소프트웨어 유지보수 도구를 위한 자료 저장소의 설계

최 은 만[†]

요 약

성공적인 소프트웨어 개발 환경을 구축하기 위하여 도구 사이의 인터페이스가 잘 정의되어야 한다. 최근에 활발히 연구되는 CASE 도구들이 잘 결합하여 일관성 있는 환경이 되기 위해서는 자료 저장소의 설계가 중요하다. 특히 유지보수를 위한 도구 개발은 원시 코드의 변경과 프로그램으로 표현된 객체들 사이의 상호관계를 잘 나타내도록 자료 저장소를 설계하여야 한다. 본 논문에서는 소프트웨어 유지보수 환경 구축에 필요한 자료 저장소 설계에 통합 방법을 도입하였다. 유지보수 대상 객체들을 정의하고 객체들 사이의 관계를 나타내었으며 자료 저장소를 접근하기 위한 방법도 제안하였다. 이 방법에 의하면 유지보수하는 동안 변경을 관리하기 위한 버전 및 형상관리뿐만 아니라 질의 서비스, 자료교환 서비스가 효율적으로 이루어진다.

A Design of Data Repository for Software Maintenance Tools

Eun Man Choi[†]

ABSTRACT

It has been commonly accepted for a while that a successful tool environment must provide for smooth interfaces between its tools. Recent integrated CASE environment must be based on well-integrated data repository which supports a tightly coupled, consistent environment. For the maintenance tool, it requires a careful design of the maintenance chest's database. Information about a program exists in many different forms after analyses have been performed. It must be possible to associate and select objects for data repository from this information as necessary. This paper suggests a new integrated scheme for the data repository in building software maintenancec environment. The scheme provides many basic services, including storage and management of objects/entities and links/relations; version and configuration control, query service, data interchange service.

1. Introduction

One area of software engineering environments that has received relatively little attention is the database where the information used by the environment is kept. The reason is that most of the old tools had proprietary repository for their own use and didn't want to communicate each other. However, recent Integrated-CASE efforts[1, 2, 3] significantly influence the design of repository of today's CASE environment. The repository acts not only as a storehouse for knowledge gained about a program, but as the system controller and interface mechanism for the tool collection. As tools will evolve over time, the database must support the easy addition and removal of tools.

Software maintenance toolchest is a set of software tools that support program understanding, assess the impact of modification, rebuild code after alterations, and test to validate changes. It is based on incremental analysis of all data in the system including

† 정 회 원 : 동국대학교 컴퓨터공학과 교수
논문접수 : 1994년 10월 25일, 심사완료 : 1995년 3월 1일

source code, data flow graphs and metrics. Maintenance tools should provide a variety of analysis and viewing functions to assist in understanding and regression testing various aspects of code changes. Therefore, requirements for data repository of maintenance tools are different from those for the general CASE environment.

Existing data repository for software engineering environment were reviewed from the viewpoint of requirements for software maintenance tool's data repository. Data objects and data dependency required for building software maintenance tools were described and assessed in Chapter 2. This paper suggests the efficient combination of DBMS and software maintenance tools among the many possibilities. In addition, all the details in data scheme of repository were provided in Chapter 4. That kind of design strategy makes smooth interface among software maintenance tool fragments. Chapter 5 explains how the tool fragments access information stored in data repository, and how to hook up the whole mantenance toolchest.

2. Requirements for a Software Repository

The three major requirements of the software repository are incremental analysis, analysis on demand and reuse of existing tools and results. Incremental analysis means that as the source code being studied is changed, only the changes(deltas) are re-analyzed. This applies to the derived data tables as well as the source code changes. Each of the data objects in software maintenance tools have a 3-dimensional structure. Each instance of an object consists of the base structure and the deltas made to them as the underlying source code is changed as well as multiple instances of a data object. These deltas are also stored in the database and the analysis functions work on the deltas.

$$
\begin{array}{rl}
\text{S(Source)} & + \Delta S_1 + \Delta S_2 + \ldots + \Delta S_n \\
\text{ST(Structure Tree)} & + \Delta ST_1 + \Delta ST_2 + \ldots + \Delta ST_n \\
\text{DG(Data-flow Graph)} & + \Delta DG_1 + \Delta DG_2 + \ldots + \Delta DG_n \\
\text{CG(Control-flow Graph)} & + \Delta CG_1 + \Delta CG_2 + \ldots + \Delta CG_n \\
\text{TC(Test Case)} & + \Delta TC_1 + \Delta TC_2 + \ldots + \Delta TC_n \\
\cdot & \\
\cdot &
\end{array}
$$

(Fig. 1) Maintenance repository

The data in software maintenance repository should be analyzed horizontally or vertically. Horizontal analysis(based object plus deltas) reconstructs the current version of an object. Vertical analysis(delta from several objects) is used to construct a new version of some object from the current deltas of related objects.

Because the analysis function is expensive, the data objects in the software repository are not automatically updated when there is a source code change. Analysis on demand requires that the data be updated only when actually requested by a user. For example, if the user requests the current value of a metric, the database is queried and if the most recent source code delta is newer than the last update of the metric, the metric is recalculated from the data it is derived. This can cause a cascade of recalculation back through the dependency graph from the metric to the source code.

2.1 Required Data Objects

Software maintenance tools produce and use a variety of semantic information of the program. The user might want different kinds of information implicit in the text that must be detected in order to fully understand the program for maintenance. For example, repository should keep the control flow se-

mantic of the program which is required in maintenance. That provides test paths which satisfy certain condition. Another type of information contained in programs, the data flow of the program, concerns the changes or consistences in values associated with the names of program objects throughout the course of the program.

For modifications and updates, data dependency information is essential. From dependency information, change effect analysis can determine which source text is affected by a certain change. From nodes in the control flow graph, the slicer can find the structure tree that corresponds to that node. It can help to determine the source text that is interested in.

⟨Table 1⟩ represents data objects that are required for building software maintenance tools. There are two classes of data objects, base and delta object. The delta objects are those that are modified by changes produced by analyzing changes in the source code.

⟨Table 1⟩ Required Data Objects

Data Objects	Roles
Source delta	- keep the source code in delta unit with start and end line in editor - each delta action is stored as a single record in table - keep program's file name, version number
Structure tree	- store the tree node type, label, name - start line and column of node in editor - keep the pointer to parent and children's node
Chunk/Noteped	- store the chunk/noteped name, key field, description - keep the starting and ending line - chunk table has child pointer to keep containment relationship
Symbol table	- has symbol name, type, class, declaration line number - store information about scope of symbol
Usage	- contains the information concerning usage of the symbol - keep the reference lines and types(declaration, left-hand-side, right-hand-side, procedure call, procedure declaration)
Test coverage	- keep the nodes of certain test path
Test harness	- keep module names and parameters, stub or driver names
Test path	- keep all subpaths that would be possible in module
Test case	- test paths, test data
Control flow graph	- node type, label, name, parent, children nodes, start, and end line number
Data flow graph	- node type, label, name, symbol name, liveness, anomaly, def/use relationship
Linemap	- keep the line number and flag describing delete
Version number	- contain information on the largest version of data for given file name and object
Error message	- for consistency and maintenance in data repository
Operator counts	- for Halstead metric
Procedure	- procedure information(parameters, size)
Halstead metric	- Halstead metric data
Logical ripple effect	total interface
McCabe metric	McCabe complexity of module
Information flow	complexity by data flow

2.2 Data Dependency

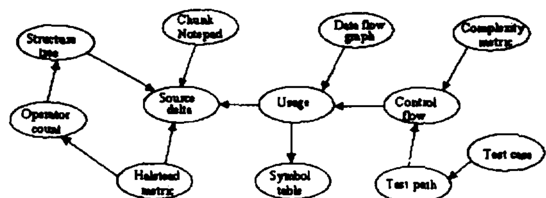
It is important task in designing data repository to figure out the relationship among data objects. There are few stand alone data objects in software maintenance. Most of data objects are generated based on the other data objects by analysis tools such as test case generator, dependency analyzer, complexity metric, data flow analyzer. For example, test path object comes from analysis of control flow object and test harness object.

As described earlier, data dependency is retrieved when changes make incremental analysis. For instance, if the programmer makes change certain part of code and wants to see the change effect, maintenance tools retrieve usage table first, and then check data dependency to find out what objects should be reanalyzed.

(Fig. 2) shows data dependency graph among data objects for software maintenance. Dependency relations has two types of semantic such as usage or part-of. Usage relationship is represented by single head arrow and part-of relationship by circle merged lines.

3. Control Issues in Software Maintenance Tools

Whenever an software maintenance tool requests data from the repository, the data may be out of date. The design philosophy of analysis on demand, rather than an soon as the data is available, is responsible for this. The knowledge of how to perform the up-



(Fig. 2) Data dependency among data objects

dates on the data has to be encoded somewhere in maintenance tools. One approach is to de-localize the control and have each of the tools know which of the others it must call to get the current values of each data structure. A second approach is to store the control knowledge in the database. Both of the these approaches are methods for encoding the knowledge contained in the data dependency diagram.

Centralized control has advantages such as small function coded in database, little knowledge about new tools, tool interface standards. But, adding new tool may required changes to repository. In de-centralized control, tools only need to know what data they need. However, tool uniformity is hard to enforce and simple data repository is not well-suited to this. A new tool developer must have knowledge of the internals of the other tools.

There are two reasons to prefer centralized control. The first is that a new developer need only study the data repository to know how and where to put new analysis modules. The second is that the analysis modules and the access code would be written as a number of small, single function programs. These are easier to understand and maintain.

A model of control that centralizes the activity is that of *active objects*[4]. This is a related technique to that of object-oriented design. Active objects are defined as 'an object in which a high degree of autonomous responsibility and control is vested'. In our context, the control of when data is updated is given to the data objects that need the data.

In [5], active objects called KNOs are defined. These are similar to the objects are defined in the software maintenance tools in that they are responsible for the use and creation of specific kinds of knowledge. They describe some of these autonomous objects as being marionettes in that they have string that are pulled by the user and in turn manipulate other objects. KNO can detect when data they are dependent on have changed and will update themselves. As interfaces between tools, active objects can act to convert data from the form in the repository to that needed by the tool. This can greatly ease the integration of existing tools in to the environment.

4. Design of the Data Repository

This chapter describe the specifications of the data repository design. It includes the specifications of the schema and the interface library that provides access to this data. Additionally, this chapter describes the notation used to describe the data repository, the conceptual framework of the interface library, the users view of the interface library, task assignments, contingency plans.

4.1 Delta Tables

The delta tables are those that are modified by changes produced by analyzing changes in the source code. They consist of the structures presented below with an action field that shows how the table entry was produced. The possible actions are Add, Replace, and Delete.

The entries in the delta tables are self-describing. Each has all the information needed to determine its correct position in the table within it. For example, the structure tree table contains entries that are nodes in the structure tree. Each node entry contains the node id of its parent and its children. Thus, the order of the table doesn't matter as the tree can be reconstructed from the information in the tree nodes.

(1) Source delta

All the other tables are derived from the source deltas and the source code. Each delta action is stored as a single record in the table. The attributes of source delta are the following.

srcdelta(action, file_name, version, start_line, num_lines, code)

(2) Structure tree

Several maintenance tools depends on a program structure tree. This tree is produced as needed by analysis tools and stored in the repository for use by the other tools. The tree structure follows the structure of the program and easy to obtain by parsing the code. It is also easy to modify by updating some pointers.

The structure tree is a doubly linked list of complex nodes, each of which represents a basic program structure. The top level node is the module. Each lower level is a complex node of a lower-level structure. The leaves are simple nodes which represent simple program structure such as assignment statements. The information should be kept in each node are position, type, label, name, start line, start column, value, parent, and children.

structure(action, file_name, version, nodeid, type, label, name, start_line, start_column, value, ischild, parent, num_child, children)

Nodeid is the position of the node which is represented by the preorder numbering. Type is the classification of the mode(assignment, case, call, goto, if, procedure, etc.). The label field of the node is usually null, but only needed when goto's are used. The name field described the name of procedure, call, block, or package.

(3) Chunker/Notepad tables

The chunker provides a method for maintenance programmer to define text descriptions of portions of code during the process of reading and understanding the code. These descriptions are optimally displayed instead of the original code. For this function, the following data repository will be needed.

chunks(action, file_name, version, chunker_number, child, next, name, key_field, description, start_line, end_line)

Notepad provides a method for maintenance programmer to write descriptions with the code during the process of reading and understanding the code. These descriptions, which are optimally displayed along with the original code, assist maintenance programmer in building understanding of the code. That is a small note on the code to help explain what it is doing.

notes(action, file_name, version, notename, keyfield, description, location)

(4) Symbol table

The symbol table is generated by the parser and used by many parts of the maintenance tools. The following information should be in symbol table.

symtbl(action, file_name, version, symbol_id, symbol, type, class_value, decl_line, size, level, parent, line_num, next_symbol)

Class_value means class dependent value such as return type, passed parameters, etc. Size represents the width of symbol in characters. Level means symbol nesting level. Parent is a previous symbol identifier in this scope. Next_symbol means next symbol in the same scope.

(5) Usage table

The usage table contains the information concerning usage of the symbol. Each tuple

shows a reference to given symbol. The line number that the symbol was declared on is in the symbol table. For each node in usage table can be multiple references.

usage(action, file_number, version, usageid, symnode, symbol, refline, reftype)

Refline is a string of three parts, the line number of the reference, the column number and the type of reference. The types of reference are declaration, use, define, procedure call, call on use, procedure declaration.

(6) Test coverage

Test coverage keep the all paths for testing certain module.

testcover(action, file_name, version, test_plan, traversals, node_num)

(7) Test harness

Test harness provides the information of test driver, test stub, parameters.

testharn(action, file_name, version, module, parameter, param_type, param_value, stub_driver)

(8) Test path

The test paths are represented by three tables, pathtab, subtab, and path. These are shown in the following.

pathtab(action, file_name, version, create_date, path_num, status, beg_node, end_node)

subpath(action, file_name, version, create_date, seq_num, change_type, path_num, beg_node, end_node)

paths(action, file_name, version, create_date, path_num, type, node_status, curr_node, succ_node)

(9) Control flow graph

The control flow graph is used by several of maintenance tools. The main structure of graph is given as follows.

cfg(action, file_name, version, nodenum, cfgtype, cfglabel, cfgname, startline, startcol, endline, endcol, value, structparen, numchild, complex, previous, next, cfgparent, children)

(10) Data flow graph

Data flow graph is a repository for keeping the information to analyze data anomaly, liveness, etc.

dfg(action, file_name, version, nodeid, dfgtype, dfglabel, symbol, sym_name, reach, available, live, mod, used, anomaly, dfg_action, startline, startcol, endline, endcol, value, parentid, children, complex, previoid, next)

4.2 Non-delta Tables

Non-delta tables are changed without needing to back out the change. In addition, they are analyzed by examining the whole table, not just the changes. Examples include the linemap(line number table), the sequence number table, tables of metrics. Usually these will not have the version number as a key as the information contained in the them is version independent.

5. Examples

Incremental analysis and analysis on demand are two features that differentiate our design method from other software engineering environment's. The maintenance of objects deltas will be explained by example further here.

All the data objects can be incremental. This requires a great deal of record keeping and large number of objects in the repository. The maintenance depository will store all the base objects and the deltas, with possible exception of the source code. The base of the source code is stored in files but the source code deltas will be stored in the database.

The user must be able to get not only the current version of all the data in the database, but be able to re-construct previous versions.

Since each of the objects in the repository will have deltas, the maintenance tools must keep track of how each object current level is related to the others. Knowing the relationships between version levels is how the tools determine when an object needs to be updated.

The other part of the problem is determining the appropriate version level of other objects to be used in updating the current object. The dependency graph tells what other objects the current object must examine to get the data it needs but not which version in the other objects corresponds to the version of the object being updated. (Fig. 3) shows the example of delta update.

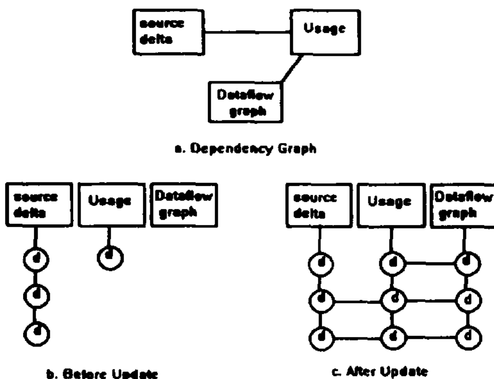
A common problem in software engineering environment is to keep the data up to date. Many of the data objects described here manage this problem by allowing constraints and procedures to be attached to the data objects.

6. Interface of Data Repository

The interface is a layer that is built between the software maintenance tools and data repository. This allows either side of the layer to change without affecting the other. Using a layered approach will make this translucent, if not transparent to the software maintenance tools.

There are two parts to the interface. One is the library of access routines and the other is the control table. The control table contains information on what to do if the caller requests data that is not there. Each of the delta tables has a generator table from which the information in the table is generated. Some tables may have multiple generators and some, like the source delta table, have a virtual generator in the user.

The library of access routines are the following.



(Fig. 3) the example of delta update

7. Conclusion

Program analysis is a key function of software maintenance tools. Maintenance tools extract various information from the data repository during program analysis. A main issue in design of maintenance tools is what data objects are really needed by tool components and how they are represented in efficient way. Maintenance tools requires special information repository to support analysis, making structured change and checking the adequacy of test cases.

This paper discussed the issue related to

〈Table 2〉 Access routine for data repository

Access routine	Function
get_delta(tableid, file_name, version)	Return the entries in the given table that are part of the given version.
get_all_delta(tableid, file_name, version)	Same as get_delt() but puts all the entries in the table pointed to
get_object(tableid, file_name, version, nodeid)	Return a pointer to a specific object with the indicated table
put_object(tableid, file_name, version, nodeid)	Store the object pointed to by pointer into table with the given node id
put_delta(tableid, file_name, ptr, action)	Make a delta entry in the table. It also the version number if it is different from current one if it is next.
get_table(tableid, pointer)	Fetch the entire table indicated by replacing it with the one pointed to

what kind of data objects the maintenance needs and how they should be organized. For handling delta, data repository has source delta objects and keeps the modification action. This papers suggestion provides both horizontal(all version of one data object) and vertical(all data objects of one version) delta analysis. The data repository was designed based on the philosophy of easy tool extension. Finally the interface of data repository is designed.

For the further research, the analysis of relationship between data objects can be suggested. Traditional data model can not represent sufficiently complex objects in CASE[7]. The design of data repository in maintenance needs a method that can represent various relationship between data objects such as part-of, version-of, reference.

References

[1] M. Chen, "A Framework for Integrated CASE", IEEE Software, Vol. 9, No. 2, March, pp. 18-22, 1992.

[2] I. Thomas, B. Nejmeh, "Definitions of Tool Integration for Environments", IEEE Software, Vol. 9, No. 2, pp. 29-35, 1992.

[3] A. Brown, P. Feiler, K. Wanau, "Past and Future Models of CASE Integration", 5th International Workshop on CASE, pp. 36-45, 1992.

[4] C. Ellis, S. Gibbs, "Active Objects:Realities and Possibilities", In W. Kim, H. Lochovsky, editors, Object-Oriented Concepts, Databases, and Applications, Chapter 22, ACM Press, pp. 561-572, 1989.

[5] D. Tsichritzis, et. al, "KNOs:Knowledge Acquisition, Dimension and Manipulation Objects", ACM Transactions on Office Information Systems, Vol. 5, No. 1, pp. 96-112, 1987.

[6] E. M. Choi, A. von Mayrhauser, "Support for Program Understanding during Maintenance via Chunking". 7th Midwest Computer Conference, pp. 125-134, 1993.

[7] S. Hudson, R. King, "The Cactis Project:Database Support for Software Environments", IEEE Transactions of Software Engineering, Vol. 4, No. 6, pp. 709-719, 1988.

[8] K. Dittrich, W. Gotthard, and P. Lockemann, "DAMOKLES-the Database System for the UNIBASE Software Engineering Environment", Data Engineering, Vol. 10, No. 1, pp. 37-47, 1987.

[9] F. Gallo, R. Minot, and I. Thomas, "The Object Management System of PCTE as a Software Engineering Database Management System", ACM SIGPLAN Notices, Vol. 22, No. 1, pp. 12-15, 1987.

[10] J. Keables, K. Roberson, and A. von Mayrhauser, "Dataflow Analysis and its Application to Software Maintenance", Proc. of IEEE Conference on Software Maintenance, pp. 355-347, 1988.

[11] S. Yau and S. Liu, "A Knowledge-based Software Maintenance Environment", Proc. Computer Software and Applications Conference, pp. 72-78, 1986.

최 은 만

1982년 동국대학교 전자계산학과 졸업(학사)
1985년 한국과학기술원 전산학과(전산학석사)
1993년 Illinois Institute of Technology 전산학과(전산학박사)
1985년~88년 한국표준연구소 연구원
1988년~89년 한국데이타통신(주) 주임연구원
1993년~현재 동국대학교 컴퓨터공학과 전임강사
관심분야 : 소프트웨어 개발 환경, 소프트웨어 유지보수, 소프트웨어 재사용, 객체지향 소프트웨어설계