I.J. Education and Management Engineering **2011, 1, 63-67**
Published Online July 2011 in MECS (http://www.mecs-press.net)
DOI: 10.5815/ijeme.2011.01.12

Available online at http://www.mecs-press.net/ijeme

Method of Uninitialized Variable Detecting for C++ Program

[a]Wan Lin, [b] Liu Juan, [c] Wang Qinzhao, [d] Zhang Wei

[a, b, c, d] *Computer Department Armored Force Engineering Institute Beijing, China, 10072*

Abstract

We present a testing approach which is a partially automated, partially manual inspection process that reports defects in C++ source code. In this paper we considered one of the faults type- uninitialized Variable- which the approach can detect. Uninitialized Variable is a common kind of error in programs written in C++, it often causes error result or system collapse. This paper analyses the classical C++ uninitialized variable errors, and describes a detecting method of uninitialized variable errors combining the advantage of ASI technology which based on static analysis.

Index Terms: software testing; uninitialized variable syntax tree; controlling flow graph

1. Introduction

Software testing is very labor-intensive and expensive; It accounts for approximately 50% of the cost of a software system development.

Dynamic testing is a critical part of total quality assurance but it has many limitations, including:

•It is expensive and time-consuming to create, run, validate and maintain test cases and processes.

•Code coverage drops inexorably as the system grows larger, meaning that testing validates less of the system.

•It can be difficult and time-consuming to trace a failure from a test case back to the root cause so that developers know what code to change.

Static testing can avoid these limitations. As a static testing technology, software inspection or code review is a visual examination of source code to detect defects. Automated software inspection technologies are now emerging by overcome many of the disadvantages inherent in manual inspections. These technologies can locate many common programming faults. This paper introduces a method of uninitialized variable detecting. Before program running, warning messages regarding possible software defects would be reported by scanning the using of variable.

Corresponding author:
E-mail address: www_l@tom.com

2. Abstract syntax tree analysis for C++

The starting point for the representation is abstract syntax trees. A parser reads the source code and produces an abstract syntax tree, which models all of the structural information contained in the source code.

The first application is to scan the C++ code for information about names, expressions, types and so on.

We imposed several requirements during the static analysis process. Lexical analysis generates syntax nodes for tokens such as constants and names. Parsing generates an abstract syntax tree with internal nodes representing appropriate syntactic constructs. Semantic analysis makes passes over the syntax tree to resolve names and operators and to transform the tree into a standard form.

2.1. Abstract syntax tree structure

All the faults detecting algorithms are based on abstract syntax trees. An abstract syntax tree is a syntax tree that has been cleaned up or abstracted to reflect the underlying language constructs without any syntactic sugar. It includes leaf nodes that represent non-keyword terminals in the language as well as internal nodes that represent syntactic constructs. The abstract syntax tree includes all the relevant information of the source and is a full representation of the source program. For example, an "if" statement such as "if (a<b) q=c+d else q=e+f" would be represented as an abstract syntax tree as follows:

The top-lever node would represent the entire if statement. There are three subtrees under the if statement node:

- A subtree that models the condition "a<b" would be under the if statement via the if-condition attribute.
- The subtree that models the assignment "q=c+d" would be under the attribute "if-then actions".
- The subtree that models the assignment "q=e+f" would be under the if-else-action attribute.

Figure 1 shows part of the parse tree of this example:

2.2. Abstract syntax tree pass through

The second application involved the generation of program dependence graphs for C++. These were needed for a research project that attempts to help the user restructure programs. Because the algorithms for uninitialized variable detecting are based on control flow graph, the next step is to annotate this abstract representation with information about the control structures (control flow) of the application. As while as the abstract syntax tree been pass through, the algorithms worked. Some inspection points are reported about uninitialized variable after once scan of the tree. Our service based on ASI technology and the algorithms would deliver reports that show the cause and location of software defects in C++ applications.

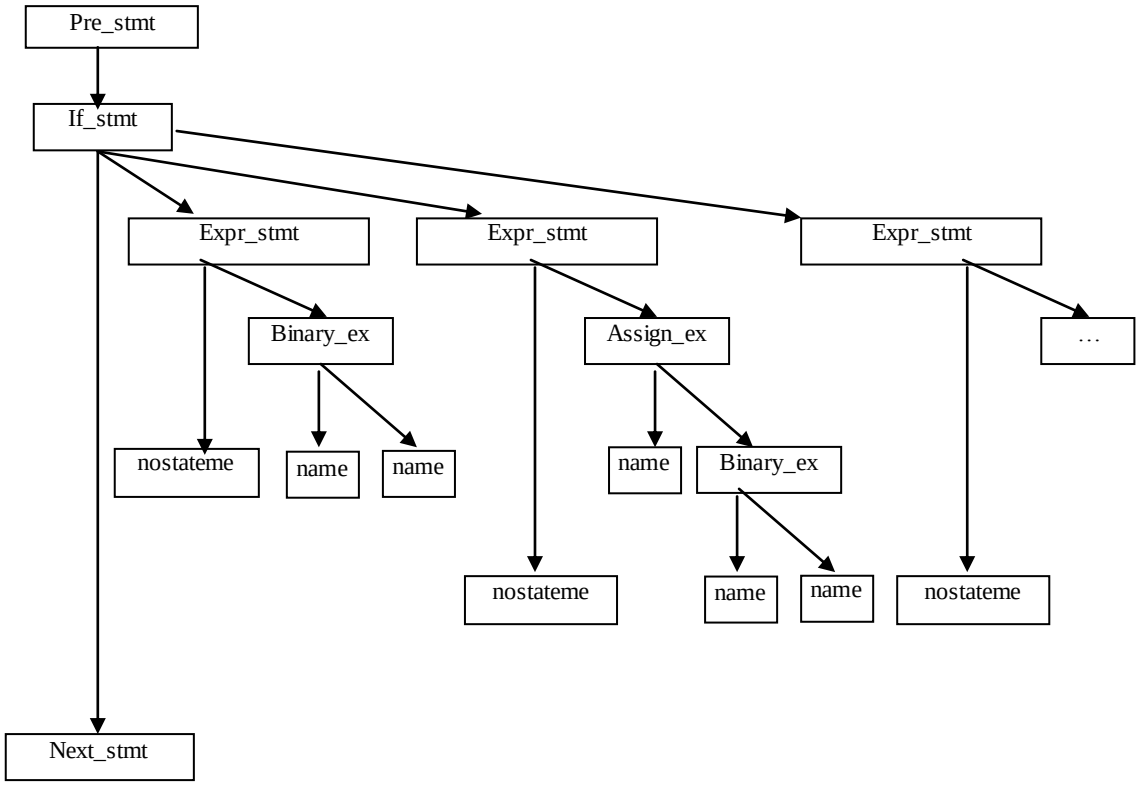

Fig. 1 Abstract syntax tree of statement "if (a<b) q=c+d else q=e+f"

3. Uninitialized variable detecting algorithms based on control flow graph

The inspection points identified algorithms is the key point of this method. The quality of this testing method lies on the efficiency of the algorithms.

3. 1 Uninitialized variable faults analysis

The fault of uninitialized variable can be defined as using of automatic variable before it is initialized. For example:

```
121 char *readline(char *buf) {
122     char c;
123     char *savebuf = buf;
124     while (c != EOF && (c = getchar()) != '\n' )
125         *buf++ = c;
126     *buf = '\0';
127     if ( c == EOF && buf == savebuf
128         return NULL;
129     else
```

```
130             return savebuf;
131 }
```

In this example the local variable c is checked for having the value of EOF before it is loaded with the next character. Because c is not initialized the first time the loop is executed, this is a use of an uninitialized variable.

Another example:

```
100     int a[10]={0,2,4,6},x;
101     scanf("%d",x);
102     for (i=0;i<10;i++)
103         if(x>a[i]&&x<=a[i+1])
104   {   for(j=9;j>i+1;j--)   a[j]=a[j-1];
105                 a[j-1]=x;
106         }
```

In this example the array variable a defined on line 100 only be initialized partly. Then the use on line 103 may be cause uninitialized variable faults.

There are some other conditions may result in this fault type. We do not show them one by one here.

3. 2 Detecting algorithms

In fact, defecting algorithms is the main point for system efficiency and usability of the testing service. Every fault type must have a corresponding defecting algorithms.

The algorithms for uninitialized variable can be designed as below:

Control flow graph of the program be described as G=（N， E）, in which V would be the set of the nodes and E would be the set of the edges. And e=(T(e),H(e)) means the twins of neighboring node, in which the edge e leaves the node T(e) and reaches the node H(e).

① For some variable i, its defining node be marked as Di, assigned node be marked as Initialization (i), and using node be marked as Use(i).

②For all assigned node $n, n \in Initializtion(i)$, if the condition of T(e)＝n or H(e)＝n is true, deleting the node and its link edges in the control flow graph. Then get the sub-graph $G'(N', E')$.

③For using node $n \in Use(i)$, if in the sub-graph $G'(N', E')$, there is one or more passageway from its defining node Di to itself n, an inspection point(IP) would be reported. That means that this variable i may be using without initialization.

There are two targets in evaluating the automated detecting algorithms. One is its complication degree in time. The other is veracity and integrality of the faults report. Our algorithms consider all these targets.

4. Conclusion

Software failures are expensive and time consuming to detect, which cause significant damage directly or indirectly to end users. ASI provides a fast and cost-effective way to improve software quality.

We have build a prototype system for the fault type described above. During using of this tool, we find that there are many advantages of this method, such as working without any test case, faults can be located and so on. All these points are the user concerned and limited in other methods.

References

[1] David Evans. Static Detection of Dynamic Memory Errors[C]. USA: ACM Conference on Programming Language Design and Implementation, 1996. Page 44-52.
[2] Nurit dor., Michael Rodeh, Mooly Sagiv. Detecting Memory Errors via Static Pointer Analysis. USA: ACM Special Interest Group on Programming Languages, 1995. Page13-22
[3] Automated Software Inspection. Technical White Paper.
[4] Dr. Josef Grosch. C++ Abstract Syntax. Technical White Paper.