International Journal of Computing and Digital Systems
@ 2014: Scientific Publishing Center, University of Bahrain

Localizing Program Bugs Based on Program Invariant

Iman Jamnejad[1], Ali Heidarzadegan[1], Hamid Parvin[1], Hamid Alinejad-Rokny[2]

[1]Department of Computer Engineering, Beyza Branch, Islamic Azad University, Beyza, Iran
[2]School of Computer Science and Engineering, The University of New South Wales, Sydney, Australia

E-mail address: jamnejad@beyzaiau.ac.ir, parvin@beyzaiau.ac.ir, heidarzaden@beyzaiau.ac.ir, H.Alinejad@ieee.org

Received 18 Feb. 2013, Revised 7 Dec. 2013, Accepted 15 Jan. 2014, Published 1 May. 2014

Abstract: Software error detection is about finding and prediction software logical error before deliver it to customers by some of automatic approaches. Some of logical errors are hidden in source code and cannot be found simply and no piece of software is free of logical errors. As software is written by humans, errors can always occur. So, finding and prediction of errors is one of most important issues in software development cycle.
Program logical error localization and program testing are two of the most important sections in software engineering. Programmers or companies that produce programs will lose their credit and profit effectively if one of their programs delivered to a customer has any drawback. Nowadays there are many methods to test a program. Invariant are program valuable properties and relations which are true in all executions. This paper suggests a framework to localize the program logical errors by extraction of knowledge from invariants using a clustering technique.

Keywords: Invariant Rules, Dynamic Detection, Variable Relations, Software engineering, Verification

1. Introduction

Software error detection is about finding and prediction software logical error before delivering it to customers by some of automatic approaches. Some of logical errors are hidden in source code and cannot be found simply and no piece of software is free of logical errors. As software is written by humans, errors can always occur. So, finding and prediction of errors is one of most important issues in software development cycle.

Program error localization is one of the most important fields in program production and software engineering. Programmers or companies that produce programs will lose their credit and profit effectively if one of their programs delivered to a customer has any drawback. While there are many methods to test a program, it is the lack of an appropriate method for localizing a program logical errors using extraction of knowledge from invariants [3]. This paper will offer a framework to localize the program logical errors before its release. This method is based on extracting the knowledge from program invariants called Logical Error localizator Based on Program Invariants (LELBPI).

Invariant are program valuable properties and relations which are true in all executions. For example in a sort function such as bubble sort, while leaving the function, all the elements of the array are sorted so invariant (array a sorted >=) is reported. Such properties might be used in formal specification or assert statement. Invariant is introduced by Robert and Floyd [13]. They are a set of rules that govern among the values of variables in the programs in such a way that they remain unchanged in the light of different values of the input variables in the consecutive runnings of a program. There are three types of invariant generally: pre-condition, loop-invariant and post-condition. However post-conditions are considered as a kind of invariants. In this paper, where it is addressed invariant, it only implies to the invariant of post-condition type.

Since invariants repeat the properties and relations of program variables, invariants can express the behavior of a program. Therefore after an updating to the code, invariants can determine which properties of the code remain unchanged and which properties are changed. Invariants are kind of documentation and specification [6]. Since specification and documentation are essentials in software engineering, Invariants can be used in all processes of software engineering from design to maintenance [9]. There are two different approaches to detent invariants, static and dynamic.

In the static approach the syntactic structure and runtime behavior of program are checked without actually running of code [1]. Data-flow is a kind of invariant which is traditionally used in compilers for optimizing of codes. Data-flow analysis can determine the properties of program points. Abstract interpretation is a theoretical framework for static

E-mail: jamnejad@zhariau.ac.ir

analysis [7] and [11]. The most precise imaginable abstract interpretation is called the static semantics or accumulating semantics.

In contrast to static analysis, dynamic invariant detection tools elicit invariants by actually executing of the code with different test suits and inputs. Properties and relations are extracted through the execution of the code. Dynamic invariant detection emerged to software engineering realm during recent ten years by Daikon [9]. By using different test suits in different executions, in each program point, variable properties and relationships are extracted. These Program points are usually the points of entry and exits of program functions. Extracted properties and relations are invariants. These invariants are not certainly true but indeed they are true in all executions in test suits. One of the most important advantages of this approach is that what invariant reports not only shows the properties and relations of variables via execution but also utters the inputs properties and relations. This advantage of dynamic invariants doubles its usage.

Invariants have significant impact on software testing. Also, invariants are useful in comparing two programs by programmers and can help them check their validity. For instance, when a person writes a program for sorting a series of data, s/he can conclude that his program are correct or has some bugs by comparing his program invariants against the invariants of a famous reliable sort program; such as Merge Sort. Here, the presupposition is that in two sets (a) invariants detected in the program and (b) invariants detected in the merge sort program, must be almost the same. Additionally, invariants are useful in documentation and introduction of a program attributes; i.e. in cases where there are no documents and explanations on a specific program and a person wants to recognize its attributes for correcting or expanding program, invariants will be very helpful to attain this goal, especially if the program is big and has huge and unstructured code.

Specification by invariant is one of the best methods to show the behavior of a program. Many programmers use invariants to test programs and identify their bugs. Indeed, invariants are a set of rules between the values of variables that are extracted in such a way that they remain unchanged in the light of different values of the input variables in the consecutive runnings of a program.

Invariants are usually extracted in three types: pre-condition, loop-invariant and post-condition. Post-conditions are a set of rules between the values of variables when program is executed and finished. However post-conditions are considered as a kind of invariants. In this paper, where it is addressed invariant, it only implies to the invariant of post-condition type.

Daikon is the suitable software for dynamic invariant detection developed until now in comparing other dynamic invariant detection methods. However this method has some problems and weaknesses and thus, many studies have been carried out with the aim of improving Daikon performance which has resulted in several different versions of Daikon up to now [5] and [10]. For instance latest version of Daikon includes some new techniques for equal variables, dynamically constant variables, variable hierarchy and suppression of weaker invariants [5].

Jose and Majumdar have presented a new algorithm for error cause localization based on a reduction to the maximal satisfiability problem called MAX-SAT, which asks what is the maximum number of clauses of a Boolean formula that can be simultaneously satisfied by an assignment [16]. They have just used Boolean Invariants. Konighofer and Bloem have presented a novel debugging method for imperative software, featuring both automatic error localization and correction. The input of our method is an incorrect program and a corresponding specification, which can be given in form of assertions or as a reference implementation. They have used symbolic execution for program analysis [17]. This paper suggests a framework to localize program logical errors by using a serie of tools such as Daikon which derives program invariants [10].

The some invariant-extractor methods will be explained in following section. Briefly, the used method is to first collect a repository of the evaluated programs. Then using their reliabilities with regard to the invariants assigned by an expert the programs are clustered. The clustering is done regarding to their invariants likeliness. For example, all types of sorting programs including bubble sort, merge sort, insertion sort, etc and their invariants stand in the same cluster. LELBPI checks its tested program invariants with all sets of cluster invariants that are available in its repository. After that LELBPI calculates their similarity measures with the clusters and selects the cluster with maximum similarity. If difference number with one set of the clusters invariants in the repository is zero then the program will be true else it will be a new one or belongs to another of the pre-defined clusters; besides it has some error(s) that must be eliminated.

2. INVARIANTS

Invariants in programs are formulas or rules that are emerged from source code of program and remain unique and unchanged with respect to running of program with different input parameters. For instance, in a sort program that its job is to sort array of integers, the first item in the array must be bigger than the second item and the second item must be bigger than the third, etc. Invariants have significant impact on software testing. Daikon is the suitable software for dynamic invariant detection developed until now in comparing other dynamic invariant detection methods. However this method has some problems and weaknesses and thus, many studies have been carried out with the aim of improving Daikon performance which has resulted in several different versions of Daikon up to now [5] and [10]. For instance latest version of Daikon includes some new techniques for equal variables, dynamically constant variables, variable hierarchy and suppression of weaker invariants [5].

Invariants in programs are sets of rules that govern among the values of variables and remain unchanged in the light of different values of the input variables in consecutive runnings of the program. Invariants are very useful in testing software behavior, based on which a programmer can conclude that if its program behavior is true [6] and [15]. For instance, if a programmer, considering invariants, realizes that the value of a variable is unwillingly always constant, s/he may conclude that its codes have some bugs.

Also, invariants are useful in comparing two programs by programmers and can help them check their validity. For instance, when a person writes a program for sorting a series of data, s/he can conclude that his program are correct or has some bugs by comparing his program invariants against the invariants of a famous reliable sort program; such as Merge Sort. Here, the presupposition is that in two sets (a) invariants detected in the program and (b) invariants detected in the Merge Sort program, must be almost the same. Additionally, invariants are useful in documentation and introduction of a program attributes; i.e. in cases where there are no documents and explanations on a specific program and a person wants to recognize its attributes for correcting or expanding program, invariants will be very helpful to attain this goal, especially if the program is big and has huge and unstructured code.

There are two ways for invariant detection that are called static and dynamic. In the static way, invariants are detected with the use of techniques based on compiler issues (for example, extraction of data flow graphs of the program source code). Dynamic way, on the other hand, detects invariants with the help of several program runnings by different input parameter values and based on the values of variables and relations between them. Dynamic methods will be explained in more detail in next section [8].

Every method has some advantages and disadvantages which will be debated in this paper. There are some tools as Key & ESC for java language and LClint for C language for static invariant detection [1] and [4]. In static detection, the biggest problem is the difficulty with which a programmer can discover the invariants. Tracing of codes and detection of rules between variable values are a difficult job especially if the programmer wants to consider such cases as pointers, polymorphisms and so on.

In dynamic methods, the biggest problem is that they are careless and time-consuming and, more importantly, do not provide very reliable answers

First, confirm that you have the correct template for your paper size. This template has been tailored for output on the 21cm X 28cm Paper Size.

3. RELATED WORKS

A. Background

There are many machine learning based and statistical based approaches to fault prediction. Software fault prediction models have been studies since 1990s until now [3]. According to recent studies, the probability of detection (71%) of fault prediction models may be higher than probability of detection of software reviewer (60%) if a robust model is built [14].

Software fault prediction approaches are much more cost-effective to detect software faults compared to software reviews. Although benefits of software fault prediction are listed as follows [2] and [12]:

- Reaching a highly dependable system
- Improving test process by focusing on fault-prone modules
- Selection of best design from design alternative using object-oriented metrics
- Identifying refactoring candidates that are prediction as fault-prone
- Improving quality by improving test process

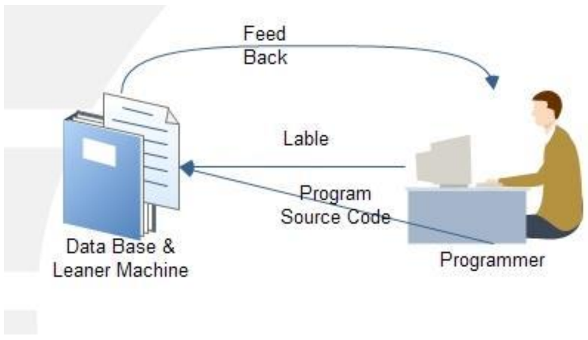


Figure 1. Suggested framework for LELBPI

B. *Daikon Algorithm*

Daikon first runs program with several different input parameters. Then it instruments program and finally in every running of the program saves variable values on a file called data trace file. Daikon continues its work with extracting the values of variables from data trace files and by using a series of predefined relations discovers the invariants and saves them. Daikon discovers unary, binary and ternary invariants. Unary invariants are invariants defined on one variable; for instance, X>a presents variable X is bigger than a constant value. For another example X (mod b)=a shows X mod b=a. X>Y, X=Y+c are also samples of binary invariants and Y=aX+bZ is a sample of ternary invariant considered in Daikon in which X, Y, Z are variables and a & b are constant values.

Daikon will check invariants on the next run of the program on the data trace file and will throw them out from list of true invariants if it is not true on current values of variables. Daikon continues this procedure several times while concluding proper reliability of invariants [8].

C. *Problem Definition and Motivation*

None of previous approaches cannot predict and localize the all of software faults and failures because, the all of impact factors have not been known yet. By the way, data mining and statistical methods essentially have not 100% accuracy. Whereas software failure prediction is very valuable, so many researchers have been doing wide efforts about it. On the other hand, software testing is the most critical and time consuming phase in software development. This paper will be offered a proper model to failure localizes based on invariants.

Programmers or companies that produce programs will lose their credit and profit effectively if one of their programs delivered to a customer has any drawback. While there are many methods to test a program, it is the lack of an appropriate method for localizing a program logical errors using extraction of knowledge from invariants [3]. Also, existent methods cannot assurance that software will be error free. Suggested framework tries to predict all of software logical errors.

4. SUGGESTED FRAMEWORK

Since the similar programs with the same functionalities have more or less the same invariants, such invariants may be considered as behaviors of the programs. Although it is highly probable that there is a program with job similar to job of another program plus an auxiliary job, these two programs are not considered as the same programs. This is due to high rate differences in their invariants. In other words, if the two programs just do the same job, their invariants are almost the same. Suggested framework is explained in this section. Informally, the LELBPI is shown in Figure 1.

Figure 1 depicts that program invariants detected by Daikon software are sent to LELBPI and machine then processes them and give a feedback to programmer. Then a label indicating the validation of LELBPI prediction is returned to it. It is notable that the rules included in the Data Base of the learner machine must be in the general forms. For example to show that an array is sorted, the corresponding rule is similar to "*array_name sorted by >=*" (denoted by *rule 1*); besides the invariants similar to "*array_name[1]<= array_name[2]*" (denoted by *rule 2*) is eliminated provided that the *rule 1* is available. Details of framework are illustrated in activity diagram which is shown in Figure 2.

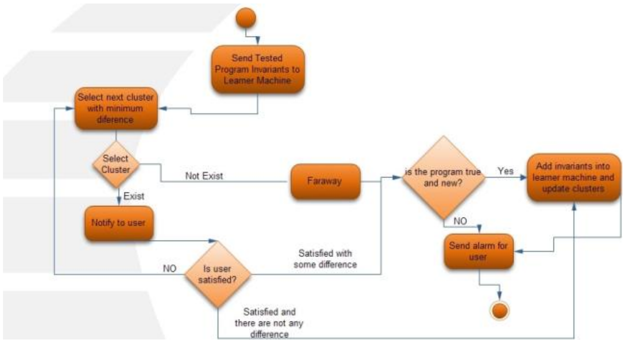
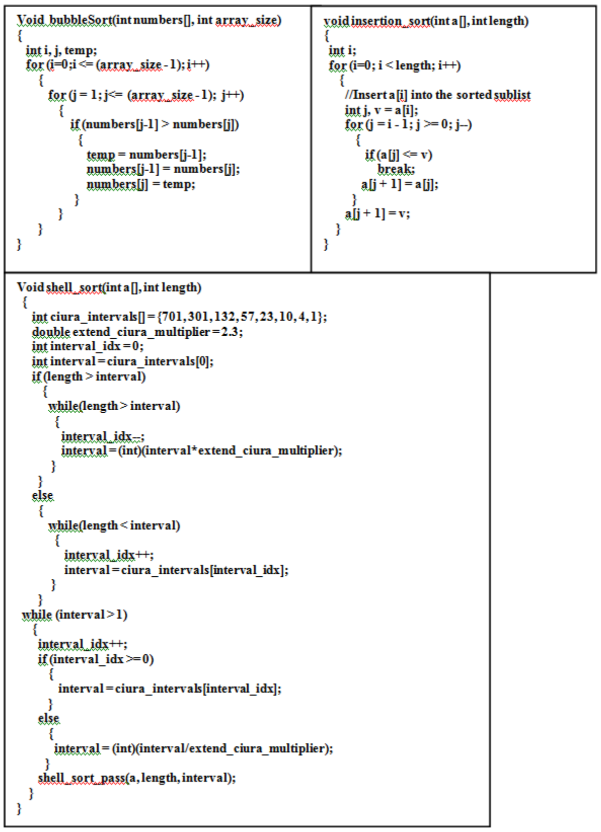

Send Tested Program Invariants to Learner Machine
Select next cluster with minimum diference
Select Cluster
Not Exist
Exist
Faraway
Is the program true and new?
Yes
NO
Add invariants into learner machine and update clusters
Notify to user
Send alarm for user
NO
Is user satisfied?
Satisfied with some difference
Satisfied and there are not any difference

Figure 2. Error localizator framework activity diagram

First, program invariants are sent to LELBPI then invariants are compared with all of clusters agent and select cluster with minimum difference. If all of cluster agents are far away then it will be asked from programmer that "This program is unknown, is it new and reliable?" If user answers ok s/he is sure that program is true, however invariants will insert to the database located in LELBPI and then we update its clusters.

While entered invariants by user is resemblant with some cluster, head of cluster which has minimum difference will be shown to user and if user confirms it, provided that difference number is zero (or be less than a pre-defined threshold) then tested program invariants inserted in the database which locates the right machine learner and updates machine learner clusters, else if difference number is more than zero (or be greater than the pre-defined threshold) then just sends an alarm to user which program has some logical error(s).

While the presented cluster to the user dissatisfies programmer then LELBPI selects the next minimum cluster and sends its matter to user. If all similar clusters presented to user can not satisfy programmer then user is asked again if "This program is unknown, is this program new and reliable?" Again if user answers ok and s/he is sure that program is true, nevertheless invariants will insert to database located in LELBPI and updates its clusters.



```
Void bubbleSort(int numbers[], int array_size)
{
  int i, j, temp;
  for (i=0;i <= (array_size - 1); i++)
    {
      for (j = 1; j<= (array_size - 1); j++)
        {
          if (numbers[j-1] > numbers[j])
            {
              temp = numbers[j-1];
              numbers[j-1] = numbers[j];
              numbers[j] = temp;
            }
        }
    }
}
```

```
void insertion_sort(int a[], int length)
{
 int i;
 for (i=0; i < length; i++)
   {
     //Insert a[i] into the sorted sublist
     int j, v = a[i];
     for (j = i - 1; j >= 0; j--)
       {
         if (a[j] <= v)
           break;
        a[j + 1] = a[j];
       }
     a[j + 1] = v;
   }
}
```

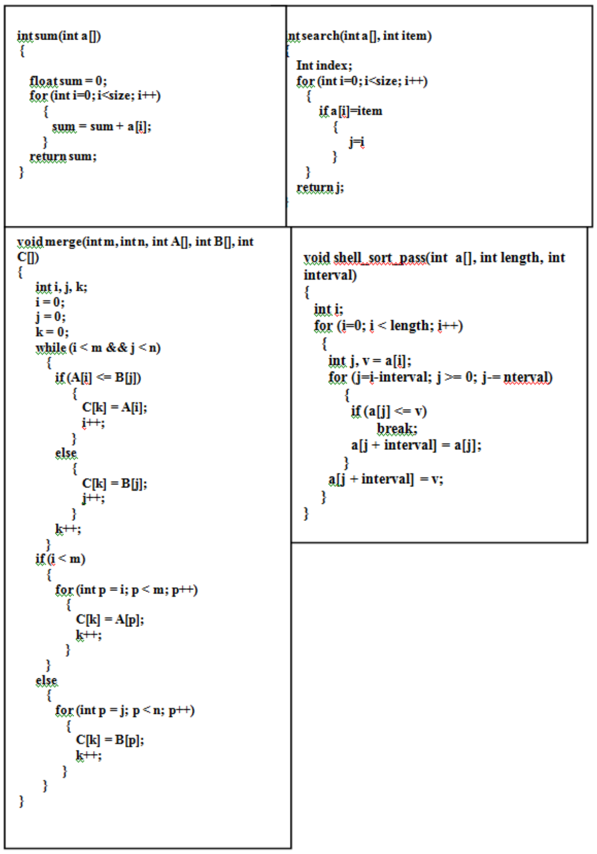
```
Void shell_sort(int a[], int length)
 {
   int ciura_intervals[] = {701,301,132,57,23,10,4,1};
   double extend_ciura_multiplier = 2.3;
   int interval_idx = 0;
   int interval = ciura_intervals[0];
   if (length > interval)
     {
      while(length > interval)
        {
         interval_idx--;
         interval = (int)(interval*extend_ciura_multiplier);
        }
     }
   else
     {
      while(length < interval)
        {
         interval_idx++;
         interval = ciura_intervals[interval_idx];
        }
     }
  while (interval > 1)
    {
     interval_idx++;
     if (interval_idx >= 0)
       {
        interval = ciura_intervals[interval_idx];
       }
     else
       {
        interval = (int)(interval/extend_ciura_multiplier);
       }
     shell_sort_pass(a, length, interval);
   }
}
```

Figure 3. Some exemplary programs

It is clear that learner machine database is empty initially and it will gradually be filled by adding true and reliable programs manually by user feedbacks. The greater the number of record in database, the more accurate and valid results of suggested framework

A. Variable Matching

Suppose that the agent of cluster that must be compared with the tested program invariants had two integer variables denoted by *i* and *j.* Also assume that entered invariable to learner machine had two different integer variable denoted by *m*, *n*. Before comparing invariants, LELBPI must match either $i \rightarrow m$ and $j \rightarrow n$ or vice versa. Algorithm that is used in LELBPI matches all possible permutations of *i, j* to *m*, *n* and amount of difference on each combination will be computed and software will select combination which has minimum difference in compare with other combination. More detailed explanation is offered in section five.



```
int sum(int a[])
{

  float sum = 0;
  for (int i=0; i<size; i++)
     {
       sum = sum + a[i];
     }
  return sum;
}
```

```
nt search(int a[], int item)
{
 Int index;
 for (int i=0; i<size; i++)
   {
     if a[i]=item
       {
          j=i
       }
   }
 return j;
```

```
void merge(int m, int n, int A[], int B[], int
C[])
{
   int i, j, k;
   i = 0;
   j = 0;
   k = 0;
   while (i < m && j < n)
     {
       if (A[i] <= B[j])
          {
            C[k] = A[i];
            i++;
          }
       else
          {
            C[k] = B[j];
            j++;
          }
       k++;
     }
   if (i < m)
     {
       for (int p = i; p < m; p++)
         {
           C[k] = A[p];
           k++;
         }
     }
   else
     {
       for (int p = j; p < n; p++)
         {
           C[k] = B[p];
           k++;
         }
     }
}
```

```
void shell_sort_pass(int  a[], int length, int
interval)
{
  int i;
  for (i=0; i < length; i++)
    {
     int j, v = a[i];
     for (j=i-interval; j >= 0; j-= nterval)
       {
        if (a[j] <= v)
             break;
        a[j + interval] = a[j];
       }
     a[j + interval] = v;
    }
}
```

Figure 4. Second set of exemplary programs

For decreasing runtime, LELBPI standardizes variable names before invariants of true program are added to database, it means that all variables with the same data type rename to one series of standard and recorded name. For example all integer variables in all programs are renamed to int_a, int_b... sequentially and all integer arrays in programs are renamed to array_int_a, array_int_b… sequentially, and for each invariant set, the numbers of variables in all data types are buffered. Machine can permute variables with higher speed.

B. Scalability

For justifying this method is convenient and efficient, run time of this method is estimated on this section. Informally run time can be characterized as:

$$\text{TIME=O} \left(Inv + K \times \binom{Max(f,r)}{Min(f,r)} \times m \times g \right)(1) \qquad \alpha + \beta = \chi. \qquad (1) \qquad (1)$$

where Inv is the run time for extracting tested program invariants, *K* is cluster number, *f* is maximum of number of variables of one type in tested program, *r* is the maximum number of variables if one type in agent of clusters, *m* is maximum number of invariants in agent of clusters, and finally *g* is maximum of invariants number in tested program invariants. As you saw, first of all, the framework must extract tested program invariants with an appropriate tool as

Daikon. Then, these invariants must be compared with invariants of every clusters agent. Also, every combine of variables in invariants must be matched with variables in same type in agent of clusters.

Pay attention that *f* and *r* are limited because as is said in section 4.1, before than adding invariants in database variables are standardized. So, run time is acceptable.

5. EXPERIMENTAL RESULTS

For validating this framework, software is implemented and their results are shown in this section. Invariants of six true programs including bubble sort, merge sort, insertion sort, shell sort, compute sum of array elements and search into array are added to software database initially. Used algorithms are shown in Figure 3 and Figure 4.

The results of pre-mentioned algorithms with this supposition that every used array in programs have random values collected in Table 1. It is necessary to note that all of invariants in this paper come from Daikon software.

In Table 1 "a sorted by <=" means that *a[i]<=a[i+1]*. Also "a=orig (a [])" means that "a" array elements remain unmodified in exit of program.

With respect to invariants in Table 1, two clusters are created and programs are grouped in the two. These two clusters are related to *sort* and *search*. Agent of sort cluster is "a sorted by <=" and agent of search cluster is "a=orig (a [])".

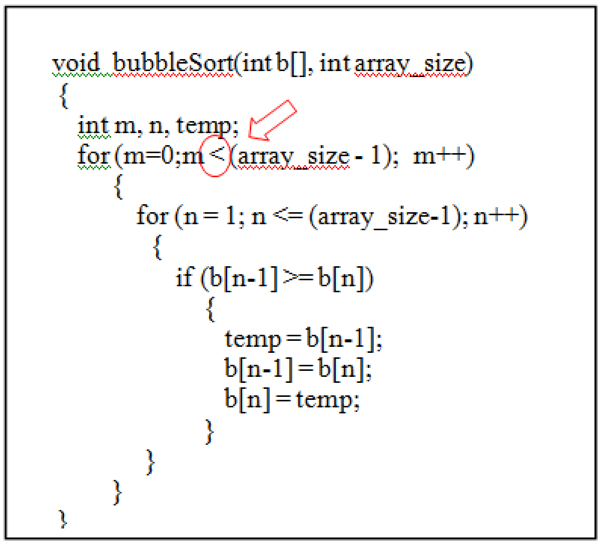
Now consider below code which is bubble sort that programmer don’t check last element of array. As you know, it is a common error in programming. This code has an array that its name is *b* and two integer value with names *m, n*.

TABLE I. PROGRAMS AND THEIR INVARIANTS

Row	Program Name	Invariants
1	Bubble Sort	1. a sorted by >= 2. i= Length(a) 3. J=Length(a)-1
2	Insertion sort	1. a sorted by >= 2. i= Length(a)-1
3	Shell Sort	1. a sorted by >= 2. interval=1
4	Merge Sort	1. a sored by >= 2. b sorted by >= 3. c sorted by >= 4. i < k 5. j <k 6. k= Length(c) 7. a=orig(a[]) 8. b=orig(b[])
5	Sum of array elements	1. a =orig(a[]) 2. i= Length(a)-1
6	Search array	1. a=orig(a[]) 2. i<= Length(a)-1

Invariants for algorithm 1 in Figure 5 that are calculated by Daikon software with this supposition that length of array is six is shown at below:

1. m= Length(b)-1
2. n= Length(b)-1
3. b[0]<=b[1]
4. b[1]<=b[2]
5. b[2]<=b[3]
6. b[3]<=b[4].



```
void  bubbleSort(int b[], int array_size)
{
  int m, n, temp;
  for (m=0;m < (array_size - 1);  m++)
    {
      for (n = 1; n <= (array_size-1); n++)
       {
         if (b[n-1] >= b[n])
           {
             temp = b[n-1];
             b[n-1] = b[n];
             b[n] = temp;
           }
       }
    }
}
```

Figure 5. Algorithm 1: Faulty version of bubble sort

Number of differences in invariants of every cluster agent and these invariants are shown in Table 2. Here, machine just compares invariants which exist on variables in cluster agent. It is because tested program may do anything as well as agent function. So if tested program has any invariants on those variables that are missing in set of variables in comparing agent of cluster then these invariants can't be considered as differences. Also because invariants of an agent include intersection of invariants of all programs in the cluster, some invariants may be in invariants of the current programs while they are not in the invariant of its cluster agent. However invariants on variables in cluster agent must be strictly in tested program invariants.

Table 2 presents that tested program stands in sort cluster and it have sort matter. Now tested program will be compared with all of program invariants in sort cluster. Results of these comparisons are collected in Table 3.

TABLE II. COMPARE INVARIANTS

Cluster Name	Variable Matching	Differences
Sort	b[]=a[]	3
Search	b[]=a[]	10

It is clear from Table 3 that tested program has minimum differences from bubble sort provided that *m* variable be assigned to *i* variable and *n* be assigned to *j*. It is clear that it is a valid result. Terminal result is: program subject is "sorting" and has some logical errors because number of differences is not zero.

TABLE III. TABLE TYPE STYLES

Program Name	Row	Variables Matching	Differences

Bubble Sort	1	b[]=a[]
		m=i
		n=j
			1

Bubble Sort	2	b[]=a[]
		m=j
		n=i
			3

Insertion Sort	3	b[]=a[]
		m=i
			2

Insertion Sort	4	b[]=a[]
		n=i
			3

Merge Sort	5	b[]=a[] or b[]=b[]
		m=k
		n=i
			10

Merge Sort	6	b[]=a[] or b[]=b[]
		m=k
			10

		n=j	

Merge Sort	7	b[]=a[] or b[]=b[]
		m=i
		n=j
			10

Merge Sort	8	b[]=a[] or b[]=b[]
		m=j
		n=i
			10

Merge Sort	9	b[]=c[]
		m=i
		n=k
			4

Merge Sort	10	b[]=c[]
		m=j
		n=k
			4

Merge Sort	11	b[]=c[]
		m=k
		n=i
			4

Merge Sort	12	b[]=c[]
		m=k
		n=j
			4

Merge Sort	13	b[]=c[]
		m=i
		n=j
			4

Merge Sort	14	b[]=c[]
		m=j
		n=i
			4

6. CONCLUSION AND FURTHER WORKS

Daikon is a method to discover likely invariants by dynamic methods. Also Daikon's team has been doing many researches about invariants application. They also do some researches to test software based on execution program with several different input parameters and extract and check invariants on every run of program. In this paper a new framework based on Daikon, is proposed to incrementally detect errors of different programs. In this framework, one cluster is produced per invariants of each program type. The suggested framework is gradually reinforced. It can predict every error that has been learned until now. One of this framework limitations are that the user who learn it must be expert. If the user was expert and learner machine has been learned sufficiently, it can predict and identify programs that have logical error(s).

For future direction of research will focus on program filtering in every cluster that may reduce efficiently comparing tasks. For example as one of filtering, machine can just compare programs which have same variable data type for example have three integer values and two array data types. Another action that can be done is that machine as well as true program invariants can learn from false program invariants and machine learns that programmer where and how have fault in program commonly.

ACKNOWLEDGMENT

This work is supported by Islamic Azad University, Zarghan Branch, Iran.

REFERENCES

[1] B. Weiβ., "Inferring Invariants by Static Analysis in KeY", Thesis, Karlsruhe university, 2007.

[2] C. Catal, B. Diri, "Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem", Information Sciences, vol. 179, no. 8, pp. 1040–1058, 2009.

[3] C. Catal, "Software fault prediction: A literature review and current trends", Expert Systems with Applications, vol. 38, pp. 4626-4636, 2011.

[4] D. Evans, J. Guttag, J. Horing, Y.M. Tan, "LCLint: A Tool for Using Specification to Check Code", Proc.Second ACM SIGSOFT Symp, pp. 87-96, 1994.

[5] J.H. Perkins, M.D. Ernst, "Efficient Incremental Algorithms for Dynamic Detection of Likely Invariants", Proc. ACM SIGSOFT Symp, pp. 23-32, 2004.

[6] J.W. Nimmer, M.D. Ernst, "Automatic Generation of Program Specifications", Proc. Int"l Symp. Software Testing and Analysis, pp. 229-239, 2002.

[7] M.B. Dwyer, L.A. Clarke, "Data Flow Analysis for Verifying Properties of Concurrent Programs", Proc. Second ACM SIGSOFT Symp, pp. 62-75, 1994.

[8] M.D. Ernst, J. Cockrell, W.G. Griswold, D. Notkin, "Dynamically discovering likely program invariants to support program evolution", IEEE TSE, vol. 27, no. 2, pp. 99–123, 2001.

[9] M.D. Ernst, J.H. Perkins, P.J. Guo, S. McCamant, C. Pacheco, M.S. Tschantz, C. Xiao, "The Daikon System for Dynamic Detection of Likely Invariants", Science of Computer Programming, vol. 69, no. 1-3, pp. 35-45, 2007.

[10] M.D. Ernst, W.G. Griswold, Y. Kataoka, D. Notkin, "Dynamically Discovering Program Invariants Involving Collections", Technical Report UW-CSE-99-11-02, 2000.

[11] N.D. Jones, F. Nielson, "Abstract interpretation: A semanticsbased tool for program analysis", Handbook of Logic in computer Science, vol. 4, pp. 527–636, 1995.

[12] R. Lencevicius, U. Ho È lzle, A.K. Singh, "Query-Based Debugging of Object-Oriented Programs", Proc. Conf. Object-Oriented Programming, Systems, Languages, and Applications, pp. 304-317, 1997.

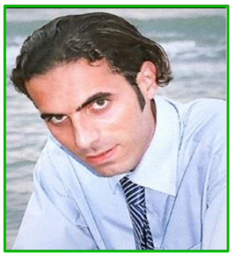
[13] Robert, W. Floyd, "Assigning meanings to programs. In Symposium on Applied Mathematics", American Mathematical Society, pp. 19–32, 1967.

[14] T. Menzies, J. Greenwald, A. Frank, "Data mining static code attributes to learn defect predictors", IEEE Transactions on Software Engineering, vol. 33, no. 1, pp. 2–13, 2007.

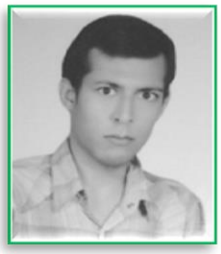
[15] Y. Kataoka, M.D. Ernst, W.G. Griswold, D. Notkin, "Automated Support for Program Refactoring Using Invariants", Proc. Int"l Conf. Software Maintenance, pp. 736-743, 2001.

[16] M. Jose, R. Majumdar, "Cause clue clauses: error localization using maximum satisfiability", 32nd ACM SIGPLAN conference on Programming language design and implementation, pp. 437-446, 2011.

[17] R. Konighofer, R. Bloem, Automated error localization and correction for imperative programs, Formal Methods in Computer-Aided Design (FMCAD), pp. 91-100, 2011.



Hamid Alinejad-Rokny - is a member of The University of New South Wales, Sydney, Australia. He is the author/co-author of more than 55 publications in technical journals and conferences. He served on the program committees of several national and international conferences. He was Guest Editor-Chief for special issue at IJFIPM. Also He is Deputy Editor-Chief at International Journal of Software Engineering And Computing and he is editorial board member at IJSEI, IJFIPM, JETWI, IJSCIP, IJCSCS, IJCNT and IJEIS. His research interests are in the areas of Data Mining, Bioinformatics, Artificial Intelligence and Biological Computing



Hamid Parvin - born in Nourabad Mamasani in 1984. He received his B.Sc. degree from Shahid Chamran University in Software Engineering in 2006. He received his M.Sc. degree in Artificial Intelligence from Iran University of Science ad Technology, Tehran, Iran under supervision of Behrouz Minaei-Bidgoli in 2008. He received PhD degree in Artificial Intelligence in Iran University of Science and Technology, Tehran, Iran under supervision of Behrouz Minaei-Bidgoli. He has published several journal papers among which there are 11 SCIE indexed. He has also published many papers in various book chapters. He is now a faculty member in the Islamic Azad University, Nourabad Mamasani Branch. His research interests are ensemble-based learning, evolutionary learning and data mining.

Mohamad Iman Jamnejad - born in Shiraz in 1984. He received his B.Sc. degree from Sajad University in Software Engineering in 2006. He received his M.Sc. degree in Software Engineering from Islamic Azad University, Science and Research Branch, Ahvaz, Iran. His research interests are robotic and data mining.

Ali Heidarzadegan - born in Shiraz in 1981. He received his B.Sc. degree from Meibod University in Software Engineering. He is a student in M.Sc. degree in Software Engineering in Islamic Azad University, Science and Research Branch, Yasuj, Iran under supervision of Hamid Parvin. His research interests are ensemble-based learning and data mining.