On Enrichment of Dynamically Detected Invariants in Arrays Case

Hani Fouladgar, Zahra Rezaei, Mohsen Parvin, Alinejad-Rokny Hamid
Science and Technology Branch
Islamic Azad University
Iran
{fouladgar, rezaei, m.parvin}@iust.ac.ir, alinejad@gmail.com

ABSTRACT: *Software engineering is the work of designing, implementing and modifying of software to build software fast and have a high quality, efficient and maintainable software. Invariant helps programmer and tester to perform some aspect of software engineering. Since arrays and pointers are more probable to be faulty, invariants which report these properties are more useful. By presenting first and last elements of arrays we can detect the carelessness in using index which mostly happens in loops. The idea of employing number of mutual elements between two same type arrays can help us to catch faults in the cases that an array is gained from changes in another array. These two properties help us to capture lots of prevalent faults and therefore try to remove them. This paper proposes an interesting sort of extension over Daikon like tools. It can generate more invariant in the case of array.*

Keywords: Dynamic invariant detection, Software testing, Array property, Array's first and last elements, Mutual element between arrays

Received: Received 10 June 2011, Revised 2 July 2010, Accepted 5 July 2011

1. Introduction

Invariants are some properties of different program points which are true in all executions of the program. These properties can be seen in *formal specification* or *assert statement.* For example in a sort program that its job is to sort array of integers, invariant (array a sorted >=) is reported. This idea is introduced in [1]. Invariants develop data structures and algorithms and are employed in all aspects of software engineering from design to maintenance [2]. Invariants represent program properties; hence after an update to the code, invariants can show properties which remain unchanged or those that are violated through code revision. Invariants, somehow, are treated such as *documentation* and *specification* of a program. Specification is used in all software engineering steps like design, coding, verification, testing, optimization and maintenance. Invariants might be detected by *static* and *dynamic* approaches.

Static analysis checks syntactic structure and runtime behavior of program without actually running [3]. Static analysis is an entirely automated method. *Data-flow analysis* is a traditional technique which is employed in the compilers in order to collect necessary information for optimization. Data-flow analysis can determine the properties of program points. In practice the collected information is considered invariant. *Abstract interpretation* is a theoretical framework for static analysis [4]. The most precise imaginable abstract interpretation is called the *static semantics* or *accumulating semantics*. Another related static analysis technique is *predicate abstraction*. In this technique, an absolute and finite set of predicates is used to induce abstract domains.

Dynamic invariant extraction from program context is brought to software engineering realm during recent ten years. In contrast to static analysis, dynamic invariant detection extracts program properties and information by executing of the program with different inputs. Dynamic invariant detection, first time, was quoted by *Daikon* [2] - a full-featured and robust implementation of dynamic invariant detection. With the help of test suits and different executions of a program as well as using an invariant inference system, properties of specific point of program (usually function entries and exits) are revealed. These points, lexically, are named *program points*. The properties are not certainly true, but indeed, are determined through several executions on test cases with specific confidence. An important attitude of process of dynamic invariant extraction is that what invariant presents is not only program behavior but also indicates assumption of test cases. This nature of invariant causes double usage of it, so dynamically-produced invariant determines program specification more properly.

This paper concentrates on the dynamic extraction of invariants. We introduce some ideas which improve the effect of invariants that could be useful in bug detection and testing. We propose two extensions to the Daikon like tool for invariant detection. The extensions are concerned with the treatment of arrays. We employ more properties of arrays to get better results. The rest of this paper starts with related work (section 2) and continues with paper terminology (section 3) and contributions (section 4). Then we bring some simple examples to clarify the ideas (section 5). In section 6 some actual examples and adjustment for our ideas are presented. We evaluate our idea in section 7. Finally we conclude the paper and talk about future work (section 8).

2. Related Work

In this section, we discuss some implementations of dynamic invariant detection. Many valuable efforts have been done in this field but we mention here only these ones which are more relevant.

Dynamic invariant detection, as mentioned, is quoted by *Daikon* [2]. Daikon is the most prosperous software in dynamic invariant detection developed until now, comparing with other dynamic invariant detection methods [2]. However this software has some problems out of which the most serious one is being time-consuming.

DySy proposes a dynamic symbolic execution technique to improve the quality of inferred invariant [7]. It executes test cases like other dynamic invariant inference tools but, as well, coincidentally performs a symbolic execution. For each test unit, DySy results in program's path conditions. At the end, all path conditions are combined and build the result.

Software Agitator is a commercial testing tool which is represented by Agitar and is inspired by Daikon [5]. Software agitation is a testing technique that joins the results of research in test-input generation and dynamic invariant detection. The results are called *observations*. Agitar won the Wall street Journal's 2005 Software Technology Innovation Award.

The DIDUCE tool [6] helps the programmer by detecting errors and determining the root causes. Besides detecting dynamic invariant, DIDUCE checks the program behavior against extracted invariants up to each program points and reports all detected violations. DIDUCE checks simple invariants and does not need up-front instrument.

3. Terminology

In this section we bring some terms which are used frequently in this paper. The aim of the section is to help readers obtain a better perception of the paper.

Definition 1. *Invariants* could be defined as prominent relation among program variables. Invariants in programs are formulas or rules that are emerged from the source code of a program and remain unique and unchanged with respect to the running phase of a program with different parameters.

Definition 2. *Program points* are specific points in a program, such as the *Enter* or *Exit* point of a function, which serve as report points for variable relations and invariants. Most frequent program points in use are the Enter and Exit points of sub-programs and functions.

Definition 3. *Pre-conditions* of a program point are the conditions, relations and invariants that hold immediately before approaching to that program point. In the case of sub-programs or a function *Enter* point of a sub-program or a function acts as its pre-condition.

Definition 4. *Post-conditions* of a program point are the conditions, relations and invariants that hold immediately after leaving from that program point. In the case of sub-programs, a function *Exit* point of a sub-program or a function is considered as its post-condition of it. Typically, post-condition also contains relations between the original value of a variable and its modified one (before and after that program point). In other words, invariants in post-conditions contain relations between variables in pre-condition and post-condition.

4. Paper Contributions

Software testing is one of the most time consuming parts of software engineering because regarding inputs, different executing paths happen and unchecked paths can be defective. Even if a software tester checks all paths, the code can be faulty. In this situation, because of their structure, *arrays* and *pointers* are more probable to be faulty. In the C program language, an array is a kind of pointer. Therefore if any improvement is achieved for the handling of arrays, it can be simultaneously considered as an improvement for the pointers. Another source of fault points can be the *exceptions* or the *try-catch* blocks in our case. Exceptions are handled by the catch block and this prevents the programmer or the tester form detecting the faults. This means that a program might operate as expected but it actually has some faults.

The first and the last elements of an array possess very crucial properties because these elements are impacted by the carelessness in using the indexes. By involving some array elements in invariant detection, a dramatic improvement in fault detection might happen. The number of these elements can be the least size of an array or they can be optional. This contribution exposes inattention in using index which mostly happens with the first and the last indexes and corresponding to the first and last elements of an array.

Besides employing array elements, enlisting the number of mutual elements of same type arrays for each program point is useful in detecting faults. In other words, for each program point, the number of elements' values which are shared in two different same type arrays is employed in invariants detection. It helps the programmer to evaluate his program in the cases that an array is gained from changes in another array. The mutual elements show the correct elements which should be unchanged through the process. We discuss more about this contribution in the next sections and clarify the number of mutual elements of same type arrays for each program point.

Overall our contributions comprise the following:

- Adding the number of first and last elements of an array as new variables for invariant detection.
- Enlisting the number of mutual elements of same type arrays for each program point

5. Illustration of Contributions

To clarify the contributions we discussed earlier, we present some pieces of program code and their post-condition invariant. The programs are in the form of pseudo-code and do not assume the use of any specific programming. For presenting our ideas, the *Exit* program point invariants which represent post-condition properties for a program point are used because post-condition properties can show both the pre-condition and post-condition values of variables.

5.1 First and Last Elements of Array

For invariant detection, we might propose the use of the number of first and last elements of an array. The number of these elements can be the least size of the array or can be optional. This contribution exposes carelessness in using index which mostly happens to first and last indexes and corresponding to the first and last elements of array. To illustrate the effeteness of this view, consider Figure 1.

Figure 1 presents the code of bubbleSort() function. It accepts 2 as input one of which is the array and another is the length of the array. The output is the sorted array. In this example, the index *j* starts at 1 instead of 0 so the first element of array is not considered in the sorting. With the contribution of the produced invariants from the first and last elements of the array the fault is detected. Adding the first and last elements of each array, as part of the related invariants in the *Exit* point of the bubbleSort() function is illustrated in Figure 2.

The presented invariants in Figure 2 are in the form of Daikon output. For array *x*, *x[-1]* is the last element of *x*, *x[-2]* the element before the last one and so forth. In Figure 2, line 17 shows that the first element of the input array always equals to the first element of the return value. Lines 18 to 23 show that the rest of the elements are sorted. Therefore obviously only the first element is never involved in sorting. This helps the programmer to detect the fault.

```
int  *bubbleSort(int  *digits,int  length)
{
   int  *numbers;
   number<-digits;
     for(i=1;i<length;i++)
         for(j=1;j<length-i;j++)
             if(numbers[j]>numbers[j+1])
           {
                int  temp=numbers[j];
                 numbers[j]=numbers[j+1];
                numbers[j+1]=temp;
           }
      return  numbers;
}
```

Figure 1. Program A: Inattention in using index

5.2 Number of Mutual Elements Between two Arrays

Another property, which can help the programmer, might be the number of mutual elements of same type arrays for each program point. It is helpful for a programmer to test the code in situations that an array comes from another one. To illustrate the idea you may consider Figure 3.

```
1     digits[]  >=  return[](lexically)
2     digits[]  ==  orig(digits[])
3     orig(length)  ==  size(return[])
4     return  !=  null
5     return[1]  in  digits[]
6     return[2]  in  digits[]
7     return[3]  in  digits[]
8     return[]  elements  <=  return[-1]
9     digits[]  elements  <=  return[-1]
10   return[1]  in  digits[]
11   return[2]  in  digits[]
12   return[3]  in  digits[]
13   return[-1]  in  digits[]
14   return[-2]  in  digits[]
15   return[-3]  in  digits[]
16   return[-4]  in  digits[]
17   return[0]  ==  digits[0]
18   return[1]  <  return[2]
19   return[2]  <  return[3]
20   return[3]  <  return[-4]
21   return[-4]  <  return[-3]
22   return[-3]  <  return[-2]
23   return[-2]  <  return[-1]
24   length[0]  !=  return[0]
```

Figure 2. Related invariants to the code of Figure 1

Exit point invariants of replace() is shown in the Figure 4

```
void replace(int *d, int l, int m, int n)
{
   int i;
   for(i=0; i < l; i++)
      if(d[i] == m)
      {
         d[i] == n;
         break;
      }
}
```

Figure 3. Program B: An example of "replace code"

In Figure 4 invariants in lines 6 and 7 show the number of mutual elements between *d* (the first parameter of the function) and the return value. As expected, the number of mutual elements between *d* and the return value equals to the number of mutual elements between *orig(d)* and the return value. Also, the number of mutual elements between *d* and the return value equals to the size of *d* minus 1. The latter confirms the validity of the program since the number of mutual elements equals to the size of *d* minus 1. However, besides this invariant, other invariants quote that the return value is not sorted despite *d* is sorted and this might be a fault in the program.

```
1   d[]==orig(d[])
2   orig(l)==size(return[])
3   d[] sorted by <
4   return != null
5   orig(m) in d[]
6   Mutual(d[],return[])==Mutual(orig(d[])),return[])
7   Mutual(d[],return[])==size(d[])-1
8   d[] elements <= d[-1]
9   orig(n) in return[]
10  orig(l) in d[-1]
11  orig(l) in return[-1]
12  orig[m] !=size(d[])-1
13  orig[m] < d[-1]
14  orig[m] != return[-1]
15  orig[n] != d[-1]
16  d[-1] != return[-1]
```

Figure 4. Related invariants to the replace code of Figure 3

6. Actual Examples and Justification

In this section we describe our ideas by using some programs as actual examples. These examples show how our ideas help the programmer to detect the faults and subsequently properly address them. These examples comprise small and rather simple subprograms. Our reasonable assumption is that every program, either big or small, can be divided in small parts and might be raised in small subprograms. In other words, in all programs, when working with arrays the programmer uses iteration expressions such as the "for" block and carelessness can result independently of whether the program is big or small. Therefore to illustrate our justification we bring some actual but not large subprograms, which are caused "gold standard" invariants [9]. The goal of adding these examples is to validate our ideas and to display the feebleness of the previous efforts. Like in the previous section, the presented code does not assume the use of any specific programming language and but they comprise just pseudo-codes.

6.1 Try-Catch and Effectiveness of the Ideas

We start our justification with function the AVG() which contains a *Try-Catch* statement. This function is presented in Figure 5. AVG() has 4 parameters. It accepts an array (a[]) and the length (l) and sums all the elements in sum, then divides each array element by n/5 and finally returns the sorted array. Although the programmer has considered that if n is zero a division-by-zero happens and prevented it from happening by introducing an if-condition statement, the code has a fault. “temp” has been declared as an integer and for 0 < n < 5, n/5 is zero subsequently the variable temp can become 0 as well, and therefore division-by-zero happens. In these situations a division-by-zero exception is thrown and the return array has all its elements equal to 0 instead of being the sorted input array.

```
int*  AVG(int*,a,int  l,long*  sum,int  n)
{
    int  i,j,*numbers,temp;
    numbers=malloc(sizeof(int[l]));
    numbers[]<-0;
   Try
   {
      *sum  =  0;
       for(i  =0;i<l;i++)
      {
          *sum  =  a[i]  +  *sum;
         if(n  !=  0)
          {
             temp  =  n/5;
             a[i]  /=  temp;
           }
       }
        numbers  =  Sort(a,  l);
      }
      catch(e)
       {
          *sum  =  0;
       }
        return  numbers;
  }
```

Figure 5. Program C: First example for the justification of the proposed algorithm

In Figure 6, we present the related invariants in the *Exit* program point of the function. As mentioned earlier, invariants are in the form of Daikon output but here we add also our proposed part. Line 9 shows that the return values are sorted. By solely considering this invariant, the program appears to work properly. However, by considering lines 10 to 16 and specially lines 17 and 34, it is obvious that in some situations the sorting of the array is not reached. Lines 10 to 16 show that in some cases all the return values are equal to 0. In line 17 we observe that mutual elements between a[] and the return values can be zero and in line 34 the mutual elements between a[] and the return values can be less than l whereas it is expected to be equal to l. Over-all, we detect that in some cases the return values are not the sorted elements of a[].

6.2 A Real Code for Justification

Another example presented here as a justification has been coded in our lab and we show how we manage to detect our intentional fault by adopting our proposed method. Function mix() accepts 4 parameters as input, two arrays and their lengths. This function merges the second half elements of the first array with the inverse of the second half elements of the second array and returns the merged array. All the values of the arrays’ elements are greater than 1. The arrays’ lengths are even and dividable by 2. Consider Figure 7.

In line 7 of the subprogram, *i* is initialized to *la/2-1* instead of *la/*2 and it causes the last element of b[] to be put on last element of a[]. Fig.8 presents the result of the function.

The *Exit* program point invariants of the mix() are presented in Figure 9. Line 7 seems to be false because the last element of the output always equals to 0. Observing the same value in all executions of the program is suspicious and that being equal to 0 is a serious fault because a[] and b[] values are greater than 0. Additional helpful invariants are the ones in lines 13 and 14. Line 13

```
1   a[]  >  return[](lexically)
2   a[]  >=  return[](lexically)
3   a[]  ==  orig(a[])
4   sum  >  return[](lexically)
5   sum  >=  return[](lexically)
6   orig(1)  ==  size(return[])
7   return  !=  null
8   return[]  elements  >=  0
9   return[]   sorted  by  <=
10  return[0]  elements  >=  0
11  return[1]  elements  >=  0
12  return[2]  elements  >=  0
13  return[3]  elements  >=  0
14  return[-2]  elements  >=  0
15  return[-3]  elements  >=  0
16  return[-4]  elements  >=  0
17  Mutal(a[],return[])  >=  0
18  a[]  elements  >=  return[0]
19  a[]  elements  >=  orig[n]
20  sum  >  Mutal(a[],return[])
21  sum  >  orig(1)
22  sum  >  orig(n)
23  sum  >  a[-1]
24  sum  >  return[orig(n)]
25  return[]  elements  >=  return[0]
26  return[]  elements  <=  return[-1]
27  return[0]   <=  return[1]
28  return[1]   <=  return[2]
29  return[2]   <=  return[3]
30  return[3]   <=  return[-4]
31  return[-4]   <=  return[-3]
32  return[-3]   <=  return[-2]
33  return[-4]   <=  return[-1]
34  Mutal(a[],return[])  <=  orig(1)
35  orig(1)  <  a[-1]
```

Figure 6. Related invariants to the code of Figure 5

```
1  int*  mix(int*,  a,int*  a,int*  b,int  1a,int  1b)
2  {
3     int  i,j;
4      int*  n=malloc(sizeof(int[1a/2+1b/2]));
5     for(i  =0;  i  <  1a/2;  i++)
6        n[i]  =  a[1a/2+i];
7     for(i  =1a/2-1;j=1b-1;  j>=1b/2;  i++;  j--)
8       n[i]  =  b[j];
9    return   n;
10   }
```

Figure 7. Program D: Second real example code for justification of the proposed algorithm

indirectly shows the value of mutual elements of a[] and return[] is not equal to la/2 but line 14 is acceptable. Finally consider lines 17 and 18. Twice the value of mutual values between return[]

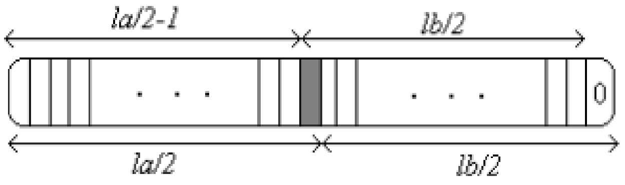

Figure 8. Schematic work of the code of Figure 7

The *Exit* program point invariants of the mix() are presented in Figure 9. Line 7 seems to be false because the last element of the output always equals to 0. Observing the same value in all executions of the program is suspicious and that being equal to 0 is a serious fault because a[] and b[] values are greater than 0. Additional helpful invariants are the ones in lines 13 and 14. Line 13 indirectly shows the value of mutual elements of a[] and return[] is not equal to la/2 but line 14 is acceptable. Finally consider lines 17 and 18. Twice the value of mutual values between return[] and a[] (or b[]) plus orig(lb) (or orig(la)) is equal to twice the value of return length. Line 18 is completely true but line 17 is false.

Over-all, by considering all shown invariants, it is detected that the subprogram does not work properly and the elements are not put in their place correctly.

6.3 A Comparison with Original Daikon

In this subsection we do a comparison to present the effect of our ideas. We consider Daikon as a bench mark. In Figure 1, we present a wrong implementation of the "bubble sort" function. In subsection 4.1 we described the whole discussion about this subprogram. Figure 2 presents the related invariants which include our proposal. Now in Fig.10 the Daikon invariants of this subprogram are presented.

```
1   a[]  ==  orig(a[])
2   b[]  ==  orig(b[])
3   b[]  elements  >=  2
4   la  +  lb-2  *  size(return[])  ==  0
5   return  !=null
6   return[-1]  elements  >=  0
7   return[-1]  ==  0
8   a[]  elements  >  return[-1]
9   b[]  elements  >  return[-1]
10  return[-1]  <  return[-2]
11  return[-1]  <  return[-3]
12  return[-1]  <  return[-4]
13  Mutual(a[],return[])  !=  return[])  !=  Mutual(b[],return[])
14  2  *  Mutual(a[],return[])  -  2  *  orig(la)  +  2  ==  0
15  orig(lb)  %  Mutual(b[],return[])  ==  0
16  Mutual(a[],return[])  +  Mutual(b[],return[])  -  size(return[])  +  1  -  0
17  2  *  Mutual(a[],return[])  -  2  *  orig(lb)  -  2  *  size(return[])  +  2  -  0
18  2  *  Mutual(b[],return[])  -  2  *  orig(la)  -  2  *  size(return[])  -  0
```

Figure. 9 Related invariants to the code of Figure 7

As shown, the invariants in Fig.10 do not help in detecting the fault in the subprogram. As a matter of fact, the return values are almost sorted; lines 6 and 9, unlike reality, they suggest that the program works properly and returns the sorted array. It is worth to say that Fig.10 shows all the related invariants which are taken by Daikon in the *Exit* program point of bubbleSort().

7. Evaluation

As previously quoted we propose two extensions to Daikon like tools. Beside there are arrays invariant, we consider these two ideas as crucial properties which can be raised as invariants. Therefore, this section discusses the matter in order to clarify and

```
..bubbleSort() ::: EXIT
digits[] >= return[](lexically)
digits[] == orig(digits[])(lexically)
orig(length) == size(return[])
return !=null
digits[] elements <= return[orig(length)-1]
return[orig(length)-1] in digits[]
return[orig(length)-1] in return[]
return[] elements <= return[orig(length)-1]
orig(length) < digits[orig(length) -1]
orig(length) < return[orig(length) -1]
digits[orig(length)- 1] <= return[orig(length)- 1]
```

Figure 10. Related invariants to the bubblesort code of Figure 1 using original Daikon

evaluate our proposed algorithm. To perform this job, we present two analyses. Fist we compare the running-time and the time-order of the original Daikon with the modified one. Then we use *relevance* [8] to measure the quality of the produced invariants. The running times of the proposed modified Daikon and the original one in terms of millisecond is shown in the Fig.11. As it is understood from Fig.11 the time-order of both modified and original versions of Daikon are linear. It is also inferred that the original Daikon, as expected, does not excel at running-time compared to the modified Daikon. The higher number of variables, the higher slope in terms of time order. However, an increase in the number of variables does not change the time order in terms of the numbers of data trace-files [2].

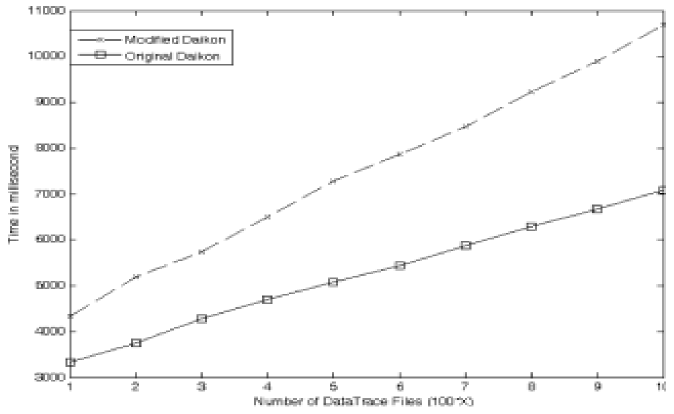

Figure 11. Time order of code of Figure 1 using different numbers of Data-trace files

From another perspective, the output of the modified Daikon over some typical programs is summarized in Table.1. As we discussed in section 6 we involve basic and small subprograms. We believe that every program, either big or small, has small parts and might be raised in small subprograms. These subprograms include arrays as their variables and effectively present the effect of the ideas.

	# of detected invariants	# of implied invariants	# of irrelevent invariants	# of prosper invariants
Delete one	50	5	4	41
element of array AVG	67	14	7	46
Replace	48	2	2	44
Mix	230	48	12	170

Table 1. Efficacy of modified Daikon in some case studies

Rows in Table.1 are representative of different sub-programs we discussed in previous sections and columns are number of different sorts of invariants. All the inferred invariants are not proper. In table.1 we proposed the number of implied and irrelevant invariant. For example if two invariants *X != 0* and *X in [7..13]* are determined to be true, there is no sense to report both because the latter implies the former.

8. Conclusion

As mentioned before, invariants attracted significant attention in software engineering in recent years. In this paper, different properties of a program code are checked at different program points. More useful properties lead to receive more valuable invariants and thereby lead to infer more prevalent faults. Hence we propose two properties which dramatically change positively the effect of invariants in the case of arrays. Since arrays and pointers are objects which are more probable to be faulty we add two useful properties to other invariants. As most of fault operating happens in the first and last elements of arrays we enhance the effect of fault detecting by employing these elements as some properties of the array. Another property which prepares a good condition to gain more useful invariants is the mutual element for same type arrays. As mentioned earlier, this property is helpful when in a program point an array is returned after changing elements in another array.

Although some ideas about arrays are valid in the case of pointers, some others inherently differ. For future work, the pointers can be dealt with in more details.

References

[1] Robert,W., Floyd. (1967). Assigning meanings to programs. *In:* Symposium on Applied Mathematics,p. 19–32. American Mathematical Society.

[2] Ernst, M. D., Cockrell, J., Griswold, W. G., Notkin, D. (2007). Dynamically discovering likely program invariants to support program evolution, *IEEE TSE* 27 (2) .

[3] Weiß, B. (2007). Inferring invariants by static analysis in KeY. Diplomarbeit, University of Karlsruhe.

[4] Neil, D. Jones and Flemming Nielson. Abstract interpretation: A semanticsbased tool for program analysis. *In:* S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, Handbook of Logic in computer Science, V. 4, p. 527–636. Oxford University Press, 1995.

[5] Boshernitsan, M., Doong, R., Savoia, A. (2006). From Daikon to Agitator: Lessons and challenges in building a commercial tool for developer testing, ISSTA.

[6] Hangal, S., Lam, M. S. (2002). Tracking down software bugs using automatic anomaly detection, *In:* ICSE.

[7] Csallner, C., et al. (2008). DySy: Dynamic symbolic execution for invariant inference, *In:* Proc. of ICSE.

[8] Michael, D. Ernst, Czeisler, AdamGriswold, William, G. Notkin,David . (2000). Quickly detecting relevant program invariants. In ICSE, Limerick, Ireland, June 7-9.

[9] Michael, Ernst, D., William, G. Griswold, Yoshio Kataoka, and David Notkin. (2000). Dynamically Discovering Program Invariants Involving Collections, Technical Report, University of Washington.

[10] Fouladgar, M. H., Minaei-Bidgoli, B., Parvin, H. (2011). Enriching Dynamically Detected Invariants in the Case of Arrays. International Conference on Computational Science and Its Applications (ICCSA 2011), LNCS, LNCS. Springer, Heidelberg, p. 622–632.