Designing Test Engine for Computer-Aided Software Testing Tools

XUE-YING MA, BIN-KUI SHENG
College of Information Management
Zhejiang University of Finance & Economics
Hangzhou, P. R. China.
hzmaxueying@hotmail.com , shengbk@yahoo.com.cn

Abstract: - With the rapid development of software scale and programming languages, it is impossible to test software manually. The case for automating the software testing process has been made repeatedly and convincingly by numerous testing professionals. Automated tests can promote the efficiency of software testing and then to increase software productivity, improve software quality, and reduce cost in almost all processes of software engineering. White-box testing is one of the most important software testing strategies that can detect error even when the software specification is vague or incomplete. This paper gives a detailed description of the design and implementation of a testing engine. The testing engine, which is the kernel of a developed structured software-testing tool for the Visual Basic and C/C++ language, mainly consists of three components: program analyzer, source code instrumentation tool and intermediate database. In the testing engine, a block division mechanism and a new block-based CFG model are introduced and some block-based test adequacy criteria are extended. The programs are divided into a sequence of blocks and then instrumented and compiled in the testing engine, and all the information related to the test is saved in the intermediate database. The testing engine, acting as an agency, associates the testing automation module with instrumented executable program rather than the source code, and therefore the testing tool can easily be developed to accommodate new requirements and different testing adequacy criteria. It is also convenient to build a testing environment for multi-languages by modifying the program analyzer only, due to the flexibility of the software architecture.

Key-Words: computer-aided software test, testing engine, program instrumentation, Intermediate database, object-oriented software-testing.

1 Introduction

Software testing is the process of executing software and comparing the observed behavior to the desired behavior. The major goal of software testing is to discover errors in the software[1], with a secondary goal of building confidence in the proper operation of the software when testing does not discover errors. With the rapid development of software scale and programming languages, it is necessary to develop a computer-aided software-testing tool for automating the software testing process. The case for automating the software testing process has been made repeatedly and convincingly by numerous testing professionals. Most people involved in the testing of software will agree that the automation of the testing process is not only desirable, but in fact is a necessity given the demands of the current market. Since most products require tests to be run many times, automated testing generally leads to significant labor cost savings over time. Automated tests can also help eliminate human error, provides faster results, and then to increase software productivity, improve software quality, and reduce cost in almost all processes of software engineering[2].

A number of automated testing tools have been developed for white-box testing, functional testing(black-box testing), GUI-based application testing, web application testing, etc., and several of these are quite good inasmuch as they provide the user with the basic tools required to automate their testing process. For example, Parasoft has developed a series of testing tools, which not only include some program language products(e.g., Parasoft Jtest, C++ Tests, .Test and so on), but also some QA testing tools that automates web application testing, message/protocol testing, cloud testing, security testing, and behavior virtualization(e.g., SOAtest)[3]. Relational Purify and other Relational's software testing products are the software testing tools for C++, VB and JAVA[4]. TestComplete of AutomatedQA Corp. is a full featured automated software testing tool that can provide unit testing, Java testing, Data-driven testing, functional testing ,etc[5]. Quick Test Professional (QTP) is an automated functional Graphical User Interface (GUI) testing tool that allows the automation of user actions on a web or client based computer application, WinRunner is a Mercury interactive enterprise functional testing tool which is used to quickly create and run sophisticated automated tests on your application, and LoadRunner is an industry-leading performance and load testing product by Hewlett-Packard, etc[6]. A fundamental strength that all white box testing strategies share is that the entire software implementation is taken into account during testing, which facilitates error detection even when the software specification is vague or incomplete. But most white-box testing tools limited to work on unit testing level(e.g. Parasoft Jtest) because of the extraordinary difficulty and complexity of the analysis and representation of the program structure, the test case design and over budget cost of testing and so on. Our motivation of developing the tool is to make the structured testing activities automatic or semi-automatic. Consequently, we have developed computer-aided software testing tool for Visual Basic, Delphi, C++, etc., which is designed for structured testing. It works on the level of unit testing and integration testing, and is extended to support regression testing. It provides auto-generation of graphs and charts, test coverage analysis automation, quality measurement automation, dynamic tracing automation, and testing execution automation.

Referring to the following Figure 1., the computer-aided software-testing tool mainly consists of two modules: Engine and Automation. The former is the kernel of testing tool. The engine reads source code (parsing), generates structure information of the program and saves it in static analysis database, then instruments the program to meet the test requirements and links the object files to produce the executable program. When running the test ~~running~~, the engine records the dynamic test information in dynamic database that is consequently used in the automation modules.

This paper addresses the issue surrounding the design and implementation of test engine for structured software-testing tools[7][9]. The main contributions of this paper are listed below:

♦ Present the architecture of software-testing tool that may provide high reusability and flexibility according to the theory of software engineering[10][11][12].

♦ Present a description of the structure of the testing engine. The test engine mainly consists of three components: a program analyzer used to divide the program into a sequence of blocks; an instrumental tool for code instrumentation; and an Intermediate database that records the test histories.

♦ Propose a block-based division mechanism and a new block-based control-flow graph(CFG) model for effectively representing the structure information of the program to be tested, and then extend the traditional program-based software test data adequacy measurement criteria based on the block-based division mechanism, and empirically analyze the subsumption relationship between these measurement criteria.

♦ Present a new technology of code instrumentation based on Dynamic Link Library.

♦ Give a complete description of the structure of the Intermediate database and the information that is stored in the database.

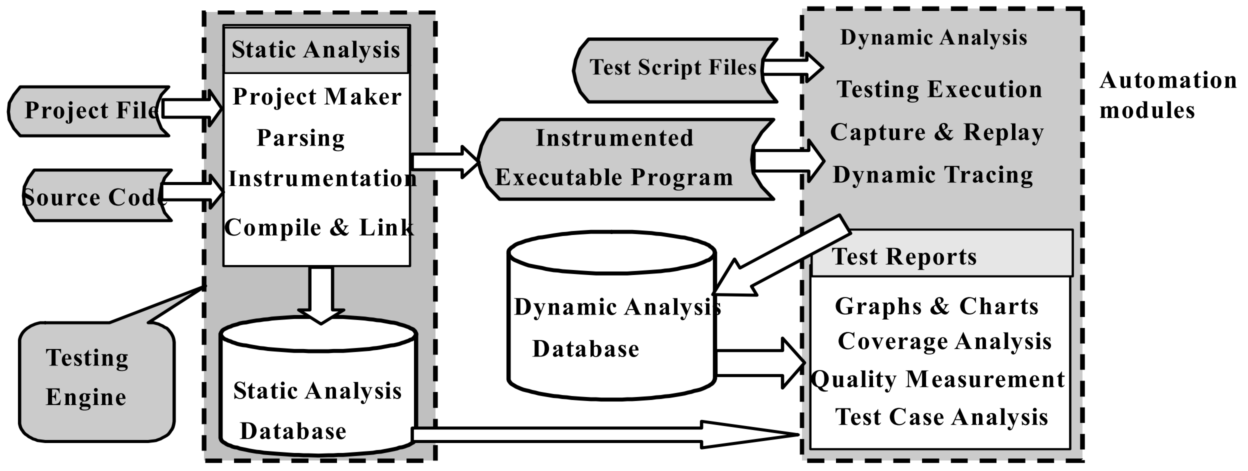


Figure 1. Architecture of the Computer-Aided Software-Testing Tool

The remainder of this paper is organized as follows: Section 2 gives a brief description of the working flow of the testing engine. Section 3 introduces the program analyzer, which consists of a block-based division mechanism, a new block-based CFG model and some block-based coverage criteria extended based on the traditional CFG model. Section 4 gives a description of the instrumental tool. In section 5, we describe the structure of the Intermediate database. Finally, Section 6 summarizes our results and discusses some future work.

2 Working Flow of the Engine

A software system generally contains more than one source code file, which is organized by project file. Thus the first step of our testing tools is to understand the project file. As the kernel of the testing tool, the engine analyzes the project file (Project Maker) and reads the source code files according to the information described by the project file, then parses the source code, and synchronously performs code instrumentation through static analysis. Finally the engine compiles the instrumented code and links the object files to produce the executable program. Static analysis is to get necessary structure information of source code. Then these data are saved into the static analysis database, which prepares for the auto-generation of graphs, charts and quality measurement, and guiding the test. Program instrumentation is to insert our own code into the specified place of the program in order to get the dynamic test information during the process of running instrumented executable program, and save them in the dynamic analysis database.

In the testing tool, only the static analysis module is directly associated with the source code. During the course of testing, the engine acts as agency that associates the testing automation module with instrumented executable program rather than the source code. So the automation modules are indirectly associated with the source code through the Intermediate database[13]. It is convenient to support multi-language for the testing tool via the engine, because we only need to modify the program analyzer in the engine without modifying the consequent automation module by modifying the grammar rules and designing a general Intermediate database. Thus we implement the reuse of modules inside the system to a large extent.

While different languages have the different grammar structures and writing styles, it is impossible to parse different languages with a uniform program analyzer. We use Bison[8] to parse a language in order to enhance the reusability of module. In order to be useful, a program must do more than parse input. We implement the semantic actions such as program division, and code instrumentation, etc., through program analyzers and instrumental tool in the engine.

3 Program Analyzer

As we mentioned above, the tool is designed for structural testing, especially for path testing. It requires complete knowledge of the program's structure (i.e., source code). Thus the first step of our testing tools is to understand the project file. The program analyzer analyzes the project file (Project Maker) and reads the source code files according to the information described by the project file, and then parses the source code.

CFGs describe the logic structure of software modules. A module corresponds to a single function or subroutine in typical languages, has a single entry and exit point, and is able to be used as a design component via a call/return mechanism. Each flow graph consists of nodes and edges. Traditionally, the nodes of CFG represent computational statements or expressions, and the edges represent transfer of control between nodes. Each possible execution path of a software module has a corresponding path from the entry to the exit node of the module's CFG. This correspondence is the foundation for the structured testing methodology. Furthermore, it is completely independent of text formatting and is nearly independent of programming language since the same fundamental decision structures are available and uniformly used in all procedural programming language[16].

In order to describe the structure information of the source program and evaluate the test adequacy efficiently, we have presented a block division mechanism and a new block-based CFG model, and extended some block-based test adequacy criteria. Same as the traditional flow graph model, the block-based CFG model presented in section 3.2 is also independent of the text-free procedural programming language.

3.1 Block Division Mechanism

According to the block division mechanism, there is only one kind of component: block in the program. A block is a sequence of program statements. Formally, it is such a sequence of statements that if any one statement of the block is executed, all statements thereof are executed. Less formally, a block is a piece of straight-line code. There exist two kinds of block named ***Node*** and ***Segment***. In the engine, all the programs to be tested will be divided into a sequence of segments and nodes, and the information of the block-based division, which will be used to guide structured testing, is saved into static database.

3.1.1 Node

There are three types of node, namely ***decision***, ***junction*** and the ***entry/exit point*** of program unit.

A ***decision*** is a program point at which the control flow can diverge. Some examples in C++ programming language, as shown in Table 1., include IF (condition), SWITCH (expression), FOR (expression; expression; expression) and WHILE (condition)(while statement or do…while statement).

A ***junction*** is a point in the program where the control flow can merge. In C++ language, examples of junctions are "DO" in DO…WHILE statement, "ELSE" in IF…ELSE statement, "CASE" and "DEFAULT" in SWITCH statement and statement labels.

The ***entry/exit point*** of program unit is defined as the begin/end point of a scope, which is the portion in program within which a declaration applies. From the point of view of branch body, although the body's end point is also a node, the begin point isn't an alone node, and it is together with the above decision or junction node.

Table 1. Decisions in C programming language

Statements	Format of statement	Descision

If statement;	if (condition)
	if body
	[else
	Else body]
		IF (condition)

Switch statement;	switch (expression)
	{
	……
	}
		switch (expression)

For statement;	for (expression; expression; expression) loop body	for (expression; expression; expression)
While statement;	while (condition) loop body	while (condition)
do…while statement;	Do loop body while (condition)	while (condition)

3.1.2 Segment

A ***segment*** is a sequence of computer statements between two consecutive branch points. It has one entry and one exit. Here the branch points include the above nodes and the position between unconditional jump statements and its next statement. In C++ language, examples of unconditional jump statements are GOTO, RETURN, BREAK and CONTINUE statements.

3.1.3 Invisible Segment

Besides the above segment, there is another special segment — ***invisible segment***, which is designed for recording the paths that have been executed. For example, any IF statement lacking ELSE part has an "IF statement invisible" segment by definition which is executed when the IF(condition) is not satisfied. For each repetition statement, there are two invisible segments. One of them is "low-end loop boundary invisible segment" which is executed if the repetition condition is never satisfied, the other is "high-end loop boundary invisible segment" which is executed when the repetition condition is no more satisfied. The "high-end loop boundary invisible" as well as the "IF statement invisible" are also called ***base invisible segment***.

3.2 Block-Based control-flow graph(CFG)

CFG is a graphical representation of a program's control structure, and plays an important role in debugging, control flow analysis and coverage analysis. In order to describe the program's structure efficiently, we present a new CFG model based on blocked division mechanism introduced in the testing engine. According to the new CFG model, the nodes of the graph represent the blocks but not the statements or expression of the program and the edges represent transfer of control between the blocks. Thus we can reduce the number of nodes of the graph in some degree and then the complexity of the flow graph.

As an example, consider the C function in Figure 2., which implements Euclid's algorithm for finding greatest common divisors. The traditional nodes of the program are numbered A0 through A13 and the corresponding control flow graph is shown in Figure 3., in which each node is numbered 0 through 13 and edges are shown by lines connecting the nodes. Otherwise, the visible nodes and segments of this module are numbered B0 through B9 (as shown in Figure 4.). Three invisible segments are introduced for better recording the executed paths based on the block division mechanism. The first one is the "IF statement invisible segment" which will be executed when the decision "if(n/m)"(B1) is not satisfied, the second is the "low-end loop boundary invisible segment" which will be executed when the decision "while(r!=0)"(B5) is never satisfied and the third is the "high-end loop boundary invisible segment" which will be executed when the decision "while(r!=0)" is no more satisfied after the loop has repeated for several times. The logical view of the corresponding block-based CFG is shown in Figure 5., in which the nodes are numbered 0 through 12. By definition, node 0 is the "IF statement invisible" segment, node 8 is the "low-end loop boundary invisible segment" and node 9 the "high-end loop boundary invisible segment".

```
A0  int euclid(int m, int n)
    { /* Assuming m and n both greater than 0,
         Return their greatest common diviser.
         Enforce m>=n for efficiency.*/
          int r;
A1      if(n>m) {
A2       r=m;
A3       m=n;
A4       n=r;
A5        }
A6   r=m%n;
A7      while (r!=0){
A8       m=n;
A9       n=r;
A10      r=m%n;
A11      }
A12   return n;
A13  }
```

Figure 2. Source listing for module "euclid"

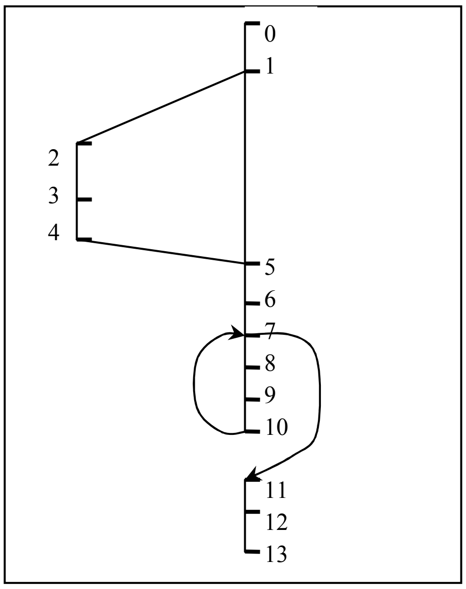


Figure 3. Traditional CFG for module "euclid"

```
B0  int euclid(int m, int n)
    { /* Assuming m and n both greater than 0,
         Return their greatest common diviser.
         Enforce m>=n for efficiency.*/
          int r;
B1      if(n>m) {
B2       r=m;
         m=n;
         n=r;
B3      }
B4      r=m%n;
B5      while (r!=0){
B6       m=n;
         n=r;
         r=m%n;
B7      }
B8    return n;
B9  }
```

Figure 4. block division for module "euclid"

The reason why the CFG shown in Figure 5. is only the logical view of the block-based CFG of module "euclid" is that, in our testing engine, the additional constraints were added to block-based CFG compared to the traditional CFG, and can serve as reference for the design of integrated circuit. Different from traditional CFG, the new block-based CFG model can not only show the sequence of the source code but also the coverage information, and is more conveniently to be used to guide testing. A sample is shown in Figure 6. In the Figure 6, the CFG is shown in a block-flow form. Different kinds of block are identified by different graphical elements. This model can show not only the blocks information include the block's sequence number, the line number of the source code the block correspond and the times executed during a test etc., but also the source codes about the block. We can get complete knowledge of the program's structure information conveniently, i.e., path information.

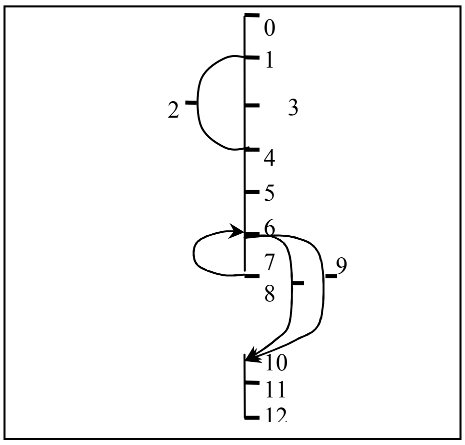


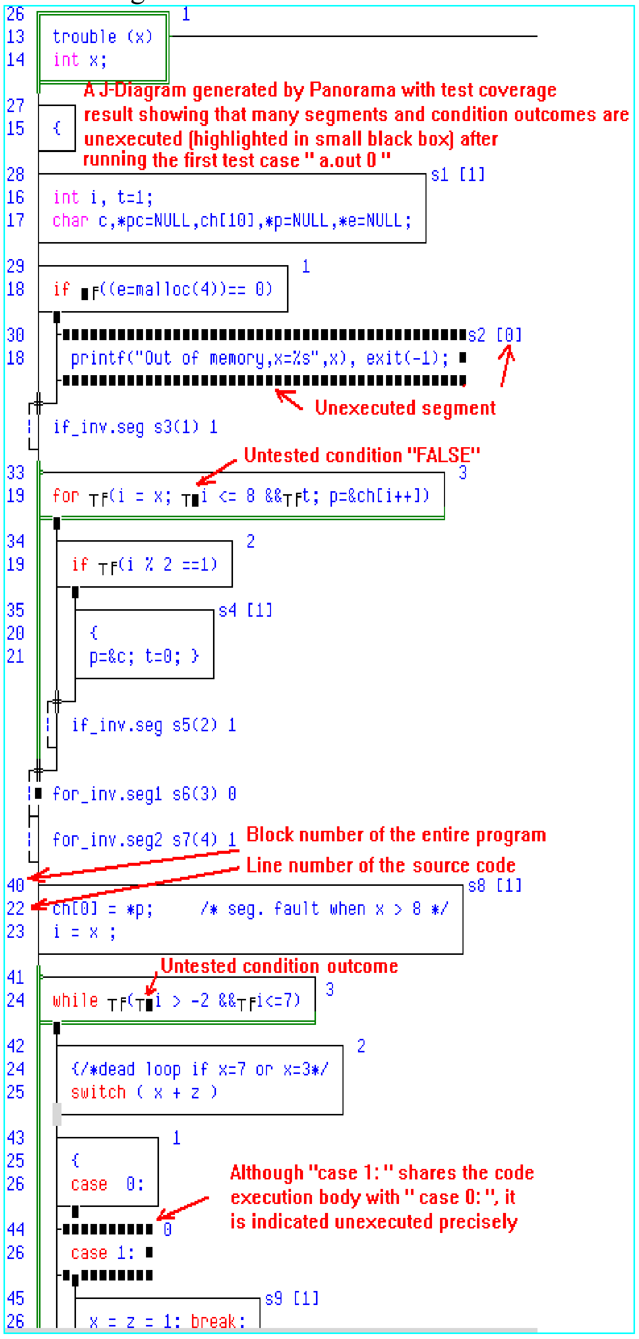
Figure 5. Logical view of Block-based CFG for module "euclid"

It is very convenient to find the corresponding code from this CFG, and can provide a lot of information to help programmers to test, debug and conduct coverage analysis. For example, testers can easily find which block is not tested and which condition/decision is not covered during a certain test. The blocks affected by the modification in new version program are easy to be identified too.

One other obvious advantage of the block-based flow graph model is the reduction of the nodes number, especially in the large scale program. Furthermore, some block-based test adequacy criteria are extended which will be discussed in section 3.3.

3.3 Block-Based Test Coverage Criteria

Code coverage is an important type of test effectiveness measurement. It describes the degree to which the source code of a program has been tested. Code coverage is a way of determining which code statements or paths have been exercised during testing. With respect to testing, coverage analysis helps in identifying areas of code not exercised by a set of test cases. Alternatively, coverage analysis can also help in identifying redundant test cases that do not increase coverage. There are various measures for coverage, such as statement coverage, branch coverage, path coverage, multiple condition coverage, and function coverage. Based on the block division mechanism, we have extended traditional test coverage criteria.


```
26        1
13  trouble (x)
14  int x;

27
15  {

28                                          s1 [1]
16  int i, t=1;
17  char c,*pc=NULL,ch[10],*p=NULL,*e=NULL;

29                          1
18  if F((e=malloc(4))== 0)

30                                          s2 [0]
18    printf("Out of memory,x=%s",x), exit(-1);

    if_inv.seg s3(1) 1

33                                           3
19  for TF(i = x; Ti <= 8 &&TFt; p=&ch[i++])

34                    2
19    if TF(i % 2 ==1)

35                    s4 [1]
20      {
21      p=&c; t=0; }

      if_inv.seg s5(2) 1

    for_inv.seg1 s6(3) 0

    for_inv.seg2 s7(4) 1

40                                          s8 [1]
22  ch[0] = *p;     /* seg. fault when x > 8 */
23  i = x ;

41                            3
24  while TF(Ti > -2 &&TFi<=7)

42                              2
24    {/*dead loop if x=7 or x=3*/
25    switch ( x + z )

43            1
25    {
26    case  0:

44          0
26    case 1:

45                    s9 [1]
26      x = z = 1; break;
```
A J-Diagram generated by Panorama with test coverage result showing that many segments and condition outcomes are unexecuted [highlighted in small black box] after running the first test case " a.out 0 "
Unexecuted segment
Untested condition "FALSE"
Block number of the entire program
Line number of the source code
Untested condition outcome
Although "case 1: " shares the code execution body with " case 0: ", it is indicated unexecuted precisely

Figure 6. Block-based CFG

3.3.1 SC0

A set of test cases of a program satisfies **SC0** if all nodes and visible segments of the program have been executed at least once.

$$\mathbf{SC0 = (}\boldsymbol{B_{exe\_segment}} \div \boldsymbol{B_{segment}}\mathbf{)*100\%} \qquad \mathbf{(1)}$$

Here $\boldsymbol{B_{segment}}$ is the number of segments in a program or program module and $\boldsymbol{B_{exe\_segment}}$ is the number of segments that have been executed at least once.

SC0, as a basic block test coverage, is better than the statement test coverage. For example, the following IF statement:

if (Condition) Statement;

When the **if(Condition)** is not true, the statement part can not be executed. According to the traditional statement test coverage, we can't identify whether this IF statement is executed or not. However, SC0 can clearly know the statement part is not executed, because there are two blocks in the above IF statement based on the block division mechanism, with one being a ***node***(Condition part of IF statement) and the other being a ***segment***(Statement part).

SC0 covers the statement test coverage. The one that satisfies **SC0** must also satisfie the statement test coverage criteria.

3.3.2 SC1

A set of test cases of a program satisfies **SC1** if it satisfies **SC0** and all basic invisible segments of the program have been executed at least once. Basic invisible blocks consist of IF, DO-WHILE, SWITCH, and high-end loop boundary invisible blocks.

$$\mathbf{SC1} = (B_{exe\_segment} + B_{exe\_binvisiblesegment}) \div (B_{segment} + B_{binvisiblesegment}) * 100\% \quad (2)$$

Here $B_{binvisiblesegment}$ is the number of base invisible segments in a program or program module and $B_{exe\text{-}binvisiblesegment}$ is the number of base invisible segments that have been executed at least once.

SC1, a standard block test coverage, is similar to the branch test coverage but better than the branch test coverage. Consider the following statements:

```
Statement1;
goto 50;
Statement2;
50: Statement3;
```

Apparently, Statement2 is a dead statement and will never be executed, but this situation can't be identified according to the branch test coverage which is based on the branch statement. Statement2 is also a segment according to the block division mechanism, so **SC1** can find this dead statement.

SC1 covers **SC0** and the branch test coverage.

3.3.3 SC1+

A set of test cases of a program satisfies **SC1+** if it satisfies **SC1** and the entire low end invisible segments of the loops in the program have been executed at least once.

$$\mathbf{SC1+} = (B_{exe\_segment} + B_{exe\_invisiblesegment}) \div (B_{segment} + B_{invisiblesegment}) * 100\% \quad (3)$$

Here $B_{invisiblesegment}$ is the number of all the invisible segments in a program or program module and $B_{exe\text{-}invisiblesegment}$ is the number of all the invisible segments that have been executed at least once.

SC1+ covers **SC1**.

3.3.4 J-Coverage

J-Coverage is defined as the ratio of the number of executed visible and invisible blocks plus executed outcomes of conditions to the number of all visible and invisible blocks plus all outcomes of conditions in a program or program module. **J-Coverage** covers **SC1+**.

$$\mathbf{SC1+} = (B_{exe\_segment} + B_{exe\_invisiblesegment} + B_{exe\_node}) \div (B_{segment} + B_{invisiblesegment} + B_{node}) * 100\% \quad (4)$$

Here B_{node} is the number of all outcomes of conditions in a program or program module and B_{exe_node} is the number of all the executed outcomes of conditions.

It is the strongest test coverage criteria provided by our testing tool.

4 Source Code Instrumentation

Instrumentation is the process of non-intrusively inserting code into the specified place of the source code that is being analyzed and then compiling and executing the modified (or instrumented) software. The instrumented executable program is prepared for coverage analysis automation and dynamic tracing automation and testing execution automation.

4.1 Instrument based on Dynamic Link Library

According to the traditional technology of the instrumentation, we first create a Lib of functions that are related to some certain operations or codes, then instrument statements into the source code file to call these functions, and finally link this Lib when we compile the instrumented source file. In order to save the data from memory to disk, we must instrument the output statement before each termination in the source code file. A fatal weakness of this method is that the data in the memory will be lost when the execution of the instrumented program don't terminated normally or interrupted by the user.

In order to avoid the weakness mentioned above, we introduced the techniques of the Dynamic Link

Library (DLL)[14][15] into our engine to collect and update the data by creating a DLL file that consists of some function used to manipulate the dynamic database. On the other hand, we should instrument some command in the source code file to call these function in DLL. When the instrumented program is executed, the data will not be saved into the memory of the program itself but of the DLL.

There are two benefits of this method. One is that the data will not be lost when the execution is terminated unexpectedly and can accurately locate where the fault happens which causes the software to collapse because the DLL is dependent on the program. The other is that the system such as the Windows can terminate the use of DLL when the user interruption or the collapse happens and the DLL will do some specific operation which the user has defined to cope with the abnormal termination.

4.2 Work Flow OF the Instrumentation

Step 1: Parsing the project file into source code files;

Step 2: For each source code file,

(1) Parsing the structure and saving the structure information into DD file, DDH file, DDC file and DS file;

(2) Parsing the structure and createing a linked chain named InstrChain which is used to describe the information of the instrumentation, shown as following;

```
struct InsPoint {
int SrcPos; // position that the code instrumented
int Style; // type of the code instrumented
int Rno; // test record point used to get the test
coverage
int Cno; //  test record point of the condition test
int Lno; // suffix of the local simulated variables
struct InsPoint *Next;
  } *InstrChain;
```

(3) Replacing the suffix of the local simulated variables in the linked chain InstrChain. Some important symbols are listed in the following.

@R , address of the logic counter of the source code file , ordered by the line number of the instrumented point in the program. The value of the variable @R in each point is same;

@C, the sequence number of the decision in the source code file and ordered by the line number of the instrumented point in the program;

@L, the suffix of the local simulated variables that is defined to remember the states of the program;

(4) Instrument the source code file according to the information in the linked chain InstrChain;

Step 3: Adding the public module to declare the global variables and the quotes of the function in the DLL; Updating the project file by adding the public module.

5 Intermediate Database

Almost all object-oriented languages have the same object techniques as classes, objects, inheritance, polymorphism and dynamic binding, etc. In order to reuse the modules inside the system, we introduced the intermediate database as the kernel of the engine based on the theory of the software construction.

There are two databases: one is the static analysis database and the other is the dynamic analysis database.

5.1 Static Analysis Database

The information in the static database is used in the test coverage analysis and quality measurement automation module, and it is also used to generate graphs and charts automatically.

We create one data file to save source code information for each program. In order not to save a string more than one time and to manage them easily, we use hash table to save the address (or pointer) of the string, but not the string itself.

The information in the static database includes structure information, class definition information, method definition information and block division information of the program, etc., as shown in Figure 7.

(1) Structure Information

The structure file_node0 is used to save the structure information, and includes the name of the source file, the path the dynamic database file, the files the programs used, the global static or dynamic variables defined in the program, the block sequences of source file and all the other information about the source file structure. All the structure nodes of the source files are linked into a chain.

(2) Class Definition Information

The data structure class_def0 shown in Figure 7. is used to save the information of the classes defined in the source file. The information includes the name of the class, the source file this class belongs to, the method defines this class, the outer class of this class, the base class and the friend class, the private and public variables and all the other information about

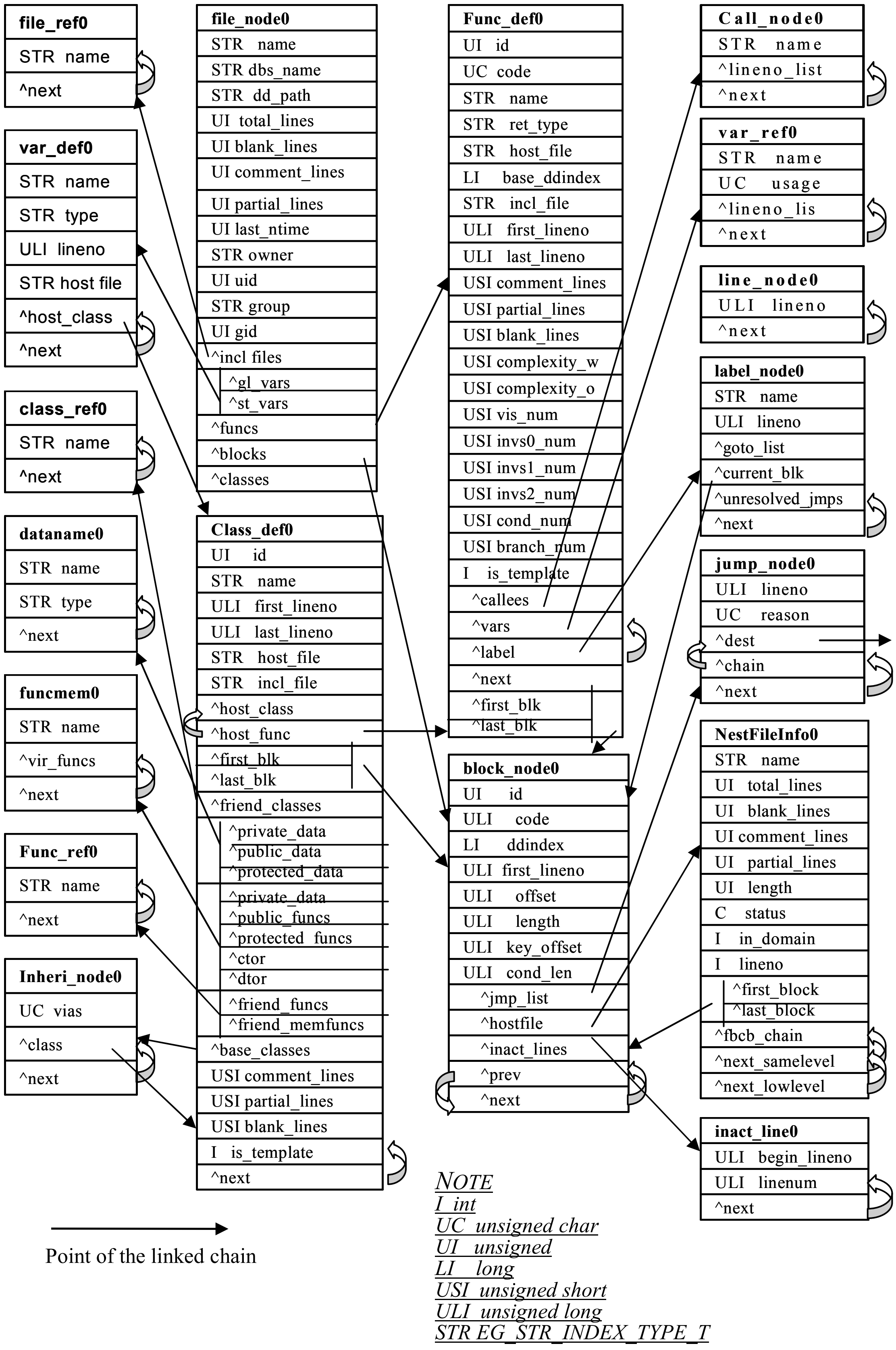


Figure3. Structure of the static analysis database

each class. In the same way, all the structure nodes related to the classes are linked together.

(3) Method Definition Information

The data structure func_def0 shown in Figure 7. is used to save the information about a method including the name of the method, the data type returned by the method, the type of the method, the source file in which the method defined, the sequence number of the first code block of the method in the dynamic database file, the first code block of the method and the last code block of the method, etc. All the structure nodes of the method are linked together too. There are 8 kinds of type about the method: macro function, non-class-member function, class-member function, virtual function, inline function, overloading function, static function, pure virtual function, etc.

Here, some data fields we should fill in with the name of the function, but sometimes the name of the function is not the true name of the function, because of the function overloading. For example, two overloading functions: int foo() and int foo(int n), their true name is same, so we should change their name into foo_v and foo_n to avoid collision when we save the name of the function in the database.

(4) Block Division Information

In the engine, all the programs to be tested will be divided into a sequence of segments and nodes. The block division information includes the type of the block, the number of the block record point, the number of the first line of the block in the program, etc. The information of each block of the program is assembled as a structure node named block_nod0 (shown in Figure 7.) and all the nodes are linked together.

5.2 Dynamic Analysis Database

The data in the dynamic database will be used in the automation module. Because dynamic analysis is implemented based on the sequence information of blocks (segment or nodes), which are not associated with a certain programming languages, but associated with the logic structure of the source code file only, the data in the dynamic database is independent of the language.

The dynamic database saves such information as the times the segment or the decision have been executed during one test, the value of each condition/decision during the process of being executed, which segment or method or class are tested when a test case is performed and so on..

There are four kinds of dynamic data file in the dynamic database such as DDH file, DDC file, DD file, and DS file. We will introduce the structure of these files in the following.

(1) DDH File. In the DDH file, there is information such as the number of the test record point in the program, the time this execution cost, whether the program related to the test record point executed and the time it needed, etc.

(2) DDC File. We use the DDC file to save the information about the value of each condition/decision during the execution of the program. The information includes the number of the condition, the number of the decision, and their value during the execution. For a statement "a&&(b || c)", we define the whole statement as a decision, and each sub-statement 'a' or 'b' or 'c' as a condition.

(3) DD File. The DD file is used to save the times each test record point executed during an execution of the instrumented program, the cost (i.e., running time) and other information about the execution. The data in DD file is saved in the format of the binary bit map.

(4) DS File. The DS file is used to save the strings that will be used to display the test coverage. We use a particular symbol '\n' (0x0A) as the separator among these strings that are not simply copied from the source file but generated during the process of program parsing. For example, a string's format used to describe a class declaration statement is that: 'class'+ the name of the class + the information of the inheritance.

6 Conclusion

This paper has presented a flexible architecture of our structure testing tools. The designing of the testing engine, which is the kernel of the testing tool, took the reusability of the component into consideration. The testing engine consists of three components: program analyzer, instrumental tool and intermediate database. In the program analyzer, we presented a new block-based CFG model for effectively guiding the structured testing and extended some block-based test adequacy criteria based on the block-based division mechanism. The introduction of the DLL technique could also be able to promote the flexibility of the handling of the unexpected or user interrupted termination. Furthermore, the Intermediate database, which acts as a bridge between the testing engine and the automation module, improves the reusability of the software component.

According to the architecture of these software-testing tools, it is convenient to support multi-language for the testing tool via the engine, because we need only modify the program analyzer without modifying the consequent automation module.

Based on the block-based division mechanism, we can easily extend the traditional test adequacy measurement criteria by modifying the adequacy calculation module.

A new block-based CFG model, which is different from the traditional CFG model, was proposed in our testing engine. This model can serve as reference for the design of integrated circuit, and show the sequence of the source code as well as its coverage information.

The Intermediate database saves all the structure information of the program and the dynamic test history information includes the test cases and the test coverage, result and cost information of each test case, so it is convenient for the testing tools to evolve to accommodate new requirements such as automated regression testing, test selection/reduction, test visualization to make the fault location easier, etc.

Furthermore, the technology of instrumentation has little influence upon the execution efficiency of the instrumented program. We have made an experiment with an audio-play program, and the testing tool of the Pure Series developed by Rational has much more influence than ours when the instrumented executable program been executed.

Our future work includes extending the test data adequacy measurement criteria that fit better for the object-oriented software test, computer aided regression test and test selection, reduction technique based on the information stored in Intermediate database and so on.

7 Acknowledgements

The author thanks Li YAO as well as other members at the Institute of System Architecture, Zhejiang University, for their insights and comments.

This paper supported in part by project ‘Study on Object-Oriented Software Testing Automation’ (60073027) from National Natural Science Foundation of China. The paper was also partially supported by project ‘Study on regression test selection for Object-Oriented Software——test case minimize techniques’ (Y200804140) from Scientific Research Fund of Zhejiang Provincial Educational Department.

References:

[1] Myers, G., The Art of Software Testing, Wiley,1989.

[2] Renjie Zheng, Computer’s Software Testing Techniques, the Tsinghua University Press, Beijing, 1992.

[3] http://www.parasoft.com/jsp/home.jsp?itemId=0,2010-10

[4] http://www-306.ibm.com/software/rational/, 2010-10

[5] http://www.automatedqa.com/lp/tc-aw-all-content.asp?gclid=CJiuhtXd06QCFcMDHAodGhPYJg, 2010-09

[6] http://www.onestoptesting.com, 2010-10

[7] Li YAO, Research On The Object-Oriented Software testing, The Doctor Degree paper of Zhejiang University, Hangzhou, 2002.12.

[8] Boris Beizer, Software Testing Techniques, Van Nostrand Reinhold Company, New York, 1990.

[9] David Kung, Pei Hsia, Jerry Gao. Object-Oriented Software Testing. IEEE Computer Society Press, 1998.

[10] Finkelstein, A. Architectural Stability, Some Preliminary Comments, http://www.cs.ucl.ac.uk/staff/ a.finkels-tein, 2000.

[11] Kazman, R., Abowd, G., Bass, L. and Webb, M. SAAM: A Method for Analyzing the Properties of Software Architectures. In: Proceedings of the 16th International Conference on Software Engineering (Sorento, Italy), 1994, pp. 81-90.

[12] Kazman, R., Asundi, J., and Klein, M., Quantifying the Costs and Benefits of Architectural Decisions, In: Proceedings of the 23rd International Conference on Software Engineering, Toronto, Canada, 2001, pp. 297-306.

[13] Clements, P. Active Reviews for Intermediate Designs (CMU/SEI-2000-TN-009), Software Engineering, Institute, Carnegie Mellon University, 2000.

[14] Yuan Weihua, Cheng Lan, Yang Zhenghua. Using DLL for PC/104 Data Acquiring in LabVIEW Platform. IN: Proceedings of the 8th International Conference on Electronic Measurement & Instruments, Xian China, 2007, Issue 11, Page 57-58,61.

[15] YoungHan Choi, HyoungChun Kim, DoHoon Lee. An Empirical Study for Security of Windows DLL Files Using Automated API Fuzz Testing. In: Proceedings of the 10th International Conference on Advanced Communication Technology, Gangwon-Do Korea , 2008. Vol.2,pp. 1473–1475.

[16] McCabe, T. and A. Watson, Software Complexity, CrossTalk: The Journal of Defense Software Engineering December 1994.