Macro-like Instrumentation Grammar for Boolean Expressions

Zhenyu Zhang Zhongxing Xu
State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences
zhangzy@ios.ac.cn *xzx@ios.ac.cn*

Zhifang Liu Xiaopeng Gao
School of Computer Science and Technology
Beihang University
iuma29@gmail.com *gxp@buaa.edu.cn*

***Abstract*—Boolean expression is a basic programming element used to evaluate the truth-values of conditions or their combinations. While Boolean expressions may have complicated logical structures, instrumenting them often needs heavyweight transformation to source code or work with low-level program implementation, which results in cumbersome code and great difficulties to maintenance. As a result, previous instrumentation is often conducted using automatic mechanism, left as the last integration step, and needs to be redone once the source code has changes. It is inflexible and not interactivable for collaborative work. In this paper, we compare several existing popular instrumentation methods and propose a friendly approach, which adds least prefix and postfix to Boolean expressions, simply wraps all conditions, and preserves all Boolean operators if any. Our method works at source-code level yet has a macro-like grammar. It is human maintainable so that programmers may manually and cooperatively modify code by instrumenting interested Boolean expressions like operating macros. We elaborate on the grammar of our method and give empirical evaluation to its performance. Our method has been used in some realistic industrial and research projects successfully.**

Key words-Boolean expression, instrumentation

I. INTRODUCTION

Boolean expressions are composed of Boolean variables, relational expressions, and logical operators applied to them [2]. It is a basic programming element used to control program switches, specify termination criteria for looping, or capture program states [8].

Boolean expressions having forms of "$!B$", "$B\&\&B$" or "$B||B$" are intuitively called compound Boolean expressions. A compound Boolean expression combines conditions with logical relations. This property increases the elegance of code with power of expression; thus is adopted by many advanced programming languages like C++, and Java. Boolean expressions (compound ones in particular) reflect versatile program information and are extensively common in program implementations [9]. Many techniques, such as program analysis [7][9][15], bug fix [24], verification [8], testing [4] [13][17][23], and fault localization [1][6][19][20][28], rely on feature spectra of Boolean expressions. To investigate the feature spectra of Boolean expressions, instrumentation is a premise of these techniques and becomes more and more important [21].

Previous instrumentation is performed at different stages and different levels of program presentations. Lightweight source-code-level instrumentation [14] is independent to the compilers used to generate binary code, since it modifies source code directly. Program transformation [11] is also popular to gain powerful functionalities. Another two alternatives intermediate instrumentation [5] and binary instrumentation [22] are easy to manipulate program elements since they work at some low-level program presentations. Dynamic instrumentation [21] works at runtime and suffers no limitation of static analysis. Instrumentation methods having different characteristics may favor different application domains. However, many applications, such as wireless sensor network, engine controllers, automotive, and avionics systems, need to work in a resource-stringent, embedded, or lightweight environment (e.g., limited CPU capability and restricted memory usage [3]). As far as we know, there is no published work addressing the instrumentation problem for Boolean expressions in such resource-stringent application domains.

In this paper, we propose a lightweight source-code-level instrumentation method, which adds prefix and postfix to a Boolean expression, wraps all its conditions, and preserves logical operators if any. Since we embed as least as revision into the code, there is no need of program transformation. As a result, the instrumented code is human maintainable so that programmers may manually instrument interested Boolean expressions and continue work directly on the instrumented code. In a collaborative team work, programmers may also submit the instrumented code continuously and do not need to mark interested sites and wait to the last step of integration.

Our contribution in this paper is threefold. (i) It is the first time that the problem of Boolean expression instrumentation in resource-stringent applications is raised. (ii) We use formal grammar to propose a lightweight source-code-level instrumentation method, which generates human maintainable code to facilitate collaborative work. (iii) We empirically report the performance of our method. The results show that it is very effective and efficient.

The rest of the paper is organized as follows. Section II gives preliminaries. Section III motivates our work. Section IV elaborates on our method, followed by Section V and VI, which respectively give discussion and evaluation. Section VII lists related work. Section VIII concludes the paper.

II. PRELIMINARIES

A Boolean expression has the grammar in Equation (1) and (2), where B and R stand for Boolean expression and relational expression, respectively [2].

978-1-4244-5392-4/10/$26.00 ©2010 IEEE

$$B \xrightarrow{def} !B \mid B\&\&B \mid B||B \mid A \quad (1)$$

$$A \xrightarrow{def} R \mid true \mid false \quad (2)$$

Boolean expressions having form of "*A*" are called ***atomic Boolean expression***. An atomic Boolean expression evaluates a ***condition***. Expressions having form of "`!`*B*", "*B*`&&`*B*" or "*B*`||`*B*" are called ***compound Boolean expressions***. A compound Boolean expression evaluates a combination of conditions.

III. MOTIVATION

Figure 1 shows a program fragment, which ensures the last two zero elements of an array. A compound Boolean expression is used in the `if`-statement on line 1. Conditions "`buf[len-1]!=0`" and "`buf[len-2]!=0`" check the values of the last and the second last element of the array, respectively. They are connected by an OR logical operator "`||`", which means that line 4 and line 5 will be triggered when either "`buf[len-1]!=0`" or "`buf[len-2]!=0`" evaluates true. On the other hand, before they evaluate, condition "`len>=2`" is evaluated in advance. It is connected to "`buf[len-1]!=0 || buf[len-2]!=0`" by an AND logical operator "`&&`", which ensures that there must be at least two elements in the buffer, before checking them.

While source-code-level instrumentation method is a popular choice, trivial solutions may not be effective. For example, Figure 2 shows a straightforward approach, which iterates each condition of the Boolean expression in Figure 1, and logs their evaluation results. A threat is that evaluating "`buf[len-2]!=0`" on line 3 of Figure 2 may cause an array bound exceeding exception if "`len>=2`" evaluates false.

Figure 3 expands the compound Boolean expression in Figure 1. Note that the original `if`-structure has been expanded to 3 `if`-statements, 3 `else`-statements, and 6 basic blocks [26]. Such a program transformation is not acceptable since the code becomes unmaintainable. The original compound Boolean expression structure is missing. Moreover, the rationale of the original logical relations becomes obscure. When a programmer needs to alter the instrumented code, he has to work on the original program and performs instrumentation again after revision. For the same reason, when a programmer wants to instrument interested Boolean expressions to collect logs at the tester side, he has two choices. The first is to instrument those interested Boolean expressions, submit the instrumented code to a CVS or a SVN server, and wait till the last integration step to work out binary code. The second is to preserve the code and compose some kind of instrumentation profile for integrating or compiling references. The former makes the instrumented code unmaintainable to others. The latter does not break the elegance of the code but need no-standard grammar or causes trouble of reuse.

```
1  if (len>=2 && (buf[len-1]!=0 ||
2                 buf[len-2]!=0))
3  {
4      buf[len-1]=0;
5      buf[len-2]=0; }
```

Figure 1. A Boolean expression sample

```
1  bool exp1=(len>=2);
2  bool exp2=(buf[len-1]!=0);
3  bool exp3=(buf[len-2]!=0);
4
5  if (exp1) log ("1T"); else log ("1F");
6  if (exp2) log ("2T"); else log ("2F");
7  if (exp3) log ("3T"); else log ("3F");
8
9  if (exp1 && (exp2 || exp3)) {
10     buf[len-1]=0;
11     buf[len-2]=0; }
```

Figure 2. An iteration instrumentation

What if we may preserve the compound Boolean expression structure and revise the code as least as possible? It may mean maintainable instrumented code, enabling of manual instrumentation (no need to use an automatic tool when there are few interested Boolean expressions), and interactivable collaboration work.

IV. OUR INSTRUMENTATION METHOD

In this section, we show the instrumentation syntax, explain the morphology, elaborate on the semantics, and give complexity analysis.

A. Syntax and morphology

Suppose *E* is the entire Boolean expression, which is defined by grammar in Equation (3) where *B* is described using grammar in Equation (1).

$$E \xrightarrow{def} B \quad (3)$$

Our instrumentation grammar is described as *I* in Equation (4), where **entr**, **eval**, and **leav** are names of procedures. The symbols *W* and *T* are synthesized by applying rewriting rules in Equation (5) and Equation (6) on *E*, respectively.

$$I \xrightarrow{def} \boldsymbol{entr}(T)\ \&\&\ W\ \&\&\ \boldsymbol{leav}(true)\ ||\ \boldsymbol{leav}(false) \quad (4)$$

$$A \xrightarrow{rewrite} \boldsymbol{eval}(A) \quad (5)$$

$$A \xrightarrow{rewrite} \_ \quad (6)$$

We use examples in Figure 1 to illustrate the rewriting rules. As a result of applying Equation (5) on *E*, the conditions "`len>=2`", "`buf[len-1]!=0`", and "`buf[len-2]!=0`" are respectively rewritten to "`eval(len>=2)`", "`eval(buf[len-1]!=0)`", and "`eval`

```
1  bool tag=false;
2
3  if (len>=2) {
4      log ("1T");
5      if (buf[len-1]!=0) {
6          log ("2T");
7          tag=true;
8    } else {
9        log ("2F");
10       if (buf[len-2]!=0)) {
11           log ("3T");
12           tag=true;
13       } else log ("3F"); }
14 } else log ("1F");
15
16 if(tag) {
17     buf[len-1]=0;
18     buf[len-2]=0;
19 }
```

Figure 3. An expansion instrumentation

TABLE 1. STATISTICS OF SUBJECT PROGRAMS

	Real-life versions	Program Description	LoC[1]	#CBE[2] / BE[3] / #C[4]	Code size explosion[5]	Execution time explosion[6]
flex	2.4.7–2.5.4	lexical parser	11.78k	91 / 700 / 807	10.67%	3.20%
grep	2.2–2.4.2	text processor	10.92k	147 / 985 / 1162	14.42%	4.58%
gzip	1.1.2–1.3	compressor	6.36k	76 / 523 / 611	11.30%	2.55%
sed	1.18–3.02	text processor	8.06k	92 / 615 / 728	11.93%	2.85%
				Mean:	12.08%	3.30%

1: average lines of code;
2: average number of lines containing compound Boolean expressions;
3: average number of lines containing Boolean expressions;
4: average number of conditions;
5: average ratio of code explosion to original code size
6: average ratio of execution time explosion to original execution time

`(buf[len-2]!=0)`". In consequence, *W* is generated as "`eval(len>=2) && (eval(buf [len-1]!=0) || eval(buf[len-2]!=0))`".

As a result of applying Equation (6) on *E*, the conditions "`len>=2`", "`buf[len-1]!=0`", and "`buf[len-2]!=0`" are rewritten to "_", "_", and "_", respectively. Finally, *T* is generated as "`_&& (_||_)`".

In consequence, the final instrumentation result of the Boolean expression in Figure 1 is shown in Figure 4.

B. Semantics for instrumentation words

We first recall that *I* in Equation (4) is the instrumentation grammar for an input Boolean expression *E*. The terms "`entr`", "`eval`", and "`leav`" are names of procedures.

The procedure "`entr`" receives argument *T*, which indicates the logical rationales of *E*. The functionality of "`entr`" is to allocate space to log evaluations to Boolean expression *E*. "`entr`" always returns true so that the following Boolean expression *W* will be never short-circuited.

All conditions in *W* are rewritten in form of "**eval**(*A*)". At the same time, *W* inherits the logical structure of *E*. Each time a condition *A* is evaluated, the procedure "`eval`" receives a truth-value typed argument as value of *A*, and returns the same truth-value. The functionality of "`eval`" is to store the input truth-value in the previously allocated space. It is clear that *W* evaluates as *E* does.

If the evaluation result of *W* is true, "**leav**(*true*)" is evaluated and "**leav**(*false*)" is short-circuited. Accordingly, the evaluation sequence [28] for "**entr**(*T*)", "*W*", "**leav**(*true*)", and "**leav**(*false*)" is ⟨*true*, *true*, *true*, ⊥⟩. The counterpart is that "**leav**(*true*)" is short-circuited and "**leav**(*false*)" is evaluated, when the evaluation result of *W* is false. The corresponding evaluation sequence is ⟨*true*, *false*, ⊥, *false*⟩. The procedure "`leav`" receives a truth-value argument and uses it as the return value. The functionality of "`leav`" is to assemble the stored evaluation results with the symbol *T*. After the execution of "`leav`", the previously allocated and used space is released.

The algorithms for "`entr`", "`eval`", and "`leav`" are simple. Pseudo code can be found in our technical report [25]. We use the C language frontend tool clang [10] to set up our experiment. The related experiment setup steps can be also found in our technical report [25].

```
1  if (foo("_&&(_||_)") && (
2      bar(len>=2) && (bar(buf[len-1]!=0) ||
3                      bar(buf[len-2]!=0))
4      ) && baz(true) || baz(false))
5  {
6      buf[len-1]=0;
7      buf[len-2]=0; }
```

Figure 4. The final instrumentation

C. Complexity analysis

Suppose there are *m* Boolean expressions and *n* conditions, the space complexity is **O**(*m*+*n*) (or **O**(*n*), since *m*<*n*).

V. DISCUSSIONS

A previous observation is that about 1/14 of all the statements contain Boolean expressions [9]. Among them, about 1/7 are compound ones [28]. Boolean expressions are widely distributed in programs, while compound ones are also common. Previous study shows that many software bugs are related to Boolean expression [12]. Paying effort to this research is of practical impact.

From Equation (5), we know that *E* is rewritten to *W* with every atomic Boolean expression *A* being substituted by **eval**(*A*), so that *W* has the same logical structure with *E*. For an atomic Boolean expression *A* in *E*, the instrumented word **eval**(*A*) is called as far as *A* is supposed to be evaluated in *E*. On the other hand, **eval**(*A*) will not be called if *A* is supposed to be short-circuited. Since W and E have identical logical structure and identical values of evaluation results for conditions, our static instrumentation accurately adapts to dynamic program behavior.

Our instrumentation method have been used in a Hong Kong ITF project [16], and many research projects about program synthesize [27], fault localization [26][28], and regression testing [18].

VI. PERFORMANCE EVALUATION

In total, four UNIX utility programs, namely, flex, grep, gzip, and sed, are used as subject programs. They are real-life medium-sized programs, and have been adopted to evaluate program analysis, debugging, and testing techniques, such as [26]. Table 1 shows the characteristics of subjects.

Table 1 shows the performance impact of instrumentation, including average code size explosion, and average execution time explosion. Take the program flex as an example. The average code size exploit rate is 10.67%; while the average execution time extension is 3.20%. Considering different programs, the mean code size explosion and execution time explosion are 12.08% and 3.30%, respectively.

In this experiment, we have the following observations. Our instrumentation increase program code size at ratio of

about 12.08%. The code increase is approximately proportional to number of Boolean expressions. Our instrumentation takes in performance slowdown to the instrumented program. The performance slowdown is about 3.30% of execution time.

VII. Related work

In some resource-stringent, lightweight, and embedded application domain, applications often need to run with limited CPU capability and restricted memory usage [3]. To address these issues, Zhang et al. [27] investigate the program synthesize problem for Wireless Sensor Network applications.

Filman and Havelund [14] discuss source-code-level instrumentation. Simple source-code-level instrumentation can be done by manual work, such as Liu et al. did in their work [20]. However, they do not address compound Boolean expressions. For some languages such as Java, source code can be compiled into intermediate code. Intermediate-level instrumentation [5] takes in more functionality but depends on features of intermediate presentations. Nethercote and Seward propose a heavyweight binary-level instrumentation method [22]. Binary-level instrumentation is also a powerful instrumentation technique, since it works on the lowest-level program presentation. But it is neither maintainable nor suitable for collaborative work. Luk et al. [21] propose a dynamic instrumentation, which uses a just-in-time compiler to insert code before executing a binary instrument.

Many program analysis, debugging, testing, and verification techniques rely on Boolean expressions in programs. For example, Ball et al. [7] propose the use of edge profile in finding hot paths. CBI [19] and SOBER [20] use Boolean expressions as predicate to support fault localization. DES [28] empirically finds that considering in the short-circuiting evaluation rule may increase their effectiveness.

VIII. Conclusion

Many applications, such as wireless sensor network, need to run with stringent resources. In this paper, we propose a lightweight source-code-level instrumentation approach, to instrument the Boolean expressions in programs. We analyze the theoretical complexities of our method, and empirically evaluate its performance using four realistic UNIX utility tools. Our observation is that our method is very effective and efficient. Future work may include adapting our method to multi-thread environment and reducing the overhead.

Acknowledgment

This research is supported by the National High Technology Research and Development Program of China (project no. 2007AA01Z145).

References

[1] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. “On the accuracy of spectrum-based fault localization”. In TAICPART-MUTATION 2007.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. “Compilers, principles, techniques, and tools”. *Addition Wesley*.

[3] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. “Wireless sensor network: a survey”. *Compuer Networks*.

[4] J. H. Andrews, L. C. Briand, and Y. Labiche. “Is mutation an appropriate tool for testing experiments?”. In *ICSE 2005*.

[5] K. Arnold, J. Gosling, and D. Holmes. “The Java™ programming language”. *Prentice Hall*.

[6] P. Arumuga Nainar, T. Chen, J. Rosin, B. Liblit. “Statistical debugging using compound Boolean predicates”. In *ISSTA 2007*.

[7] T. Ball, P. Mataga, and M. Sagiv. “Edge profiling versus path profiling: the showdown”. In *POPL 1998*.

[8] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. “Symbolic model checking without BDDs”. In *TACAS 1999*.

[9] D. Binkley and M. Harman. “An empirical study of predicate dependence levels and trends”. In *ICSE 2003*.

[10] “Clang” C language family frontend for LLVM. Available at http://clang.llvm.org/. Last accessed on Apr 20, 2010.

[11] G. A. Cohen, J. S. Chase, and D. L. Kaminsky. “Automatic program transformation with JOIE”. In *USENIX 1998*.

[12] J. A. Durães and H. S. Madeira. “Emulation of software faults: A field data study and a practical approach”. *TSE*.

[13] S. Elbaum, D. Gable, and G. Rothermel. “The impact of software evolution on code coverage information”. In *ICSM 2001*.

[14] R. E. Filman and K. Havelund. “Source-Code Instrumentation and Quantification of Events”. In *AOSD 2002*.

[15] S. L. Graham, P. B. Kessler, and M. K. Mckusick. “Gprof: a call graph execution profiler”. In *SIGPLAN 1982*.

[16] Hong Kong ITF. “An extensible fault-based predicate testing toolset for Wireless Sensor Network software application”. Available at http://www.itf.gov.hk/eng/ Search/proj_Detail.asp?Prj_code=1901. Last accessed on Apr 20, 2010.

[17] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. “Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria”. In *ICSE 1994*.

[18] B. Jiang, Z. Zhang, W. K. Chan, and T. H. Tse. “Adaptive random test case prioritization”. In *ASE 2009*.

[19] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, M. Jordan. “Scalable statistical bug isolation”. In *PLDI 2005*.

[20] C. Liu, L. Fei, X. Yan, S. P. Midkiff, J. Han. “Statistical debugging: a hypothesis testing-based approach”. *TSE*.

[21] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. “Pin: building customized program analysis tools with dynamic instrumentation”. In *PLDI 2005*.

[22] N. Nethercote and J. Seward. “Valgrind: a framework for heavyweight dynamic binary instrumentation”. In *PLDI 2007*.

[23] Y. T. Yu and M. F. Lau. “A comparison of MC/DC, MUMCUT and several other coverage criteria for logical decision”. *JSS*.

[24] X. Zhang, N. Gupta, and R. Gupta. “Locating faults through automated predicate switching”. In *ICSE 2006*.

[25] Z. Zhang. “Macro-like instrumentation grammar for boolean expressions”. Technical report ISCAS-LCS-10-09. State key laboratory of computer science, ISCAS.

[26] Z. Zhang, W. K. Chan, T. H. Tse, B. Jiang, and X.Wang. “Capturing propagation of infected program states”. In *ESEC 7/FSE-17*.

[27] Z. Zhang, W. K. Chan, T. H. Tse, H. Lu, and L. Mei. “Resource prioritization of code optimization techniques for program synthesis of wireless sensor network applications”. *JSS*.

[28] Z. Zhang, B. Jiang, W. K. Chan, T. H. Tse, and X. Wang. “Fault localization through evaluation sequences”. *JSS*.