International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:4, No:2, 2010

BugCatcher.Net: Detecting Bugs and Proposing Corrective Solutions

Sheetal Chavan, P. J. Kulkarni, and Vivek Shanbhag

***Abstract*—**Although achieving zero-defect software release is practically impossible, software industries should take maximum care to detect defects/bugs well ahead in time allowing only bare minimums to creep into released version. This is a clear indicator of time playing an important role in the bug detection. In addition to this, software quality is the major factor in software engineering process. Moreover, early detection can be achieved only through static code analysis as opposed to conventional testing. BugCatcher.Net is a static analysis tool, which detects bugs in .NET® languages through MSIL (Microsoft Intermediate Language) inspection. The tool utilizes a Parser based on Finite State Automata to carry out bug detection. After being detected, bugs need to be corrected immediately. BugCatcher.Net facilitates correction, by proposing a corrective solution for reported warnings/bugs to end users with minimum side effects. Moreover, the tool is also capable of analyzing the bug trend of a program under inspection.

***Keywords*—**Dependence, Early solution, Finite State Automata, Grammar, Late solution, Parser State Transition Diagram, Static Program Analysis.

I. Introduction

BUGS can be defined as a path through the code when gets executed causes either a run-time exception or an incorrect result with certain input values. However, unlikely an error, bug occurs due to specific inputs which developers didn't try at the time of developing whereas error occurs irrespective of inputs. As .NET® languages are gifted with managed execution environment [3], there are mechanisms to handle runtime exceptions. However, handling such errors at runtime offers little advantage. In addition, the effort needed to get rid of a bug increases with the time a bug spends in a software product. Hence, it is most crucial to detect them as early as possible. As discussed in previous paper [1], the static analysis techniques used for early detection generally try to identify the pattern associated with the bugs to find out all the occurrences all over the code base unlike testing. Usually, meeting unrealistic deadlines and insufficient testing time account for significant number of hidden bugs in the software released. However, at the very root level, the main cause of bug occurrence can be either the grammar of the programming language itself or the violation of dependencies. These causes are discussed briefly in section III. There are various tools available to detect bugs statically in different language source code. Section II gives brief information about these tools. Whereas rest of the paper is organized as follows. Section III describes the bug detection approach and section IV presents strategies used for proposing corrective solutions to the end users. The bug trend analysis carried out to propose overall solution is discussed in detail in section V. Results obtained and analysis performed are given in sections VI and VII respectively. Finally, section VIII presents future extensions and makes conclusion of the work done.

Sheetal Chavan is with Philips Electronics India Ltd., Bangalore-560045, India (phone: +91 80 4189 1644; fax: +91 80 4189 1000; e-mail: sheetal.chavan@philips.com).

Dr. P. J. Kulkarni is with Walchand College of Engineering and Technology, Sangli, India (e-mail: pjk_walchand@rediffmail.com).

Vivek Shanbhag was with Philips Electronics India Ltd. He is now with IIIT Bangalore (e-mail: vivek.shanbhag@gmail.com).

II. Related Work

There are ample amount of bug detectors available for C and C++. One of these is LCLint, which reports inconsistencies between a program and its specification [4]. Further, the ASTLog tool [5], which looks for syntactically suspicious code patterns, has been extended into the PREfast tool [6] & is used extensively within Microsoft as a bug pattern detector.

Several detection tools for Java language also exist. One of the useful tools is PMD [7], which checks for patterns in the abstract syntax trees of parsed Java source files. Another tool, FindBugs [2] uses a series of ad-hoc techniques designed to balance precision, efficiency, and usability. One of the main techniques FindBugs uses is to match source code syntactically to known suspicious programming practice, in a manner similar to ASTLog. In some cases, FindBugs also uses dataflow analysis to check for bugs. JLint [8], like FindBugs, analyzes Java byte-code, performing syntactic checks and dataflow analysis. JLint also includes an inter-procedural, inter-file component to find deadlocks by building a lock graph and ensuring that there are never any cycles in the graph. Further, Bandera [9] is a verification tool based on model checking and abstraction. To use Bandera, the programmer annotates the source with specifications describing what should be checked, or no specifications if the programmer only wants to verify some standard synchronization properties.

There are various static analysis tools available for .NET® languages also. FxCop is one of the tools which analyzes managed code assemblies (code that targets the .NET® Framework common language runtime) and reports information about the assemblies, such as possible design, localization, performance, and security improvements [10].

Many of the issues concern violations of the programming and design rules set forth in the Design Guidelines for Class Library Developers. Another tool, StyleCop provided by Microsoft® ensures that C# code incorporates style and consistency rules. StyleCop is very beneficial in conducting code reviews [11]. Further, CodeIt.Right offers the means to correct violations in C# and Visual Basic code automatically [12]. It carries out static code analysis with configurable rule sets to find code issues. This is one of the tools, which is used to ensure that code written conforms to the best coding guidelines, thus helping in writing quality code.

Resharper is yet another tool provided by JetBrains, which analyzes and highlights errors in C# code (up to C# 3.0) while typing itself, without having to compile the code first [13]. It also helps to solve problems instantly, by suggesting quick fixes for most errors. User can analyze code both in a current file and throughout the entire solution. Finally, NStatic [14] can detect errors like complex expressions (including function calls) that evaluate to constants; assignment to a variable is same as current value, redundant parameter, infinite loops, etc. Nevertheless, NStatic only supports C# as it is source code dependent.

Out of the tools that are discussed here, Findbugs serves as motivation of our work of finding similar bugs in .NET® source.

III. Causes of Bug Occurrence

A. Programming Language Grammar

As mentioned earlier, the grammar of any programming language can also be a potential source of bugs. This grammar can be well described by a context-free grammar. A context-free grammar (G) [15] can be formally defined as a 4-tuple:

$$G = (V, \Sigma, R, S) \quad (1)$$

where V is a finite set of non-terminal characters or variables. They represent different types of phrase or clause in the sentence. Σ is a finite set of terminals, disjoint with V, which make up the actual content of the sentence. R is a relation from V to $(V \cup \Sigma)^*$ such that any $w \in (V \cup \Sigma)^*$: $(S, w) \in R$. S is the start variable, used to represent the whole sentence (or program). It must be an element of V.

Some of the bugs arise from flaws in the grammar of the programming language or some are introduced due to wrong concatenation of non-terminals (symbols).

1) Wrong Concatenation of Non-terminals

One such bug, which results from wrong concatenation of non-terminals, is **'Floating-Point Equality'**, where a check is made to see if two floating-point values are equal. Corresponding grammar rules are given below.

Equality expression: Operand (Equality operator) Operand
Equality Operator: = = | !=
Operand: Literal | Identifier
Idenitifier: predefined-type.Identifier
predefined-type: bool | byte | char | decimal | double | float | int | long | object | sbyte | short | string | uint | ulong | ushort

Here, if equality expression is used with operands of type float/double then result will always be false due to floating-point truncation. Moreover, if this expression is used as a condition to take some decisions for following particular path in a program then always one particular path will be executed.

2) Flaws

Any programming language grammar may have some or other flaw due to time and space constraints that are imposed on compilers. At some places grammar is too vague such that it can't specify certain conditions like the bug pattern which arises from a flaw in the grammar rule – '**Useless control flow'**, where control flow continues onto the same place i.e., to the same or following line regardless of whether or not the branch is taken.

```
public void Method(){
        int var = 3; if (var = 2) ;
  }
```

An erroneous grammar rule is

If statement: if expression then statement

This rule allows null statement also to pass in then-part of an if-expression. Hence, to correct this, rule should have been as follows:

If statement: if expression then statement other than null

Here grammar rule becomes general and is not able to put specific condition in if statement.

B. Violation of Dependencies

Another major cause of bugs can be violation of dependencies, control or data. Most of the bugs result from violation of some dependency in a program.

One of the bug patterns is **'Dead Parameter'**, where a parameter is overwritten in a very first instruction in a method ignoring the actual value passed to the method.

```
public int mName(int x){
x = 3; return x;
}
```

If accepting a parameter/argument in a method locally is considered as a 'Read Operation', and then if the parameter read is written in a very first instruction in a method, Write-After-Read (WAR) dependency is found, which results in dead parameter bug. Thus instead if first instruction is read, i.e., in first instruction the value passed to this method is copied to some local variable and after this parameter is written then it will be Read-After-Read (RAR) dependency, which will not result in a bug. It clearly shows that Dead Parameter is a result of violation of Read-After-Read dependency.

Another such pattern is **'Redundant Null-check'**, where a value of reference type is checked against null after it's dereference.

```
public Car addCar(Car c2) {
      Car temp=new Car("");
      c2.desc = "Bolero";
      if(c2!=null)
         Console.WriteLine("stmt");
}
```

In above code snippet, variable 'c2' of user-defined type

'Car' is checked against null after it is de-referenced/used. Therefore if its value is null, a runtime null de-reference exception will occur at its first de-reference itself or otherwise second null-check is redundant. That means if statement here is not the expected one. As if-statement is an example of control dependency, it clearly shows that this bug is a result of violation of control dependency.

IV. Bug Detection Approach

First the stream of bytes constituting a program is transformed into the corresponding sequence of machine instructions and after that based on this machine code representation, static or dynamic analysis is carried out to extract out the properties and function of the program.

A. Disassembling

In the first step, code is disassembled to recover symbolic representation of program's machine code instructions from its binary representation. In Microsoft .NET® framework, it is carried out through the program **ILDASM** (IL Disassembler).

B. Code Analysis

Code analysis is the second step. It takes a program as input and tries to analyze the behavior of the program [16]. Based on the program's machine code, which is extracted in first step, code sequences that are known to be deviant (or code sequences that violate a given specification of permitted behavior) are identified. The analysis techniques that the tool uses are given in Fig. 1.

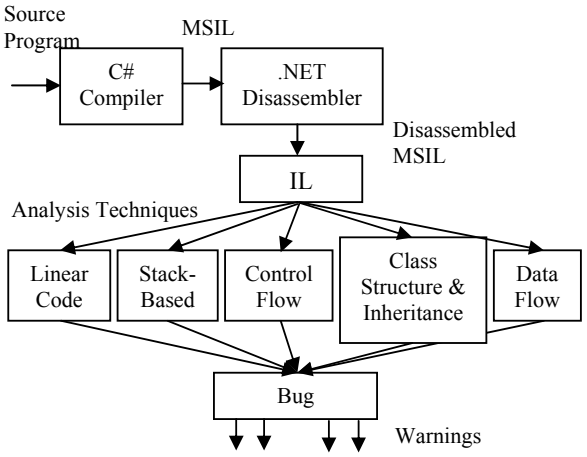

Fig. 1 Analysis Techniques

In addition to analysis techniques, the parser used plays a significant role in bug detection. BugCatcher.Net uses a parser based on Finite State Automata (FSA).

Code analyzing parser can be represented as **a Finite State Automaton (FSA)** [17],

$$A= (Q, \Sigma, \delta, q_0, F) \quad (2)$$

Where Q is the set of states, Σ is input symbols, δ transition function, q_0 is the start state, and F is the set of accepting states. In case of our parser, Σ is different program instructions/statements.

Each of the bugs has a pattern associated with which can be represented by a state transition diagram.

For instance, the parser state transition diagram for bug pattern 'Floating-point equality' is shown in Fig. 2.

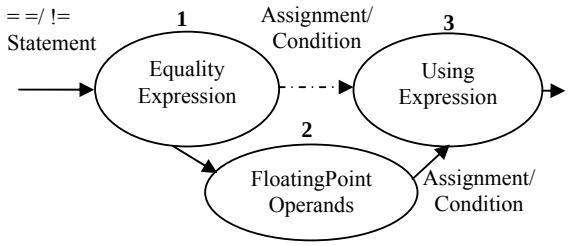

Fig. 2 Parser State Transition Diagram

The corresponding code snippet is given below:

```
float firstVar = 1.3f, secondVar = 1.3f;
if(firstVar == secondVar){ ……..}
```

In this example, *if* condition will never get satisfied due to floating-point truncation. Thus when '==' operator is used, our FSA-based parser enters in state 1 and when the two operands used for equality operator are floating-point then parser enters into state 2. Finally, if such a floating-point equality operation is used to take some decisions in a condition then third state is reached and this confirms the existence of bug.

In Fig. 2, the dotted lines represent the normal non-buggy path while solid lines represent the buggy one. Hence, if after parsing the code under inspection, it follows the buggy path of parser state transition diagram then there is a potential of finding a bug.

Many such detectors are devised for known bug patterns [1]. Some of them are very simple to handle and require simple analysis strategies like linear code scan, stack based or inspection of only class structure and inheritance hierarchy; while some are very tricky, which need complex analysis techniques like data flow and control flow. 'Floating-Point Equality' was comparatively simple pattern; the next section will discuss one of the complex patterns called 'dereferencing the return value of a method call without checking'.

Sometimes the return value from a method is used in the source code without checking against null. If called method returns a value, which is not guaranteed to be non-null, and if this value is used in caller without null-checking, then definitely there is a possibility of getting runtime exception. The detector used for this bug pattern, **'Method Return Value Checker'**, determines if, a function returns a value, that value is tested before using. Use of a value can be defined as passing it as a parameter to a function, using it in a calculation, de-referencing or overwriting with some other value before testing [18].

Testing a return value means that some control flow decision relies on the value. The checker does a data-flow analysis on the variable holding the returned value only to the point of determining if the value is used before being tested. The checker simply identifies the original variable; the

returned value is stored into and determines the next use of that variable. If the variable during its next use is an operand to a comparison in a control flow decision then the return value is deemed to be tested before being used. If the variable is used in any way before being used in a control flow decision then the value is deemed to be used before being tested.

The pattern associated with it is can be well understood by the parser state transition diagram given in Fig. 3.

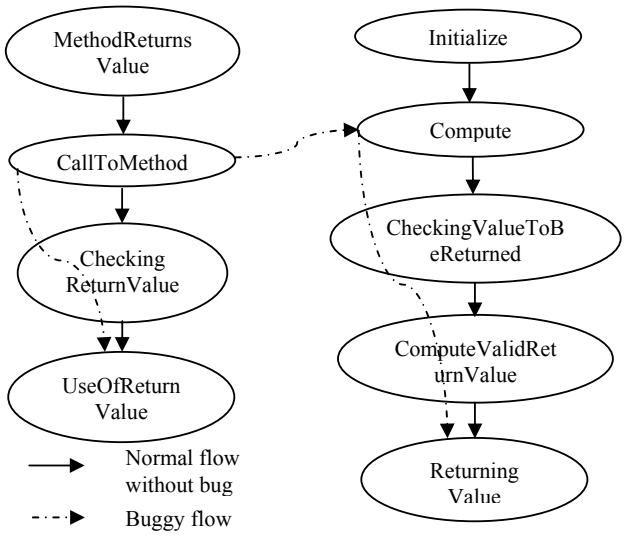

Fig. 3 Parser State Transition Diagram for dereferencing of return value without checking

The checker keeps track of those methods, which return an object of type user-defined type (user defined class). In addition, for each call, those callers are recorded which use the return value without checking against null. Thus, if an instance where a method returns a reference of user-defined type is found and that return reference is used without being checked in the caller then the warning is reported to the user.

V. Proposing Bug Corrective Solutions

For proposing a solution, the user-program should be changed in such a way that the changed program-version does not pass through the states of corresponding **Parser State Transition diagram (PSTD)** in given order i.e. it is required to break the chain in PSTD. There are two options to achieve this.

1. Change such that program does not pass through any of the states before it enters the final concluding state of the PSTD.

Suppose that there are N states in a PSTD. Then propose a change C for program P corresponding to a state S as follows

$$C(P) = Change_{Si}(P) \quad i = 1,\ldots, N\text{-}1. \qquad (3)$$

2. Alter the execution order in which program passes through the PSTD states such that changed order is not able to produce the bug. Impose the execution order O out of total 2^N orders for program P, as follows:

$$O(P) = Order_i(P) \qquad (4)$$

Where $i = 1,\ldots, 2^N$ and $i \in \{\text{Bugs-free Orders}\}$

Out of the two options above, first option was chosen as second has extra processing overhead. In addition, there is an extraneous overhead of finding out bugs-free orders.

In state based solution approach, a decision is taken about which state to handle. The options available are as follows:

a. By default, take care of immediately previous state in a parser state transition representation of a bug pattern.

b. Determine a state, which needs to be taken care of using the technique based on computing dependence. In this technique, dependence of previous and previous to previous states is calculated and then the solution which has minimum possible dependence is proposed. Following section gives the information about how to calculate the dependence of solutions.

A. Calculating Dependence

The technique used for calculating dependence is call based and hence it has method level granularity. If proposed solution indicates that change has to be incorporated in a method M, then set of possible impacted methods includes any method, which directly call M, and any method, which is directly called by M.

A method M_a *directly calls* method M_b, if call to method M_b is directly figured out in body of the method M_b.

Let $M_{Calling}(M_i)$ be the set of methods which directly call method M_i and $M_{Called}(M_i)$ be the set of methods which are directly called by M_i. Thus the set $DM(M_i)$ of dependent methods can be given by

$$DM(M_i) = M_{Calling}(M_i) \cup M_{Called}(M_i) \qquad (5)$$

Although only the methods, which are direct dependent, are considered, set can also be formed iteratively. Thus, the same process is repeated for the methods which are called by and which are calling, method M_i.

In case of method return value checker, usually corrective solution proposed, requires change to be done in parameters. Hence, for null dereference checker, granularity is parameter/variable level.

Let $M_{Using}(P_i)$ be the set of methods which use value of parameter/variable P_i, $M_{Passing}(P_i)$ be the set of methods which pass P_i to other methods and $M_{Returning}(P_i)$ is the set of methods which return P_i. Thus the set $DM(P_i)$ of dependent methods on parameter P_i is given by

$$DM(P_i) = M_{Using}(P_i) \cup M_{Passing}(P_i) \cup M_{Returning}(P_i) \qquad (6)$$

Thus, the solution whose |DM| (number of dependent methods) is minimal is preferred. While computing dependences of a particular proposed corrective solution, the tool considers structural changes also.

B. Structural Changes

These changes can be any of the following: Addition, Modification or Deletion of a programming element. Moreover, the solution, which introduces minimal structural changes, is preferred.

Thus above approach is followed to ensure that the

solution presented to the end-user, has minimal side effects.

C. Providing Early Solution

According to well-known saying, "Prevention is always better than cure", waiting for a bug to cripple in software will not be advisable. To avoid such situation, BugCatcher.Net tries to provide a solution to end-user as soon as sufficient confidence or enough confirmation is obtained about occurrence of a bug. Such a solution is referred as an **'Early Solution'.**

Next section highlights our concept of confirmation of a bug.

1) Likelihood and Confirmation of a Bug

The occurrence of a bug gets confirmed as the program passes through all the states in corresponding parser state transition diagram (PSTD) in the same order. e.g., for method return value checker, whenever any program under inspection has some method, which returns a user-defined type then it, enters in a first state of PSTD. However, only finding such method is not sufficient to conclude that program contains the bug. Instead this method should be called somewhere in the code and not even this, its return value should get used in subsequent code chunk.

Thus at first state in PSTD, there is a likelihood of finding a bug. Moreover, this likelihood gets confirmed as one traverse through the entire PSTD in same sequence. Therefore, at final state in each PSTD, there is 100% confirmation of existence of a bug in user program.

Likelihood of a bug can be expressed in terms of equations as follows.

Let N be the total number of states in a PSTD, which are needed to be followed to ensure the existence of a bug B and M be the current state. Then likelihood of occurrence of a bug B, at state M can be given by

$$\text{Likelihood}(B)_M = (100 / N) * M, \quad \text{Where } M = 1, \ldots, N \tag{7}$$

2) Approach

For providing an early solution, the option of handling immediately previous state is followed. e.g., whenever it is observed that the user is trying to use/de-reference a value of user-defined/reference type without checking it against null, the user is intimated about future potential bug by giving him/her the solution immediately, without waiting for 100% confirmation of the existence of bug.

D. Providing Late Solution

The solution, which is determined after the bug has occurred in the software, is termed as a **'Late Solution'**. This solution is based on dependence calculation technique.

Intuitively taking care of immediately previous state in a PSTD is appropriate solution, as it limits the impact of the proposed change to produce any other type of bugs or if put in other words, it limits the side-effects of proposed solution. Nevertheless, in some cases like method return value checker, this approach may not help. There can be a better solution lying in some other state of state transition representation. Hence, the technique based on computing dependences to determine which state to handle is used for late solutions.

Finally, after inspecting whole user program, the user is equipped with overall solution, which is able to make most of the program bugs-free. Next section reveals about overall solution provided.

VI. Proposing Overall Strategic Solution

Initially an attempt is made to understand the program behavior. Then after equipped with whole information about program, our tool detects some potential bugs. The result of the tool can be used to summarize the bug-trend which helps in arriving at a conclusion about the type of bug towards the program is more prone to. Then this conclusion can be used to provide '**overall strategic solution'** to end user. Following section describes about strategic solution derived.

The tool gives different types of warnings to user. The overall strategic solution provided is based on following heuristic:

The solution is based on the type of bug, which appears for more number of times in the output.

Let N be the total number of warnings shown by the tool and M be the number of different types of warnings. Then $B(P_i)$ is the set of warnings obtained as a result of running the tool on program P_i, which is represented as follows:

$$B(P_i) = B_{Type1}(P_i) \cup B_{Type2}(P_i) \cup \ldots\ldots\ldots \cup B_{TypeM}(P_i) \tag{8}$$

$$\Sigma_{j = 1,\ldots, M} (B_{Typej}(P_i)) = N$$

Then the overall solution is based on the bug type

$$\text{Solution}(P_i) \Rightarrow B_{Typej}(P_i), \; | B_{Typej}(P_i)| > N/M \tag{9}$$

VII. Results

BugCatcher.Net is tested against Philips PmsMip(Philips Medical Systems Medical Imaging Platform) and also on different open source projects taken from SourceForge.net [21] site.

TABLE I
Results

Project/Module	No. of Warnings Reported
Philips PmsMip	4297
ITextSharp [22]	1497
Airplaneawar [23]	2
SharpDevelop [24]	46
EulerSharp [25]	19
VNC Viewer [26]	58
SWAT [27]	302
Quartz.Net [28]	310
Nayatel [29]	37
NeuronDotNet [30]	0
TemplateEngine[31]	3

VIII. Analysis

A. Comparison with Existing Tools

The results obtained with this tool are compared with FxCop [10], StyleCop [11] and CodeIT.Right [12] open source freeware static analysis tools for .NET® languages. These tools are compared based on the type of bugs they are capable of detecting. Table II depicts the results of this comparison.

TABLE II
Comparison

Bug Pattern	BugCat-cher.Net	Fx-Cop	Style-Cop	CodeIT.Right
Dead Parameter	Yes	Yes	No	No
Dereferencing of Result of ReadLine() without Checking	Yes	No	No	No
Redundant Interface	Yes	No	Yes	No
Questionable Boolean Assignment	Yes	No	Yes	No
Useless Control Flow	Yes	No	No	No
Null Dereference	Yes	No	No	No
Return Null Reference	Yes	No	No	No
Bad Reminder	Yes	No	No	No
Masking of a Super class Field	Yes	No	No	No
Redundant Nullcheck	Yes	No	No	No
Floating Point Euality	Yes	No	No	No
Equality to NaN	Yes	Yes	No	No

B. Distribution of Bugs over Modules

The distribution of bugs reported by the tool is studied over different software modules to identify those modules, which are more error-prone. This helps in focusing efforts on such modules and achieving better quality software. It is often found that the distribution of errors/faults across software modules is always skewed that is a small number of modules accounts for most of the faults. This behavior is better modeled by **Weibull Distribution** [19] given below.

1) Weibull Distribution

The physicist Waloddi Weibull developed it, which helps in finding distribution of faults over software modules. It is based on the Pareto principle, which says that 20% of population owns 80% of wealth [20]. The cumulative distribution function (CDF) of the Pareto distribution can be defined as,

$$P(x) = 1 - ((\gamma / x)^{\beta}) \qquad (\gamma > 0, \beta > 0). \qquad (10)$$

The CDF of the Weibull distribution can be formally defined as

$$P(x) = 1 - \exp(-(x/\gamma)^{\beta}) \qquad (\gamma > 0, \beta > 0). \qquad (11)$$

The results are tested against Weibull distribution and it is observed that the warnings/potential bugs that the tool reported are also distributed accordingly. The experimental studies are performed on Philips PmsMip project.

Table III gives the number of bugs found in various modules because of running our tool. Fig. 4 shows the percentage of the accumulated number of potential bugs when the modules are ordered by decreasing number of faults.

TABLE III
Bug Distribution

PmsMip Module	Total bugs
FieldService	0
ProcessingService	3
Connectivity	34
Applications	109
Public	120
Tools	180
Database	275
Services	461
Base	606
Tests	1144
Viewing	1365

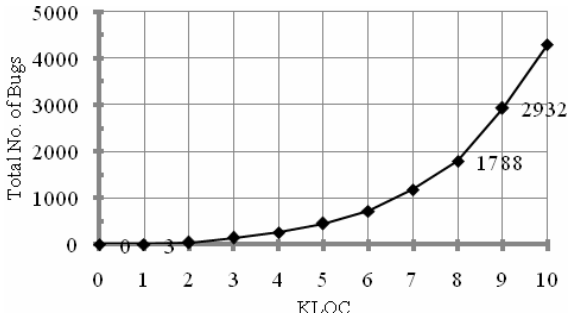

Fig. 4 Cumulative Plot of Weibull Distribution for Philips code-base

C. False Positive Rate

It gives the ratio of number of non-relevant items to total retrieved items, which is given by following expression:

$$\text{False Positive Rate} = \frac{|\{\text{Non-relevant items}\}|}{|\{\text{ Retrieved items}\}|} \qquad (12)$$

In case of our tool, false positive represents a warning reported to user, which is not a real bug. Hence, false positive rate can be given by following expression:

$$\text{False Positive Rate} = \frac{|\{\text{Buggy Warnings}\}|}{|\{\text{ Inspected Warnings}\}|} \qquad (13)$$

Fig. 5 gives the plot of false positive rate against number of warnings or errors inspected for false positives in PmsMip

codebase.

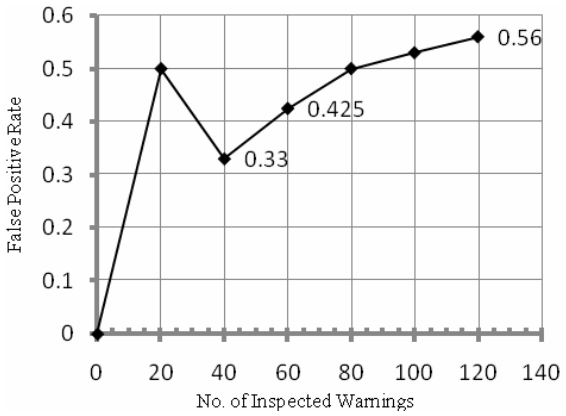

Fig. 5 Cumulative Plot of False Positive Rate Analysis

D. Precision

It gives the ratio of relevant items to the retrieved items:

$$\text{Precision} = \frac{|\{\text{Relevant items}\} \cap \{\text{Retrieved items}\}|}{|\{\text{Retrieved items}\}|} \quad (14)$$

In our case, precision can be expressed as follows:

$$\text{Precision} = \frac{|\{\text{Positive Warnings}\} \cap \{\text{Inspected Warnings}\}|}{|\{\text{Inspected Warnings}\}|} \quad (15)$$

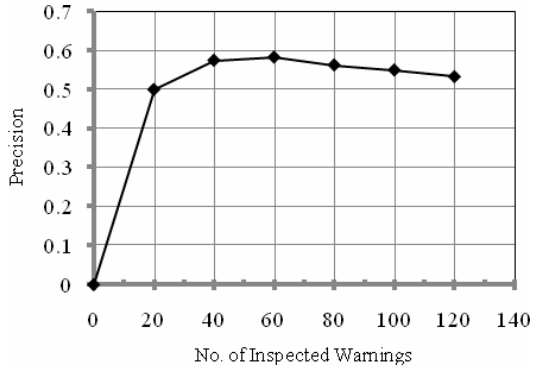

Fig. 6 Precision

E. Warning Density

It is measured as the number of warnings per size of Software Component being measured, (typically expressed in KLOC). KLOC is thousand lines of code and number of warnings represents the total number of potential warnings found while testing a tool against a particular software component.

This is similar to Defect density, one of the software quality metrics. It is useful in identifying defect prone components of software and in tracking the quality of software by measuring the percentage of defect reduction.

$$\text{Warning Density} = \frac{\text{Total Number of Warnings}}{\text{Size of Software Component (In KLOC)}} \quad (16)$$

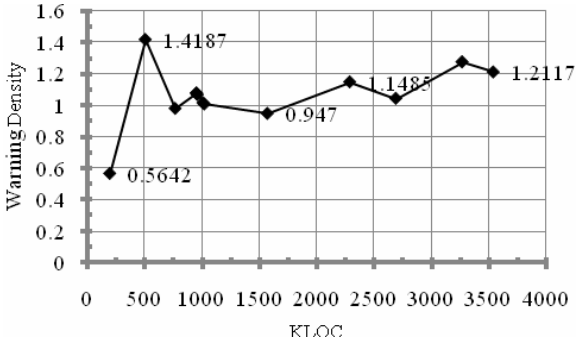

Fig. 7 Warning Density

Fig. 7 gives the plot of warning density against KLOC found in PmsMip.

IX. Conclusion and Future Work

BugCatcher.Net uses its own FSA-based parser, which helps in carrying out abstractions needed for analyzing program behavior statically in just one parse throughout the entire code. Further, the tool exhibits many useful features like capability of finding bugs by simply inspecting program code without its execution, prioritization of warnings reported, and ability to highlight the buggy source code without using source files and comparatively lower false positive rate; approximately about 45% or alternatively higher precision. Not only this, it is able to propose a solution to end user with minimal possible side effects. Moreover, tool can distinguish between definite and suspicious occurrence of a bug.

However, due to some assumptions and abstractions made while modeling the program behavior, in some cases there are few chances of incorrect modeling. Hence, some work should be done to perform more abstractions that are sophisticated. Further, the detectors have been implemented for some known bug-patterns. The implementation should be extended to detect remaining classes of bugs. In addition, a plug-in can be developed to allow end users to write their custom detectors. Moreover, the bug trend analyzing capability can be extended further to forecast the most likely bug-pattern.

References

[1] V. Channakeshava, S. Chavan, and V. Shanbhag, "Bug Detection through Static Analysis of MSIL", ACTA Press, Proceedings of Software Engineering and Applications, 2008.
[2] D. Hovemeyer and W. Pugh, Finding Bugs Is Easy, Companion of the 19th Ann. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '04), Oct. 2004.
[3] Introduction to IL Assembly Language - The Code Project - _NET, http://www.codeproject.com/KB/msil/ilassembly.aspx
[4] David Evans, John Guttag, James Horning, Yang Meng Tan, LCLint: a tool for using specifications to check code, Proceedings of the 2nd

ACM SIGSOFT symposium on Foundations of software engineering, p. 87-96, December 06-09, 1994, New Orleans, Louisiana, United States.
[5] Roger F. Crew, ASTLOG: A language for examining abstract syntax trees, USENIX Conference on Domain Specific Languages, Santa Barbara, 1997.
[6] PREfast, http://research.microsoft.com/specncheck/docs/pincus.ppt
[7] PMD, http://pmd.sourceforge.net/, 2003.
[8] JLint, http://artho.com/jlint
[9] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, H. Zheng, Bandera: Extracting Finite-state Models from Java Source Code, Proceedings of the 22nd International Conf. on Software Engineering, pages 439–448, Limerick Ireland, June 2000.
[10] FxCop: Microsoft MSDN library, http://www.got dotnet.com /team/fxcop/
[11] StyleCop: Microsoft MSDN library, http://code. msdn.Microsoft.com/sourceanalysis, http://blogs.msdn.com /sourceanalysis
[12] CodeITRight, http://submain.com/?nav=products.cir.
[13] Resharper, www.JetBrains.com/resharper/
[14] NStatic, http://wesnerm.blogs.com/net_undocumented/ 2006/02/nstatic_walkthr.html
[15] http://en.wikipedia.org/wiki/Context-free grammar.
[16] Giovanni Vigna, Reliable Software Group, University of California, Santa Barbara, Static disassembly and code analysis.
[17] J. E. Hopcroft, R. Motwani, J. D. Ullman, Introduction to Automata Theory, Langauges and Computation, second edition, Pearson Education.
[18] Chadd C. Williams, Jeffrey K. Hollingsworth, Automatic Mining of Source Code Repositories to Improve Bug Finding Techniques, IEEE transactions on software Engineering, vol. 31, no. 6, June 2005.
[19] Hongyu Zhang, On the Distribution of Software Faults, IEEE Transactions on Software Engineering, Vol. 34, No. 2, March 2008.
[20] Norman E. Fenton, Niclas Ohisson, Quantitative Analysis of Faults and Failures in a Complex Software System, IEEE Transactions on Software Engineering, Vol. 26, No. 8, August 2000.
[21] SourceForge, http://sourceforge.net/
[22] ItextSahrp, http://sourceforge.net/projects/itextsharp/
[23] Airplanewar, http://sourceforge.net/projects/airplane war/
[24] SharpDevelop, http://sourceforge.net/projects/sharp develop/
[25] EulerSharp, http://sourceforge.net/projects/eulersharp/
[26] VNCViewer, http://sourceforge.net/projects/vncviewer
[27] SWAT (Simple Web Automation Toolkit), http://source forge.net/projects/ulti-swat/
[28] Quartz.net(Quartz Enterprise Scheduler.NET), http://source forge.net/projects/quartznet/
[29] Nayatel, http://sourceforge.net/projects/nayatelids/
[30] NeuronDotNet, http://sourceforge.net/projects/neuron dotnet/
[31] TemplateEngine, http://sourceforge.net/projects/easy template/