A METRICS SUITE FOR VARIABLE CATEGORIZATION TO SUPPORT PROGRAM INVARIANTS

P.Radhika Raju[1] and Dr. A.Ananda Rao[2]

[1]Research Scholar, Dept. of CSE, JNTUA, Ananthapuramu, A.P, India
[2]Professor in CSE, JNTUA, Ananthapuramu, A.P, India

Abstract

Invariants are generally implicit. Explicitly stating program invariants, help programmers to identify program properties that must be preserved while modifying the code. Existing dynamic techniques detect invariants which includes both relevant and irrelevant/unused variables and thereby relevant and irrelevant invariants involved in the program. Due to the presence of irrelevant variables and irrelevant invariants, speed and efficiency of techniques are affected. Also, displaying properties about irrelevant variables and irrelevant invariants distracts the user from concentrating on properties of relevant variables. To overcome these deficiencies only relevant variables are considered by ignoring irrelevant variables. Further, relevant variables are categorized as design variables and non-design variables. For this purpose a metrics suite is proposed. These metrics are validated against Weyuker's principles and applied on RFV and JLex open source software. Similarly, relevant invariants are categorized as design invariants, non-design invariants and hybrid invariants. For this purpose a set of rules are proposed. This entire process enormously improves the speed and efficiency of dynamic invariant detection techniques.

Keywords

Program invariants, Design variables, Non-design variables, Metrics, Invariants.

1. INTRODUCTION

Program invariants play a central role in program verification. Generally invariants are implicit, stating the program invariants explicitly aids programmers to identify the properties that must be preserved while proving the correctness. Invariants can be detected using both static and dynamic approaches. Dynamic methods that complement static methods will be more useful and effective in analyzing the software artifacts [1,2]. However, these techniques produce the properties of relevant and irrelevant variables. That is irrelevant invariants are also inferred. Therefore, more analysis is to be done when more number of variables are present in the program which makes the job of invariant detection tools more complex and time taking. All these derived invariants are validated to prove the correctness of the program irrespective of whether they are relevant or not. However, validation of irrelevant invariants does not contribute to the correctness of the program. Further, it also distracts the concentration of the user and increases the validation time. Thus, it is necessary to consider only the relevant variables and ignore irrelevant variables. Hence, the effort and time required to analyze the irrelevant variables is reduced.

DOI : 10.5121/ijsea.2014.5505

The speed and efficiency of the tools can be further improved by categorizing relevant variables into two different types as design, non-design variables and relevant invariants into three different types. Set of rules are proposed to categorize invariants. For the purpose of variable categorization a metric suite is proposed and validated analytically. The reason for considering metrics for this purpose is that they play an important role in software development with respect to process improvement, measurement of design quality etc. [3,4]. Hence, metrics are crucial for software engineering activities. Proposed metrics are developed based on the design complexities of the variables at module level on dynamic traces which is a variation from existing work [3]. Reason for developing a metric suite after capturing the dynamic behavior of an application is to capture all implicit program invariants to verify the program correctness. So that there will not be any scope for missing any invariant. These metrics are useful at later phases in the life cycle of an application and are helpful to a big set of users who wish to prove the correctness of the program comparatively with less effort and time to that of the earlier methods.

Rest of the paper is presented in the following manner. Section 2 discusses about the related work. Section 3 explains the procedure to ignore unused variables. Section 4 discusses about the metrics suite and process of categorizing the variables. In section 5 the proposed metrics are validated analytically against Weyuker properties. Proposed concepts are applied over open source software in section 6. In section 7 invariants are categorized and section8 concludes the paper.

2. RELATED WORK

Numerous researchers have adopted and adapted the ideas of dynamic detection of likely invariants. There are two methods, static and dynamic, for detecting invariants. Among these two dynamic methods are very useful and effective in detecting the invariants. Several tools like DIDUCE [5], CARROT [6], Arnout's [7], Henkel and Diwan [8] and DAIKON are available for detecting invariants. Among the tools, Daikon is extensively used tool for detecting invariants. But still it has some drawbacks, which affects its speed and performance. The key reason for this is existence of more variables. So, an analyzer is proposed in this paper to classify variables as relevant and irrelevant variables and ignore unused variables.

In any engineering discipline measurement and metrics are crucial components, and same is the case with computer science. In software engineering, metrics are used in various contexts like estimation of software quality, process improvement etc. Focusing on software quality, lead to numerous measures of quality design. When a module evolves the effect of evolution on internal components within the module and its external relationships with other modules can be measured using metrics. Because, for any software component, metrics can assess its functionality, relationships, internal and external structure etc. So, metrics are developed for variable categorization. Two types of attribute categorization models are available [9,10]. They are semantic based categorization models and non-semantic based categorization models. Among these two, semantic based categorization models are better models [3,10]. However, in both the above said models, categorization is done by the designer conceptually or intuitively. Therefore, a quantitative model is required to perform categorization.

Key contributions of this paper are proposing a metrics suite and their usage to categorize relevant variables. Validation of proposed metrics against the set of measurement principles, Weyuker's principles is done. The concept is applied over two open source software. A set of rules are proposed for categorization of invariants into various types.

3. PROCEDURE TO IGNORE UNUSED VARIABLES

Program invariants play an important role in proving program correctness. These invariants can be identified effectively by analyzing various software artifacts. A program which takes more runtime produces more variables to be examined and more data to be investigated. Large variables such as arrays are more expensive to test than integers or booleans. So, it is important to concentrate on the key feature that influence the execution time of the program i.e., variable. A technique is essential to reduce the number of variables so that runtime of the program can be reduced.

Dynamic detection of program invariants concept when implemented with well-known tools like Daikon, determines the likely program invariants but still suffers from some drawbacks when applied on software. Consider a program with a class containing the data members and member functions. During execution when a method, having computational statements in which some of the data members are involved, is invoked the tool displays all the data members and their values of the class along with the variables that participated in the computations of the method. So, the variables that are not part of the method and its computations (irrelevant variables) are also displayed. This leads to two problems:

- **Speed**: Processing all the variables and displaying them takes more time and hence, reduces the speed.
- **Irrelevant Output**: When a method is invoked the user do not consider for the properties of the variables that are not part of the method. Outputting the properties of the irrelevant variables distracts the user from important properties about relevant variables. Furthermore, since the tool is considering more variables, some properties may be true purely by accident that are not useful.

For example consider a class *MyClassA* with 20 integer variables which are initialized with some values. It also contains a method *myMethod()* for the purpose of computing sum of two variables x1,x2 as follows.

```
classMyClassA{

int x1, x2,x3,x4,x5,x6,x7,x8,x9,x10,x11,x12,x13,x14,x15,x16,x17,x18,x19,x20;

publicMyClassA()
{
x1 = 100;
x2 = 200;
x3 = 3;
x4 = 4;
x5 = 5;
x6 = 6;
x7 = 7;
x8 = 8;
x9 = 9;
x10 = 10;
x11 = 11;
```

```
x12 = 12;
x13 = 13;
x14 = 14;
x15 = 15;
x16 = 16;
x17 = 17;
x18 = 18;
x19 = 19;
x20 = 20;
}
intmyMethod()
{
return(x1+x2);
}
}

classMyClassB{
public static void main(String args[]){
MyClassAobj=new MyClassA();
int sum = obj.myMethod();
System.out.println("sum:"+sum);
}
}
```

When this program is submitted to Daikon tool the following output is generated.

Output

```
Daikon version 4.6.4, released June 23, 2010; http://pag.csail.mit.edu/daikon.
Reading declaration files
[12:30:10 AM]:

Processing trace data; reading 1 dtrace file:

[12:30:10 AM]: Finished reading MyClassB.dtrace.gz


================================================================================
MyClassA:::OBJECT
this has only one value
this.x1 == 100
this.x2 == 200
this.x3 == 3
this.x4 == 4
this.x5 == 5
this.x6 == 6
this.x7 == 7
this.x8 == 8
this.x9 == 9
this.x10 == 10
```

```
this.x11 == 11
this.x12 == 12
this.x13 == 13
this.x14 == 14
this.x15 == 15
this.x16 == 16
this.x17 == 17
this.x18 == 18
this.x19 == 19
this.x20 == 20
================================================================================
MyClassA.MyClassA():::EXIT
================================================================================
MyClassA.myMethod():::ENTER
================================================================================
MyClassA.myMethod():::EXIT
this.x1 == orig(this.x1)
this.x2 == orig(this.x2)
this.x3 == orig(this.x3)
this.x4 == orig(this.x4)
this.x5 == orig(this.x5)
this.x6 == orig(this.x6)
this.x7 == orig(this.x7)
this.x8 == orig(this.x8)
this.x9 == orig(this.x9)
this.x10 == orig(this.x10)
this.x11 == orig(this.x11)
this.x12 == orig(this.x12)
this.x13 == orig(this.x13)
this.x14 == orig(this.x14)
this.x15 == orig(this.x15)
this.x16 == orig(this.x16)
this.x17 == orig(this.x17)
this.x18 == orig(this.x18)
this.x19 == orig(this.x19)
this.x20 == orig(this.x20)
return == 300
================================================================================
MyClassB.main(java.lang.String[]):::ENTER
args has only one value
args.getClass() == java.lang.String[].class
args[] == []
args[].toString == []
================================================================================
MyClassB.main(java.lang.String[]):::EXIT
args[] == orig(args[])
```

```
args[] == []
args[].toString == []
Exiting Daikon.
```

From the above output it is observed that whenever *myMethod( )* executes, the values of all 20 variables are displayed. But only 2 of the variables are relevant to this method. This leads to following problems:

> Here, 400 pairs of variables are compared instead of one pair for which tool takes more time to process and output all these variables. Hence, obviously reduce speed of the tool.
> The user may not consider the values and properties of other variables i.e., x3…..x20 whenever *myMethod( )* is invoked. Outputting them distracts the user from important properties about x1 and x2. Furthermore, since the tool is considering 400 pairs of variables, some properties may be true purely by accident that is not useful.

Above discussed two problems have severe impact on the performances of dynamic invariant detection the tools. To overcome these problems and to improve the performances of the tools, this paper presents an efficient technique to ignore such unused variables.

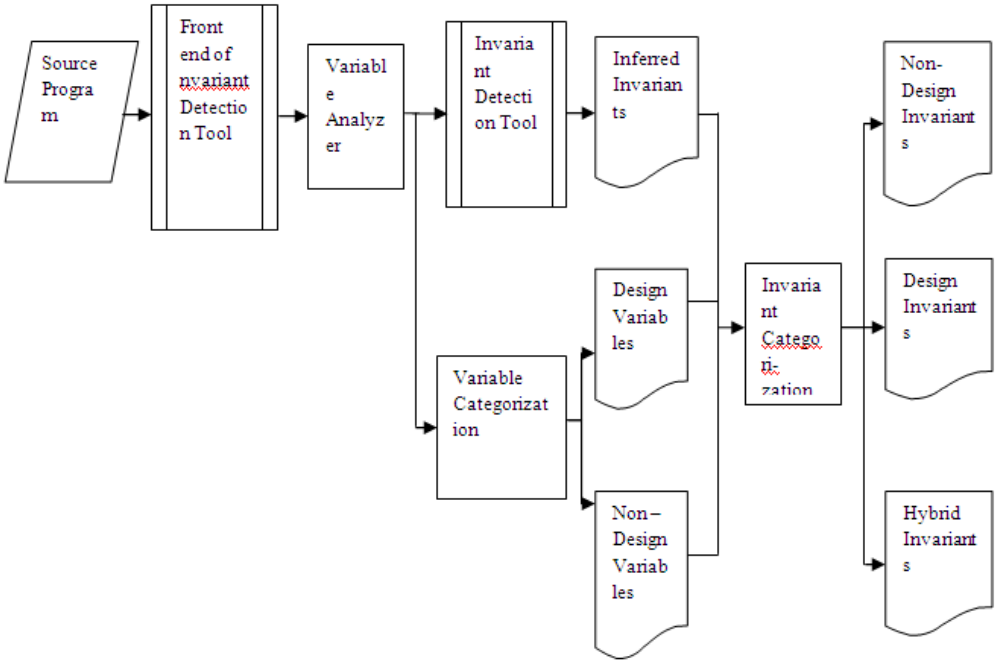

Figure1. Architecture to ignore unused variables and categorize variables, invariants

In figure 1, initially a source program is subjected to the front end of the invariant detection tool. A trace file is generated from instrumented source program, which consists of all variables both relevant and irrelevant. This trace file is given to the proposed **Variable Analyzer**, where variables are identified or recognized as relevant variables and irrelevant variables. The outcome of the variable analyzer is given to the invariant detection tool and it outputs the relevant invariants. Using variable analyzer the unused irrelevant variables are ignored which reduces the burden that used to be born by the tool in deducing all the invariants. Analyzer is discussed here

under. The variable categorization, the metrics suite, their validation and invariant categorization are discussed in the following sections.
The analyzer determines, for a method *m*, all the variables that may be read or written by *m* or by any method that *m* may call. These are the relevant variables. The trace file of the given source program from the frontend (that consists of all the variables) is the input to the analyzer. It separates the relevant and irrelevant variables using tokens. Every time a method is invoked, the analyzer identifies the relevant variables and irrelevant variables of this method and ignores the irrelevant variables. Hence, only the relevant variables can be considered by avoiding the irrelevant variables of the method. This analyzer reduces time spent on considering all the variables both relevant and irrelevant. Now, the properties of the unused variables are also suppressed which used to distract the user from concentrating on important properties of relevant variables. Hence, improves speed and performance of the tool.

Following is the output by using the proposed technique for ignoring unused variables for the above discussed example.

Output by using proposed technique

Daikon version 4.6.4, released June 23, 2010; http://pag.csail.mit.edu/daikon.
Reading declaration files
[12:26:32 AM]:

Processing trace data; reading 1 dtrace file:

[12:26:32 AM]: Finished reading MyClassB.dtrace.gz

```
================================================================================
MyClassA:::OBJECT
this has only one value
this.x1 == 100
this.x2 == 200
================================================================================
MyClassA.MyClassA():::EXIT
================================================================================
MyClassA.myMethod():::ENTER
================================================================================
MyClassA.myMethod():::EXIT
this.x1 == orig(this.x1)
this.x2 == orig(this.x2)
return == 300
================================================================================
MyClassB.main(java.lang.String[]):::ENTER
args has only one value
args.getClass() == java.lang.String[].class
args[] == []
```

```
args[].toString == []
==========================================================================
======
MyClassB.main(java.lang.String[]):::EXIT
args[] == orig(args[])
args[] == []
args[].toString == []
Exiting Daikon.
```

In the above output by using the proposed technique, values of only relevant variables are displayed. All the invariants inferred are relevant which does not consists of any irrelevant variable and irrelevant invariant. Therefore, the proposed technique helps in focusing only on the relevant and needful invariants. Otherwise, used to distract the concentration on the irrelevant invariants as observed in the earlier output without using the proposed technique. This is the case with the small program where only 20 variables are there and deals with a simple computation. But generally software will be huge in size. In such cases usage of the proposed technique is very important which drastically reduces the effort and improves the speed of the tool.

4. METRICS SUITE

Set of metrics are developed for variable categorization, based on design complexities of variables. These metrics are developed based on the interactions of the variables with the variables, methods within the module and outside the module, type (size) of the variable**.** A variable is considered as more complex (design variable) if it is larger in size, and has more interactions within and outside a module and need to be given utmost priority. Its properties must be carefully checked and are to be maintained as they are. Because, whenever there is a change in design variable of any artifact then there is a need to change the design of an artifact and same is to be propagated to related artifacts in the system. This propagation results in either design change or equivalent change in related artifacts based on the amount of change that has happened to that artifact [3,10].

If the interactions of a variable are less, then it is less complex, also referred as non-design variable. When there is a change in non-design attribute of an artifact this leads to its equivalent design change. Whenever this equivalent change is propagated to related artifacts in the system it will result in either equivalent change or no change based on the amount of change that has happened to that artifact. That means equivalents can be replaced with one another without affecting the integrity of the software systems [3,10]. So, they may be given the priority next to that of design variables. In this regard a set of metrics are proposed to categorize the variables.

4.1. Metrics for Variable Categorization

Proposed metrics are developed based on the design complexities of the variables at module level on dynamic traces.

Table 1. Size or complexity of variables types

Type	Size
Boolean	0
Character or Integer	1
Real, Float	2
Array	3
Pointer	5
Record, Struct, Object	6
File	10

The size value for different variables is an important factor in developing the metrics. So, they are taken from the table 1 that are suggested in [11,12,13]. These values are given based on the design complexities of various variables by the experienced designers. In this paper variable and variable instance, method and method instance are used interchangeably while discussing the metrics. The proposed metrics to categorize the variables are discussed below.

4.1.1. Interaction Metric of Variable with Methods Inside the Module (IM_i)

This metric quantifies the interactions inside the module. It is defined as the ratio of the number of method instances inside the module that reference the variable instance under consideration to the total number of possible method instances in a module.

$$IM_i = NM_{irv}/TM$$

where NM_{irv} is number of methods inside the module that reference the variable and TM is the total number of methods in the module.

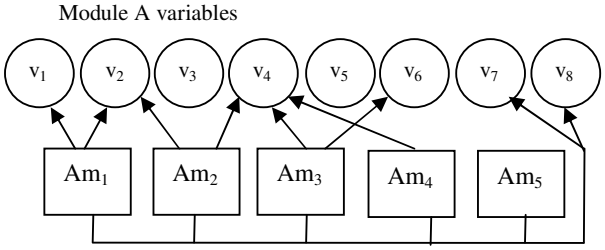

Figure 2. Module A methods interaction with its variables

If all the methods in a module reference the variable, then IM_i is maximum i.e., 1 and if none of the methods in a module reference the variable the value of IM_i is minimum i.e., 0. Consider a module A shown in figure 2 with 8 variables v_1, v_2 , v_3, v_4, v_5, v_6, v_7, v_8 and 5 methods Am_1, Am_2, Am_3, Am_4, Am_5. Here v_1, v_2 , v_3 are files, v_4, v_5 are float type variables, v_6, v_7, v_8 are integers. The figure illustrates the computation of the metric IM_i. In the figure, variables v_7 and v_8 are referenced by all the five methods. Therefore, the value of IM_i for the variables v_7 and v_8 is 5/5 =

1. The variables v_1 and v_6 are referenced by only one method and the value of IM_i is 1/5 = 0.2. Similarly, the values of IM_i for other variables can be computed.

4.1.2. Interaction Metric of Variable with Methods Outside the Module (IM_o)

This metric quantifies the interactions outside the module. It is defined as the ratio of the number of method instances outside the module that reference the instance of a particular variable under consideration to the total number of possible method instances in the referenced module.

$$IM_o = \sum_{i=1}^{N-1} \sum_{j=1}^{M} Vij \; / \sum_{i=1}^{N-1} \sum_{j=1}^{M} Mij$$

where V_{ij} = 1 if the j^{th} method of the i^{th} module references the variable under consideration and V_{ij}= 0 otherwise, M_{ij} =1 for every j^{th} method of the i^{th} module, N represents the number of referenced modules including the one that is under consideration and M represents the number of methods for each module.

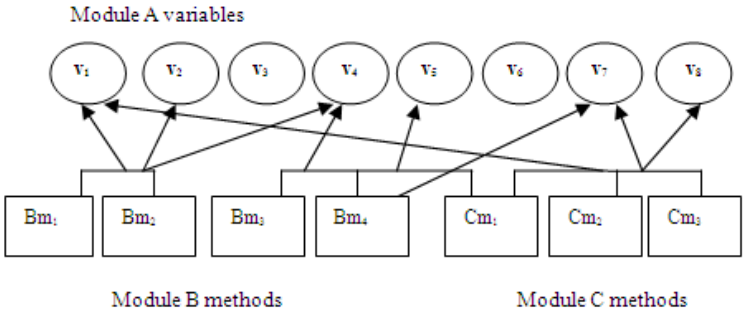

Figure 3. Module A variables interacting with the methods of Module B, Module C

The values 1 and 0 for V_{ij} and M_{ij} are chosen for convenience and normalization. For explaining the computation of metric IM_o the interactions that are shown in figure 3 are considered. In the figure module B and module C are outside related modules for module A. Module B has 4 methods Bm_1, Bm_2, Bm_3, Bm_4 and module C has 3 methods Cm_1, Cm_2, Cm_3. Therefore, the total number of methods outside module A is 7. The variable v_1 is referenced by 5 methods and the IM_o metric value for variable v_1 is 5/7 = 0.71. In a similar way, IM_o values for other variables can be computed.

4.1.3. Interaction Metric of Variable with Other Variables Inside the Module (IV_i)

This metric quantifies the interactions of variable with other variables inside the module. It is defined as the ratio of the number of variable instances inside the module that reference the variable under consideration to the total number of possible variable instances in a module.

$$IV_i = NV_{im}/TV$$

where NV_{im} is number of variable instances inside the module that reference the variable under consideration and TV is the total number of variable instances in the module.

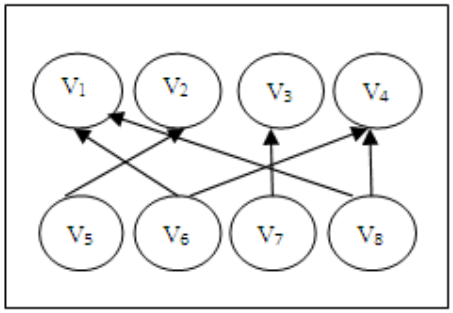

Figure 4. Variable interactions in module A

If all the variable instances in a module reference the variable, then IV_i is maximum i.e., 1 and if none of the variables in a module reference the variable the value of IV_i is minimum i.e., 0. Consider a module A shown in figure 4 with 8 variables v_1, v_2 , v_3, v_4, v_5, v_6, v_7, v_8.The figure illustrates the computation of the metric IV_i. In the figure, variables v_1 is accessed by v_6 and v_8. Therefore, the value of IV_i for the variable v_1 is 2/8 = 0.25. The variable v_2 is accessed by only one variable v_5 and the value of IV_i is 1/8 = 0.125. Similarly, the values of IV_i for other variables can be computed.

4.1.4. Interaction Metric of Variable with Other Variables Outside the Module (IV_o)

This metric quantifies the interactions of variable with other variables outside the module. It is defined as the ratio of the number of variable instances outside the module that reference the instance of a particular variable under consideration to the total number of possible variable instances that exists in the referenced module.

$$IV_o = \sum_{i=1}^{N-1} \sum_{j=1}^{M} Vij \; / \sum_{i=1}^{N-1} \sum_{j=1}^{M} Wij$$

where V_{ij} = 1 if the j^{th} variable of the i^{th} module references the variable under consideration and V_{ij}= 0 otherwise, W_{ij} =1 for every j^{th} variable of the i^{th} module, N represents the number of referenced modules including the one that is under consideration and M represents the number of methods for each module.

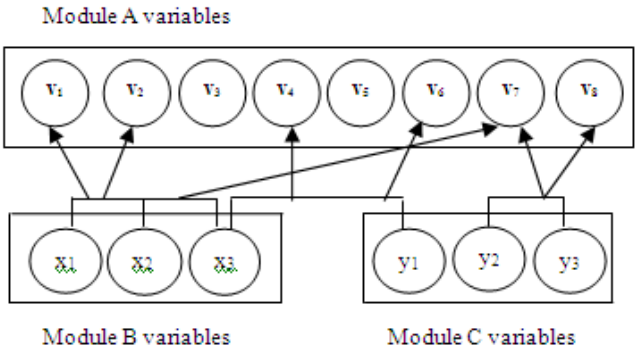

Figure 5. Variables of module A interacting with outside variables variablesmodule

In figure 5, module A has 8 variables, module B and module C has 3 variables each. All the 3 variables x1, x2, x3 of module B are accessing v1 and v2. So, the value of IV_o for the variables v_1 and v_2 is 3/6 = 0.5. The variable v_4 is accessed by x_3 of module B and y_1 of module C so, the value of IV_o for variable v_4 is 2/6 = 0.33. Similarly, the values of IV_o for other variables can be computed.

4.1.5. Metric for Variable Type (*MVT*)

It is defined as the ratio of the size of the type of a variable to the maximum size of the variable. The maximum size is the size of the file type variable (see table 1and taken from [3]). The size of a variable or parameter is a specified constant, specifying the complexity of the variable type. By using this metric it can be measured how complex a variable is. Formally it is defined as

$$MVT = S/M_s$$

where S is the size of the variable type and M_s is the maximum size of the variable type. For example, if the type of the variable v is an integer, then the type metric *MVT* for variable v is 1/10 = 0.1. Similarly, the type metric *MVT* value for variable v_1 in the above example module A is 10/10 = 1.

Table 2. Calculated values for various metrics of variables

	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
IM_i	0.2	0.4	0	0.6	0	0.2	1	1
IM_0	0.71	0.28	0	0.71	0.42	0	0.57	0.42
IV_i	0.25	0.125	0.125	0.25	0	0	0	0
IV_o	0.5	0.5	0	0.33	0	0.66	0.83	0.33
MVT	1	1	1	0.2	0.2	0.1	0.1	0.1

All the above mentioned metrics are calculated for examples shown in figures 2, 3, 4 and 5 respectively and the values are tabulated in table 2. Variables are categorized as design and non-design variables. If the criterion is such that fifty percent of the metrics have a value that is equal to or greater to the threshold value (0.5), then they are categorized as design variables. From table 2, attribute v_1,v_7 are categorized as design variables and others are categorized as non-design variables.

5. VALIDATION OF PROPOSED METRICS

E.J.Weyuker has developed a formal set of desiderata for software metrics [14]. These properties are used to evaluate numerous existing software metrics. The notion of monotonicity, interaction, non-coarseness, non-uniqueness and permutation can be found in these desiderata. Generally formal evaluation is done by evaluating metrics against the Weyuker's principles. All the proposed metrics are validated as shown below and tabulated in table 3 that shows the metrics that satisfy the corresponding property.

Property 1: Non-Coarseness

A metric μ when applied on artifacts A and B it states that $\mu(A) \neq \mu(B)$. That is a metric should not produce the same value for all artifacts. If it does so it loses its value as a measurement.

Consider *MVT* metric. This metric satisfies the above property because it considers sizes of various types of variables. When two variables, a file v_1 and integer v_2 are considered then $\mu(v_1)=10/10=1$ and $\mu(v_2)=1/10=0.1$. Hence $\mu(v_1) \neq \mu(v_2)$. Here, μ refers to the corresponding metric in discussion.

Consider IM_i metric. This metric satisfies the above property as it measures the number of method's instance accessing the variable within the module. When two variables v_1, v_2 are considered and v_1 is being accessed by 2 methods out of 5 methods , and v_2 is being accessed by 3 methods out of 5 methods. Then $\mu(v_1)=2/5=0.4$ and $\mu(v_2)=3/5=0.6$. Hence $\mu(v_1) \neq \mu(v_2)$.

Property 2: Granualarity

It states that there will be a finite number of cases for which the metric value will be same. As the universe deals with finite set of applications, a finite set of cases with the same metric value will be found. This property will be satisfied by any metric which is measured at variable level. In this paper all the metrics proposed are measured at variable level and hence all of them satisfy this property.

Property3: Non-Uniqueness (Notion of equivalence)

If A and B are distinct artifacts then the property states that $\mu(A)=\mu(B)$. That is two artifacts may have same value for the metric which implies that both artifacts are equally complex.
MVT metric satisfies this property. When two variables float v_1, float v_2 are considered then $\mu(v_1)=2/10=0.2$ and $\mu(v_2)=2/10=0.2$. Hence $\mu(v_1) = \mu(v_2)$.

Consider IM_i metric. This metric satisfies the above property. When two variables v_1, v_2 are considered and both v_1, v_2 are being accessed by 2 methods out of 5 methods then $\mu(v_1)=2/5=0.4$ and $\mu(v_2)=2/5=0.4$. Hence $\mu(v_1) = \mu(v_2)$.

Property 4: Design Details are Important

This property states that, in determining the metric for an artifact its design details also matters. When designs of two artifacts A and B are considered which are same in functionality i.e., A=B does not imply that $\mu(A)=\mu(B)$. That means specifics of the artifact design must influence the metric value.

For instance *MVT* satisfies this property. Consider designs of two methods A and B having same functionality (A=B). If v_1 is an integer variable in the design of module A and v_2 is a float variable in the design of module B then $\mu(v_1)=1/10=0.1$and $\mu(v_2)=2/10=0.2$. Hence $\mu(v_1) \neq \mu(v_2)$.

Consider IM_i metric. This metric satisfies the above property. Let in design of module A variable v_1, is accessed by 3 methods and in design of module B by 2 methods out of 5 methods, then $\mu(v_1$in A)=3/5=0.6 and $\mu(v_1$ in B)=2/5=0.4. Hence $\mu(v_1$ in A) $\neq \mu(v_2$ in B).

Property 5: Monotonicity

It states that a component of a program is always simpler than the whole program. For all A and B, $\mu(A) <= \mu(A+B)$ and $\mu(B) <= \mu(A+B)$ should hold. This implies that the metric for the combination of two artifacts can never be less than the metric for either of the artifacts.

As the type of a variable does not have any component, this property is not applicable for the metrics *MVT* and IM_i.

Property 6: Non-Equivalence of Interaction

It states that two artifacts with the same complexity need not have the same complexity after being concatenated with a third artifact. When A, B and C are three artifacts and μ(A)=μ(B) does not imply that μ(A+C)=μ(B+C). This suggests that interaction between A and C may differ from that of B and C.

This property is not applicable for *MVT*. The metric depends only on the type of the variables and concatenation has no effect on the metric value. IM_i satisfies this property. Because, methods can be concatenated and complexity of the concatenated methods will differ with that of the earlier one.

Property 7: Permutation

It states that permutation of elements within the artifact being measured may change the metric value. When there is an artifact A and by permuting the order of the statements of A artifact B is formed then this property requires that $\mu(A) \neq \mu(B)$.

This property is not applicable for *MVT*. As the metrics are developed at variable level for IM_i this property is not satisfied.

Property 8: Renaming property

When name of the measured artifact changes, the metric should not change. That is, if artifact B is obtained by renaming artifact A then μ(A)=μ(B).

MVT and IM_i metrics satisfy this property as size of the variable and method interactions will not be changed when it is renamed.

Property 9: Interaction Increases Complexity

It states that when two artifacts are combined, interaction between them can increase the metric value. When two artifacts A and B are considered μ(A)+μ(B)<μ(A+B).

This property is not applicable for *MVT*. The metric depends only on the type of the variables so there is no interaction hence, no effect on the metric value. IM_i metric will not satisfy this property because though complexity increases by combining the methods, the metric is defined at variable level and its complexity will not be affected.

Table 3. Metrics evaluation table for variables

Metric	P1	P2	P3	P4	P5	P6	P7	P8	P9
MVT	Y	Y	Y	Y	NA	NA	NA	Y	NA
IM_i	Y	Y	Y	Y	NA	Y	N	Y	N
IM_O	Y	Y	Y	Y	NA	Y	N	Y	N
IV_i	Y	Y	Y	Y	NA	NA	N	Y	NA
IV_o	Y	Y	Y	Y	NA	NA	N	Y	NA

Similarly, other metrics are validated and consolidated evaluation of all proposed metrics is tabulated in table 3. In the table, Y represents satisfied, N represents not satisfied and NA represents not applicable. It is clear from the table that the proposed metrics satisfy all the applicable Weyuker properties. Property 9 is not applicable or not satisfied by any of the proposed metrics. Hence, proposed metrics are effective and can be utilized for the purpose of categorization.

6. CASE STUDIES

Empirical validation for the proposed work is performed in this section. For this purpose two open source software RFV 2.1 and JLex 1.2 versions are considered as case studies.

RFV validation is as follows.

- Source code of this software is given as input to the front end of Daikon tool and trace file is generated.
- Using Rational Rose software, source code is reverse engineered to obtain class diagrams.
- From these trace file and class diagrams, information regarding various interactions among the classes, methods, variables and type of variables are obtained.
- This information is used to compute the values for proposed metrics.
- Based on these values variables are categorized as design and non-design.

For RFV, using above procedure and values obtained for various metrics, all the variables of RFV 2.1 are categorized into design and non-design. For the purpose of space constraint and explanation RFV-Text-Line class is considered. This class contains five variables and for all these variables, metrics are calculated and values are tabulated in table 4.

Table 4. Value of metrics for RFV 2.1

Metrics	Label	Font	Font-metrics	Color	Height-offset
IM_i	0.5	0.5	0.5	0.5	1
IM_O	0	0	0	0	0
IV_i	0.2	0.2	0.6	0.2	0.4
IV_o	0	0	0	0.3	0.3
MVT	0.3	0.6	0.6	0.6	0.1

Variables are categorized as design and non-design variables. If the criterion is such that fifty percent of the metrics have a value that is equal to or greater to the threshold value (0.5), then they are categorized as design variables. From table 4, attribute **Font-metrics** is categorized as design variable and others are categorized as non-design variables.

In case of JLex, metrics values for all the variables are calculated in the following manner.

- AspectJ compiler is used to parse the source code and parse tree is constructed.
- Facts-base generator is used to traverse the parse tree and Prolog facts are generated.
- After this, Prolog rules are written and fired against the generated facts.

In above procedure, a separate Prolog rule is written for each metric and values are obtained. In this case study Jlex.CSpec, JLex.Error and JLex.CLexGen classes are considered in which only one variable is added in this version. In JLex.CSpec class variable *m-public* is added and after

verifying the values obtained for the metrics this variable is categorized as non-design variable. Same is the case with other two classes. The reason for considering these classes is as follows.

To the earlier version of JLex these changes has been made. It is obvious to prove the correctness of JLex 1.2. In this process, how the proposed technique is useful to reduce the effort, time and number of invariants will be observed. This part of work is considered as future extension to the proposed research.

7. CATEGORIZATION OF INVARIANTS

Invariants present valuable information about a program operation and its data structures [1]. To help in further testing a program the detected invariants are incorporated as assert statements. This also helps in ensuring that these detected invariants are not violated later as the program evolves. A nearly-true invariant should be considered as a special case and needs to be brought to the programmer's notice. It even may indicate a bug. Dynamic invariants form a continuum that helps to assess impact of a change on a software system.

In the previous section variables were categorized into design and non-design variables. Based on this categorization relevant invariants can also be further categorized as *design invariants, non-design invariants* and *hybrid invariants.* The following are the rules to categorize the invariants.

- The invariants in which only the design variables take part are known as *design invariants*.
- The invariants that consist of only non-design variables are known as *non-design invariants*.
- And finally, the invariants that have both design and non-design variables are known as *hybrid invariants*.

As part of program evolution change may occur either to design or non-design variables. If the change has occurred to a non-design variable then it would be sufficient to validate only those invariants in which this non-design variable is participating to prove the correctness of the program. Any change in non-design invariant will not affect any design variable [11]. Hence, it is not required to validate the design invariants. This reduces the time spent on invariants other than non-design invariants. Before addressing the other two cases it is important to know about the change propagation.

Identification of Change Propagation

Generally, variables will have interactions with one another within a module. When there is a change in a design variable, it is required to know whether this change is restricted to itself or has propagated to other related variables. This is decided based on the threshold value over the dependencies of variables either directly or indirectly. The threshold value is specified by the expert designer. If the dependency is more than the threshold value, then the change propagates to the other related variables. Otherwise, the change will not propagate.

If a change has occurred to a design variable, then all the design invariants in which it is taking part are to be validated. In addition, the non-design invariants and other design invariants for which this change has propagated are also to be validated.

With respect to hybrid invariants, the following four cases are to be considered.

- If a change has occurred to a non-design variable, which is also present in other non-design invariants, then this invariant will be validated as discussed in the case of exclusive non-design invariants.
- If a change has occurred to a design variable, which is also present in other design invariants, then this invariant will be validated as discussed in the case of exclusive design invariants.
- If a change has occurred to a non-design variable, which is only participating in this hybrid invariant then this invariant should only be validated with respect to this non-design variable. Here, other parts of the hybrid invariant, which include either design or non-design variables need not be validated.
- If a change has occurred to a design variable, which is only participating in this hybrid invariant, then this invariant should be validated with respect to this design variable. In addition, other parts of the hybrid invariant, which include either design or non-design variables, also need to be validated because of change propagation. On similar lines, change propagation from this variable to other variables which are present in other invariants also to be identified. If any invariants are affected by this change, then they also have to be validated.

The proposed technique reduces the number of invariants to be considered for validation as a part of proving the correctness of a program.

Assume that there are ten variables in a method, a1, …, a10. Among which, six are non-design variables and four are design variables. When a change has occurred to one of the six non-design variables then it is sufficient to consider only the invariants in which this non-design variable takes part. Earlier all the invariants were used to consider for validating the correctness of a program. So, by using the proposed technique the number of invariants that are to be considered are reduced, which influence the time and effort required. As well concentrating on unimportant properties of other invariants is avoided. Similarly, change to a design variable can also be addressed.

8. CONCLUSIONS

By ignoring irrelevant variables and irrelevant invariants time required to spend by the tools on irrelevant variables and their properties is reduced. This research has developed and implemented a metric suite for categorizing variables in to two different types. This further improves the speed and efficiency of the tools by reducing the effort. These metrics are analytically validated against Weyuker properties. Proposed metrics suite is applied over two case studies and the results are presented.

As a part of future directions two extensions are considered for the above work. First one is, how the proposed technique is useful to reduce the effort, time and number of invariants is to be observed. For this purpose, a wrapper is to be developed and required to integrate with Daikon tool. Second one is, change is inevitable for any software system. Therefore, whenever a change occurs it has to be propagated to related artifacts. Hence, how these invariants are useful in change propagation can be explored.

References

[1] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. IEEE TSE, 27(2):1–25, Feb. 2001.

[2] M. Vaziri and G. Holzmann.Automatic detection of invariants in Spin. In SPIN 1998, Nov. 1998.

[3] AnandaRao A. Automation of Software Version Management Using Metrics, Ph.D thesis, Dept. of Computer Science and Engineering, IIT Madras, Chennai, 2006.

[4] Chidamber S.R. and C.F.Kemerer, A Metrics Suite for Object Oriented Design, IEEE Trans. Softw.Eng., vol. 20, no. 6, pp.476-493, 1994.

[5] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In ICSE '02: Proceedings of the 24th International Conference on Software Engineering, pages 291–301, New York, NY, USA, 2002. ACM Press.

[6] B. Pytlik, M. Renieris, S. Krishnamurthi, and S. P. Reiss. Automated fault localization using potential invariants. In AADEBUG'2003, Fifth International Workshop on Automated and Algorithmic Debugging, pages 273–276, Ghent, Belgium, September 8–10, 2003.

[7] KarineArnout, and Raphaël Simon. "The .NET Contract Wizard: Adding Design by Contract to languages other than Eiffel". TOOLS 39 (39th International Conference and Exhibition on Technology of Object Oriented Languages and Systems). IEEE Computer Society, July 2001, p 14-23.

[8] Henkel and A. Diwan. Discovering algebraic specifications from java classes. In L.Cardelli, editor, ECOOP 2003 Object Oriented Programming, 17th European Conference, Darmstadt, July 2003. Springer.

[9] Ramakrishnan R. and D.J.RamModeling Design Versions , VL DB'96, Proceedings of 22th International conference on very large databases ,September 3-6, 1996, Mumbai (Bombay), India (vijayaraman T.M, A.P. Buchmann, C. Mohan and N. L. Sarda, eds.), pp. 556-566, Morgan Kauf mann,1996.

[10] Srinath S. , R. Ramakrishnan, and D.J. Ram, A Generic model for Sematics–based versioning in Projects, IEEE Transactions on Systems , Man, and Cybernetic , Part A, vol.30,no. 2, pp. 108-123, 2000.

[11] Bandi R.K. , V.K. Vaishnavi, and D.E. Turk, Predicting Maintenance Performance Using Object-Oriented Design Complexity Metrics, IEEE Trans. Softw. Eng., vol. 29, no.1,pp. 77-87,2003.

[12] Abbott D, A Design Complexity Metric for Object-Oriented Development, Master's thesis , Dept. of computer science, Clemson University,1993.

[13] Chen J-Y and Lu J-F, A New Metric for Object-Oriented Design , Information and Software Technology, pp. 232-240, April 1993.

[14] Weyuker E.J., Evaluating Software Complexity Measures, IEEE Trans. Softw. Eng., vol.14, no. 9,pp. 1357-1365,1988.

Authors

P.Radhika Raju received B.Sc. (Comp. Sci) from Sri Krishnadevaraya University, Ananthapuramu; MCA degree from Indira Gandhi National Open University (IGNOU), India and M.Tech degree in Computer Science from Jawaharlal Nehru Technological University Anantapur, Ananthapuramu, A.P, India. She is now pursuing her Ph.D degree from JNT University Anantapur. She authored a text book and has publications in National and International Journals/Conferences. Her research interest include primarily Software Engineering. She is now working as a Lecturer in department of Computer Science and Engineering, JNTUA College of Engineering, Ananthapuramu, A.P.

Dr. A.Ananda Rao received B.Tech degree in Computer Science and Engineering from University of Hyderabad, A.P, India and M.Tech degree in Artificial Intelligence and Robotics from University of Hyderabad, A.P, India. He received Ph.D degree from Indian Institute of Technology Madras, Chennai, India. He is Professor in Computer Science & Engineering and currently discharging his duties as Director Industrial Relations & Placements and SCDE, JNT University Anantapur, Ananthapuramu, A.P, India. Dr. Rao has more than 100 publications in various National and International Journals/Conferences and authored three text books. He received one Best Research Paper award, one Best Paper award for his papers. He received many awards such as Best Teacher Award form Govt. of Andhra Pradesh in 2014, Best Computer Science Engineering Teacher award from ISTE, AP Chapter in 2013. His main research interest includes Software Engineering and Databases.