GemStone (Bretl *et al.*, 1989) OODBMS is used]. To successfully generate methods that implement messages of objects, we need to provide functions that, upon invocation by the schema generator, generate the instance and class methods for the classes in the schema. These functions are provided by *language behavior description* and *software behavior description*. The *software class description* describes the language-independent elements of the schema.

The *software database* is a central repository shared by all tools. It stores not only the information derived from the source of the programs such as its structure, data flow and control flow, but also the information derived during the process of software analysis. SAMS tools are implemented by operations (messages and methods that implement them) associated with the different classes in the software database.

The process of building the software database starts by preparing the *language description, software class description, language behavior description* and *software behavior description*. The language description is used to generate *intermediate schema*. It is also used as the input to parser generator tools to generate the *intermediate object generator*. The intermediate schema is translated to the data definition of the target OODBMS. Having successfully loaded the schema, the software database can be populated via the object generator. The object generator produces an object-oriented representation of the source files from an abstract (DBMS independent) representation referred to as intermediate database. This representation is generated by the intermediate database generator. Once the objects representing the code in the source file are loaded into the software database, analysis tools can be applied to produce the various aspects of software at different levels of granularity.

3.2. SAMS facilities for supporting corrective maintenance process

Corrective maintenance process is one of the several maintenance processes supported by SAMS. As shown in Figure 6, the information needed for corrective maintenance are *structure, functionality, execution path, bug* and *testcase*. SAMS provides facilities for acquisition, representation, manipulation and browsing of such information.

The majority of the classes in the structure model describes the programming language objects such as statement, expression, function, declaration and their legal compositions according to the syntax and semantics of the language. These objects together with other objects such as File and Program in the software domain are the structural building blocks of software systems. The schema for programming language objects are derived from the specification of the languages automatically. The specification can be in BNF extended with semantics or in attributed grammar (Aho *et al.*, 1986). The schema for the structure model is populated automatically with the objects generated from the source code and Make files of the software system. Ketabchi (1993a) describes the generation of object models from formal language specification in detail.

Currently, SAMS provides two levels of structural abstraction to help manage the complex structure of software systems. We refer to these levels of abstractions as large granularity and small granularity structure. The structural components at large granularity structure are *applications, programs* and various kinds of *source files*. An application is defined to be a collection of programs. A program is in turn defined to be a collection of source files. The facilities of the large granularity structure browser provide a convenient way for browsing and understanding the configuration of software systems of

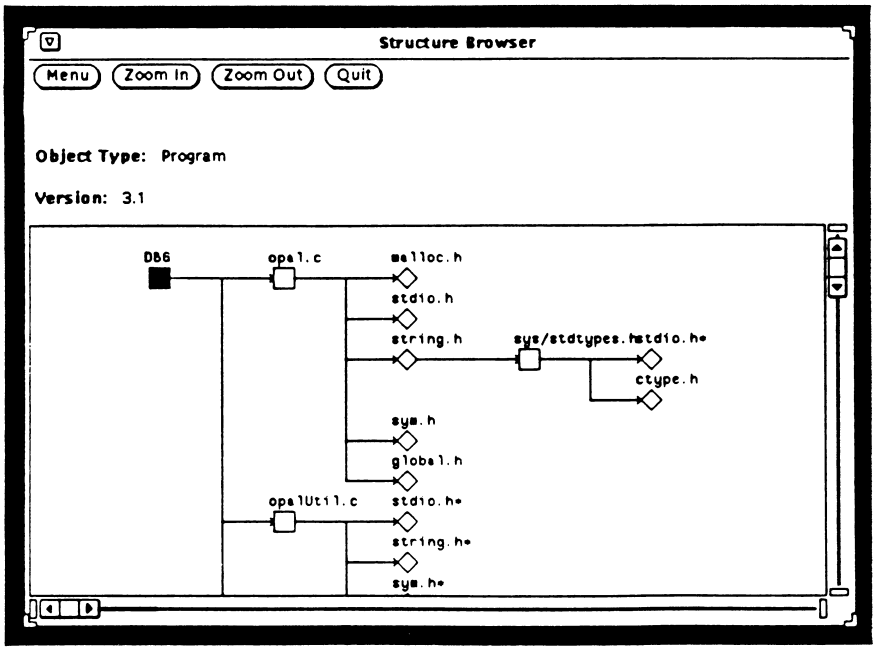

FIGURE 15. Large granularity structure browser.

varying degrees of complexity. Figure 15 illustrates the large granularity structure representation of a C program.

The structural components at the small granularity level are the language specific constructs that make up the source files. Identification of the syntactical entities of the system (such as functions, statements, variables, etc.) and understanding of the relationships among these entities are provided at this level. Currently SAMS supports application programs written in the C programming language, and the make facilities. Understanding the syntax and semantics of the C language, SAMS is able to automatically build the structure given the source of a C file. Figure 16 illustrates the small granularity structural representation of a C file (left window), annotation related to a selected component (right window) and the text of the selected component (lower window). Annotation refers to textual description of a software component. These descriptions are different from in-line comments in that they are not part of the source but are associated with the specific components by the analyzer of the software.

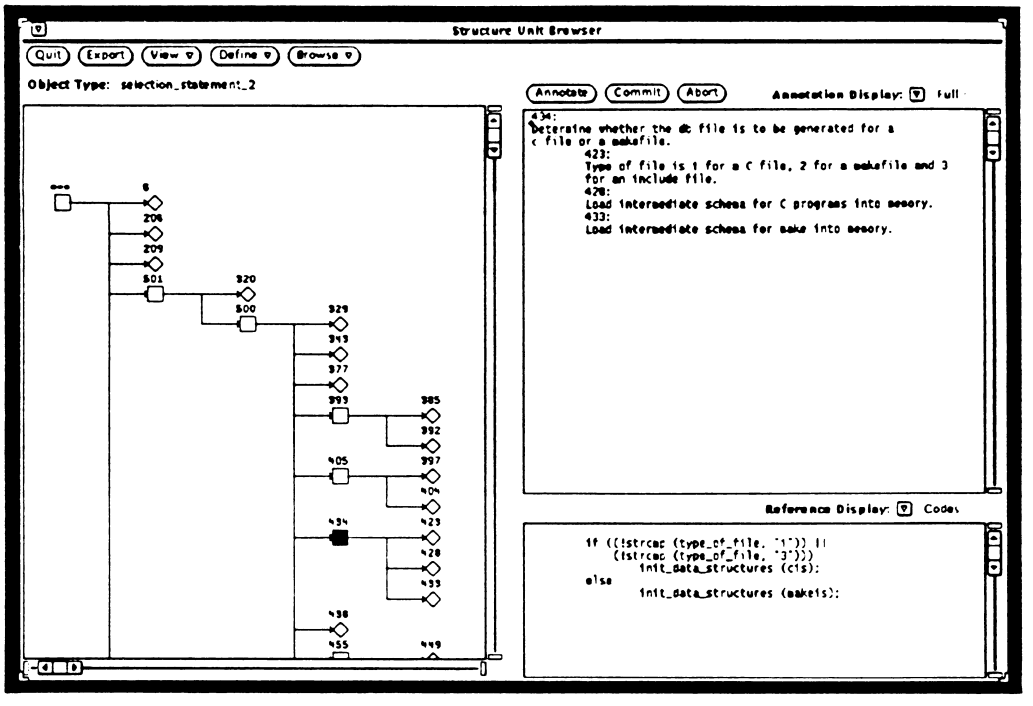

FIGURE 16. Small granularity structure browser.

SAMS provides a set of analysis facilities such as text browser, structure browser, execution path browser, call graph browser and cross reference browser to aid software maintainers to better understand the software system. SAMS integrates the results of the analysis using these facilities, annotates software components based on external requirements and functional specifications and captures the maintainers' understanding of the software's functionality as a functionality hierarchy. The functionality model, therefore, provides a mapping between requirements, functionality and structural components. Unlike the structure model which is built automatically, the functionality model is built by the software maintainers in interactive analysis sessions. Users of SAMS can select a functionality and ask for software components which achieve that functionality, or they can select a software component and ask for the functionalities in which that component participates.

SAMS stores all the execution paths that can be statically generated. Functionality and testcase objects can be associated with execution paths. Users can select an execution path and identify all functionality in which the path participates. All data which is referenced or modified within a path can be identified. And testcase associated with an execution path can be executed.

Bugs and testcases are represented as objects in SAMS which are created from the external information provided in bug reports and testcases submitted by the users of the software system. A set of tools related to the bug and testcase models are supported by SAMS which can provide the facilities such as bug and testcase objects creation, modification, retrieval, browsing, and bug and testcase association to functionality objects.

4. CONCLUSION

An effective way to handle the complexity of corrective maintenance is to formalize the corrective maintenance process. This has been the focus of this paper. We believe this approach applies equally well to other software maintenance processes (Chu *et al.*, 1993). We have prototyped a system called SAMS based on this approach. Our goal is to extend our approach and the prototype to develop a comprehensive software analysis and maintenance system which supports adaptive, preventive, perfective and extensive maintenance processes as well.

An important improvement to the corrective maintenance process we have presented in this paper is the association of bugs with the functionality of the software system and a rigorous maintenance process based on this association. The significance of this association is that it fills the gap between the functional description of the software and the code. This association allows maintainers to quickly identify the segments in software that are responsible for the defect. In addition to bugs, testcases can also be associated with functionality (black box testing) and structural components (white box testing) which can be used to not only verify correctness of changes, but also to aid software maintainers in localizing bugs.

ACKNOWLEDGEMENTS

The work on SAMS project has been funded by US West Advanced Technologies and Santa Clara University.

REFERENCES

Aho, A. V., Sethi, R. and Ullman, J. D., (1986) *Compilers Principles, Techniques and Tools*. Addison-Wesley, Reading, MA.

Adam, A. and Laurent, J. P. (1980) Laura, a system to debug student programs. *Artificial Intelligence*, **November**, 75–122.

Biggerstaff, T. J. (1989) Design recovery for maintenance and reuse. *IEEE Computer*, **July**, 36–49.

Bretl, R. *et al.* (1989) The GemStone data management system. In Kim, W. and Lochovsky, F. H. (eds.), *Object-Oriented Concepts, Database, and Applications*, pp. 283–308. Addison Wesley, Reading, MA.

Chen, Y., Nishimoto, M. Y. and Ramanmoorthy, C. V. (1991) The C information abstraction system. *IEEE Trans. Software Eng.* **16**,

Chu, J., Jambor-Sadeghi, K., Ghiassi, M. and Ketabchi, M. A. (1993) A systematic approach to adaptive maintenance. In *Proc. SURF '93* Orlando, FL.

Cousin, L. and Collofello, J. S. (1992) A task-based approach to improving the software maintenance process. In *IEEE Conf. Maintenance*, pp. 118–126.

Ghiassi, M., Ketabchi, M. A. and Sadeghi, K. J. (1992) An integrated software testing system based on an object-oriented DBMS. In *25th Proc. Hawaii Int. Conf. Software Sciences* Vol. 2, Hawaii.

Gupta, N. K. and Seviora, R. E. (1984) An expert system approach to real-time system debugging. In *Proc. First Conf. on Artificial Intelligence Applications*, pp. 336–343. CS Press, Los Alamitos, CA.

Harjani, D. and Queille, J. (1992) A process model for the maintenance of large space systems software In. *IEEE Conf. Software Maintenance*, pp. 127–136,

Johnson, W. L. and Soloway, E. (1985) Proust: knowledge-based program understanding. *IEEE Trans. Software Eng.*, 267–275.

Ketabchi, M. A. (1990) An object-oriented integrated software analysis and maintenance. In *Proc. Conf. on Software Maintenance*,

Ketabchi, M. A., Lewis, D., Dasananda, S., Lim, T., Roudsari, R., Shih, K., and Tan, J. (1989) Object-oriented database management support for software maintenance and reverse engineering. In *IEEE COMPCON*,

Ketabchi, M. A. (1993a) *Object Generation from Formal Language Description*. Technical Report, Santa Clara University, Santa Clara, CA.

Ketabchi, M. A. (1993b) *Object Generation from Formal Language Description*. Technical Report, Santa Clara University, Santa Clara, CA.

Ketabchi, M. A. and Sadeghi, K. (1994) *Applying Object Technology to Software Analysis and Maintenance*. Technical Report, Santa Clara University, Santa Clara, CA.

Lukey, F. J. (1989) Understanding and debugging programs. *Int. J. Man–Machine Studies*, 189–202.

Murray, W. R. (1989) Heuristic and formal methods in automatic program debugging In. *Proc. Ninth Int. Joint Conf. on Artificial Intelligence*, pp. 15–19. Morgan Kaufmann Publishing, Palo Alto, CA.

Reasoning Systems (1985) *REFINE™ User's Guide*. Reasoning Systems Inc.

Sadeghi, K. (1994) *A Comprehensive Solution for Software Analysis, Maintenance and Reverse Engineering*. PhD Thesis, Santa Clara University, Santa Clara, CA.

Sedlmeyer, R. L. *et al.* (1983) Knowledge-based fault localization in debugging. In *Proc. ACM SIGSoft/SIGPlan Software Engineering Symp. High-Level Debugging*, ACM, pp. 25–31.

Seviora, R. E. (1987) Knowledge-based program debugging systems. *IEEE Software*, **May**, 20–32.

Shahmehri, N., Kamkar, M. and Fritzson, P. (1990) Semi-automatic bug localization in software maintenance In. *IEEE Conf. on Software Maintenance*, pp. 30–36.